INVITED: A Modular Digital VLSI Flow for High-Productivity SoC Design

Brucek Khailany[†], Evgeni Krimer[†], Rangharajan Venkatesan[†], Jason Clemons[†], Joel S. Emer[†◇], Matthew Fojtik[†], Alicia Klinefelter[†], Michael Pellauer[†], Nathaniel Pinckney[†], Yakun Sophia Shao[†], Shreesha Srinath[‡], Christopher Torng [‡], Sam (Likun) Xi[*], Yanqing Zhang[†], Brian Zimmer[†]

[†]NVIDIA, [‡]Cornell University, [*]Harvard University, [◇]Massachusetts Institute of Technology

ABSTRACT

A high-productivity digital VLSI flow for designing complex SoCs is presented. The flow includes high-level synthesis tools, an object-oriented library of synthesizable SystemC and C++ components, and a modular VLSI physical design approach based on fine-grained globally asynchronous locally synchronous (GALS) clocking. The flow was demonstrated on a 16nm FinFET testchip targeting machine learning and computer vision.

KEYWORDS

High-Level Synthesis, VLSI Design, SoC Design, Machine Learning

1 INTRODUCTION

As Moore's law has provided an exponential increase in transistor density in SoCs, the unique features we are able to include in SoCs are no longer predominantly limited by chip area constraints. Instead new capabilities are increasingly limited by the engineering effort and team sizes associated with digital design and verification. As markets demand more performance, energy efficiency, and specialization, SoC design effort is increasing each generation.

Today's most complex SoCs contain billions of transistors and take thousands of engineer-years to implement. They are primarily designed using hand-coded Register Transfer Level (RTL) models with hierarchy and replication for managing complexity. RTL models are verified for functionality and performance against golden reference models using fast RTL simulators and a mix of constrained-random coverage-driven verification, directed tests, and formal techniques. For physical design, hierarchy is used to divide these large SoCs into manageable partition sizes that can be handled by floorplanning and place-and-route tools. However, inter-partition interfaces and timing closure for fully synchronous designs are still a challenge, particularly in the presence of on-chip variation. To address these challenges, we introduce two novel improvements to a digital SoC design flow that reduce the large engineering effort associated with implementing complex SoCs:

This research was, in part, funded by the U.S. Government, under the DARPA CRAFT program. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

DAC '18, June 24–29, 2018, San Francisco, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3199846

- *Object-Oriented High-Level Synthesis (OOHLS) based design*: We propose a C++ object-oriented library-based approach to digital design. The OOHLS approach includes a communication abstraction and SystemC implementation for latency-insensitive design [4]; *MatchLib*, a library of commonly used hardware components in SystemC and C++; and an HLS-based flow for synthesizing SystemC/C++ models to RTL.
- *Fine-grained Globally Asynchronous Locally Synchronous (GALS) Clocking*: We propose a GALS system to simplify hierarhical digital VLSI design. Per-partition clock generators and correct-by-construction top-level asynchronous interfaces eliminate top-level clock distribution and timing closure requirements without substantial area or latency penalties.

In this paper, we describe these innovations in more detail as part of an overall high-productivity C++-to-layout design flow. We also demonstrate the suitability of the design flow to a real chip by implementing a 87M transistor SoC testchip targeting machine learning and computer vision in 16nm FinFET technology.

2 FRONT-END DESIGN METHODOLOGY

2.1 Background

Several approaches have recently emerged to raise the level of design abstraction above RTL for reduced design effort. For example, Chisel [3] and PyMTL [9] let hardware designers code in Scala and Python by augmenting these programming languages with hardware-specific features. Bluespec [11] provides a HDL which uses triggered operations for sequencing and method-based module interfaces. Generator-based approaches, such as Genesis2 [13] or the Chisel Rocket Chip generator [2], reuse code by defining flexible templates for hardware blocks that the designer can customize using architectural parameters.

HLS has also emerged as another promising approach for raising the level of design abstraction [10]. HLS tools for ASIC design such as Catapult (Mentor Graphics) or Stratus (Cadence) take descriptions of hardware functionality in C, C++, or SystemC, then run compilation and scheduling transformations to map to an optimized RTL implementation. C++ can be coded in an untimed style and SystemC can be coded in a "loosely-timed" style, with HLS tools managing automatic pipelining and resource contention management, enabling design exploration tradeoffs without changing source code. Even with recent advancements in HLS tools, adoption at SoC design houses is still typically limited to simple feedforward pipelines, such as small blocks for image or signal processing functions. Connecting hardware blocks using memory or interconnect structures still typically relies on RTL scaffolding. More wide-scale

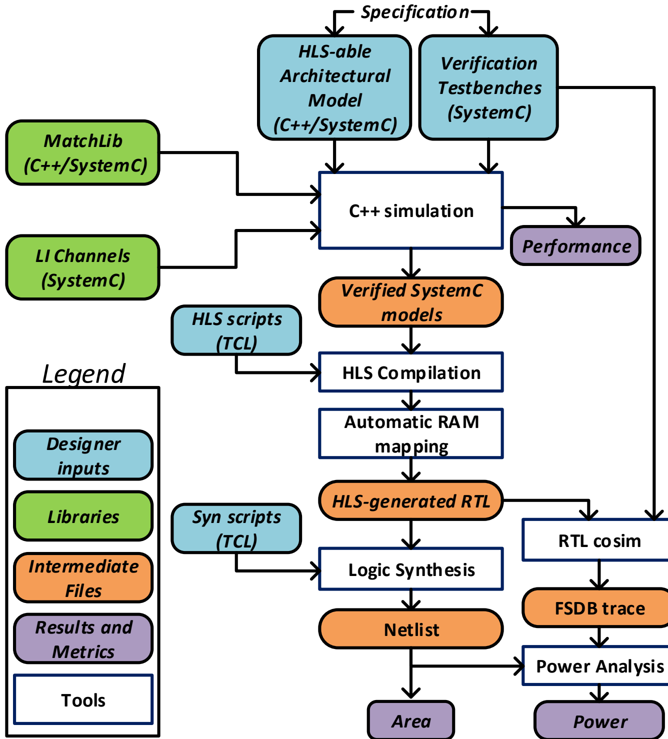

Figure 1: Proposed OOHLS design methodology

adoption of HLS has been limited due to concerns about Quality of Results (QoR), usability for complex pipelines and control structures, lack of formal C-to-RTL equivalence tools, and compatibility with typical design and verification flows that support coverage metrics and ECOs.

2.2 OOHLS Overview

Our goals with the development of an OOHLS flow were to leverage the familiarity of C++ and growing maturity of HLS tools, embrace library-based design for its reuse and modularity advantages, and address some of the shortcomings of existing HLS-based flows. Figure 1 shows an overview of our proposed front-end design methodology. Architectural models are coded using a mix of SystemC and C++ code synthesizable by HLS tools [14]. Testbenches are written using a mix of SystemC and C++ functions, but are free to use arbitrary C++ code including common libraries such as STL and Boost. The novel aspects from OOHLS are highlighted in green, where we provide the following contributions to reduce design and verification effort:

- *Latency-Insensitive Channels*: an on-chip communication abstraction and implementation for latency-insensitive design [4] compatible with HLS tools that enables a fast and accurate SystemC performance model.
- *MatchLib (Modular Approach To Circuits and Hardware Library)*: a well-maintained object-oriented library of commonly used hardware components synthesizable by HLS.

Once architectural models are developed with instantiated LI channels and MatchLib components, we use a more standard downstream flow. HLS tools run compilation, pipelining, and scheduling optimizations that map loosely-timed SystemC models to cycle-accurate RTL. Standard logic synthesis flows generate gate-level netlists and area estimates. Power analysis tools are used to estimate power dissipation.

One key advantage provided by OOHLS is support for native C++ simulation of synthesizable architectural models with high simulation speed and performance accuracy. Since C++ is already commonly used for architectural simulators and testbenches, designers are able to quickly ramp up on OOHLS compared to other approaches built on less commonly-used programming languages. OOHLS also enables code reuse of well-maintained libraries by leveraging C++ language support for modularity, encapsulation, and abstraction within MatchLib and LI channel implementations. Compared to IP reuse at the RTL level, modularity and reuse can be increased by using libraries at the C++ level. HLS tools allow the decoupling of functionality in the architectural models from design constraints and process information in HLS and synthesis scripts. This decoupling also enables design space exploration without changing source code or using generator-based approaches for optimal pipelining.

Encapsulation of high Quality of Result (QoR) C++ coding styles within MatchLib components also allows HLS-generated RTL to have area, power, and performance more competitive with hand-optimized RTL. While we do not expect state-of-the-art HLS tools to necessarily outperform well-tuned hand-written RTL, preliminary experiments across a range of datapath modules and small functional units show that comparable QoR (±10%) can be achieved through appropriate code optimizations and design constraints. With fixed design team resources and higher design team productivity, the proposed methodology can in fact lead to new architectural features being implemented that would have otherwise been excluded due to time constraints. More time can then be spent on researching and designing such new units or features in order improve the overall area, power, or performance of a product compared to a baseline flow using manual RTL design and verification.

2.3 Latency Insensitive channels

The ability to separately develop components within the proposed OOHLS flow is essential. The Latency-Insensitive (LI) design paradigm [4] proposes a methodology guaranteeing functional correctness of the system with arbitrary communication delays between components. As such, LI design is widely used in Networks-on-Chip (NoCs) and interconnect protocols such as the ARM Advanced Extensible Interface (AXI) [1]. The LI design paradigm can naturally be implemented using LI channels [6]. LI channels are a natural fit for inter-unit communication in an HLS-based environment [12, 15]. Since HLS tools automatically pipeline units to schedule the design under given constraints, unit latencies are not known a priori. LI channels provide the flexibility of connecting separately compiled units with arbitrary latency and throughput while maintaining functional correctness on inter-unit interfaces. LI channels also provide the extensibility of adding retiming registers on inter-unit interfaces to ease timing pressure or aid floorplanning. Moreover, the physical implementation of LI channels can include clock-domain crossing logic or even packetize/depacketize logic to send data between a producer and a consumer across a NoC.

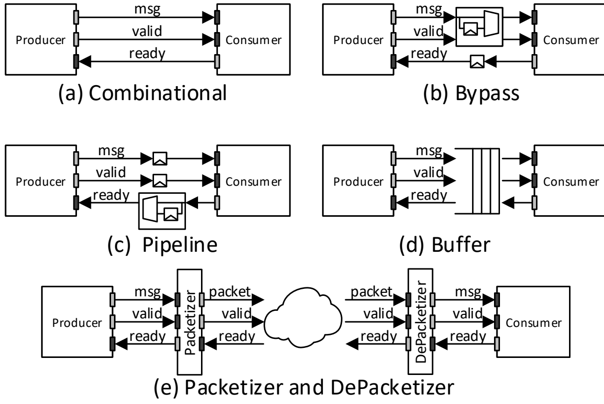

Figure 2: *Connections* channel implementations

To support LI channels within SystemC, we found the following design characteristics to be important. First, the API must support complex datatypes in C++ included templated structs. Second, for modularity, the implementation of channel ports within each unit should not need to modified for different channel properties (e.g. buffered versus combinational, blocking versus non-blocking interfaces). Third, implementations should support performance-accurate simulations in SystemC, where elapsed cycle counts during C++ simulation are similar to HLS-generated RTL for performance signoff. Finally, channel implementations should contain verification hooks for perturbing timing in order to simulate the effects of timing differences that will arise after HLS maps designs to RTL. After surveying existing SystemC communication libraries provided by HLS tool vendors, we determined that the above criteria were not satisified. Therefore, we developed a new C++ API and library called *Connections*. In this paper, we highlight three contributions of *Connections*:

- A novel API that decouples ports from channels.
- Introduction of a *sim-accurate* model that enables performance-accurate simulation.
- Enhanced verification support through stall injection capabilities in the channel enabling designers to verify hard-to-cover test cases.

Table 1: Proposed API of *Connections*, reflecting unified terminals (ports) and different kinds of channels

Port	Functions
In<T>	Pop(), PopNB()
Out<T>	Push(), PushNB()

Channel	Description
Combinational<T>	Combinationally connects ports
Bypass<T>	Enables DEQ when empty
Pipeline<T>	Enables ENQ when full
Buffer<T>	FIFO channel
Packetizer<T>, DePacketizer<T>	Network channels

The proposed API, summarized in Table 1, uses unified endpoint objects as input/output terminals (or ports) regardless of the connecting channel type selected later. Such polymorphic ports enable the functionality of individual components to be captured in a reusable library. *Connections* supports communication between components via a wide range of channel types ranging from combinational direct connections to packetizer/depacketizers used within a network as shown in Figure 2. Designers can choose the appropriate channel type depending on context. For example, the same on-chip memory module could be used in an accelerator as well as a many-core processor by connecting it using a dedicated combinational/buffered channel to meet the bandwidth requirements of the accelerator, while using an on-chip network to ensure the scalability of a many-core processor.

Connections also supports performance-accurate simulation natively in SystemC. *Connections* implements both a *signal-accurate* model used during HLS and a *sim-accurate* model used for SystemC performance simulations. The following code snippet of a producer executing a non-blocking *push* illustrates the differences:

```
valid.write(true);    //set valid bit
msg.write(bits);      //write data bits
wait();               //one cycle delay
valid.write(false);   //clear valid bit
success = ready.read();
```

In the code example, data bits are written to the channel when the valid signal is set. With the *signal-accurate* model, in the next cycle, the valid bit is cleared and ready bit is checked for success of the operation. A *wait* statement is used to describe the delay between set and clear operations of *valid* signal. Since the valid bit clear occurs a cycle later, we refer to it as a *delayed* operation. Similarly, a *pop* routine uses a *delayed* operation to clear the *ready* signal. In the case of multiple ports used in a single loop, all *wait* statements would be compiled and overlapped by the HLS scheduler. However, a SystemC simulator would execute these *wait* statements sequentially, which can at best result in an elapsed-cycles discrepancy and at worst result in functional errors or deadlocks.

In the *sim-accurate* model we provide an alternative implementation of these handshake routines using non-synthesizable SystemC mechanisms. We achieve accurate performance behavior in the *sim-accurate* model by avoiding handling *delayed* operations within the main execution thread. Instead, in the case of a producer, for a *push* operation, data is written into an output buffer and a helper thread is used to transmit data from all output buffers with valid data. Similarly, on the consumer side, a helper thread is responsible for receiving data. For a *pop* operation, data is consumed from an input buffer. Thus, the delayed operations of handling *ready* and *valid* signals are eliminated from the main thread of execution, allowing the model to maintain cycle accuracy.

Figure 3 depicts measured cycles per transaction in an arbitrated crossbar design with a varying number of ports using different approaches. For the *signal-accurate* model, since the source of error for measured cycles is in IO port routines, the error grows with the number of ports. However, the *sim-accurate* model matches measured RTL throughput for all configurations.

Connections also contains built-in support for assisting functional verification of SystemC models. System-level verification is a challenging task, requiring rigorous examination of all possible timing interactions between different blocks. Leveraging the advantages of LI design, we add an option to inject random stalls into any channel by randomly withholding *valid*. Using this technique, modified timing of unit interactions can be created without changing

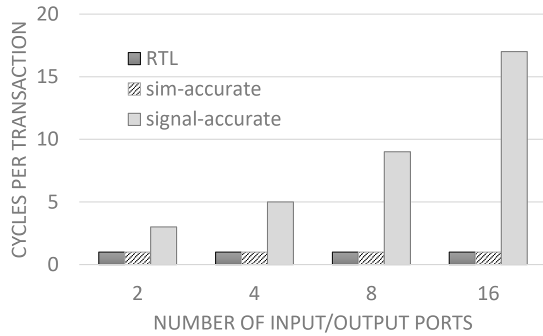

Figure 3: Simulated SystemC cycles per transaction for an arbitrated crossbar with varying number of ports

any design or testbench code. Such testing assists in quickly covering complex corner case scenarios that otherwise would require significant dedicated test development effort.

2.4 MatchLib

Inspired by object-oriented libraries in software engineering, we envision that a well-maintained library of commonly used hardware components will enable significant design productivity benefits. MatchLib[1] is the proposed hardware library for implementing this OOHLS methodology vision. A list of currently implemented components is presented in Table 2. All components are classified as either *C++ functions*, *C++ classes*, or *SystemC modules*. C++ functions are untimed, process one or more inputs, and produce one or more outputs. They are generally used to describe datapath functions and are often called from other parts of MatchLib. C++ classes are objects that include state and untimed methods or functions that read and/or update the state. For example, the mem_array class includes an array of data as internal state with read and write methods for accessing or updating the state. Similarly, the arbiter includes state for storing priorities and a *pick* method for selecting among its inputs and updating its state. Finally, SystemC modules are used for hardware primitives with feedback or clocked behavior more easily described within clocked processes. SystemC modules are coded in a loosely-timed manner where details of hardware pipelining can be left to the HLS tool.

One advantage provided by OOHLS is simple abstractions for common functions and data structures that enable designers to easily achieve good QoR compared to naively written synthesizable C++ code. MatchLib's preoptimized component implementations allow designers to focus on higher level algorithmic decisions. As an example, we discuss a case study of a crossbar implementation. Consider a N-element crossbar modeled in C++ as a permutation of data elements between an input and output array. A naive approach would loop over elements in the input array and write elements to the output array using the following implementation. In this case, *src* is a loop index, *dst* is a N-element array indicating which output each *src* input should be routed to, and *out* is an output port with a write method. We refer to this as a *src-loop* implementation:

```
for(int src=0;src<LANES;++src) {
  out[ dst[src] ].write( in[src] ); }
```

[1]MatchLib is currently shared with a limited audience of collaborators from industry and academia. As part of our future work, we are considering making some or all of the library available as an open source project.

Table 2: MatchLib components

	Component	Description
C++ functions	Float	Floating-point arithmetic functions (mul, add, mul-add)
	Crossbar	N-to-N switch w/ configurable bitwidths
	Encoder/Decoder	1-hot encoders and decoders
C++ classes	FIFO	Configurable FIFO
	Arbiter	1-out-of-N round-robin selector
	Mem_array	Abstract memory class
	Vector	Vector helper container w/ vector operations
	Connections	Modular IO supporting LI channels
	Arbitrated Crossbar	Crossbar w/ conflict arbitration & queuing
	Arbitrated Scratchpad	Banked memories w/ arbitration & queueing
	Reorder Buffer	Queue w/ in-order reads, out-of-order writes
SystemC modules	Serializer/Deserializer	N-bit packets to/from M cycles of (N/M)-bit packets
	Cache	Configurable linesize, capacity, associativity
	Scratchpad	Banked memory array with crossbar
	SFRouter	Store-and-Forward NoC router
	WHVCRouter	Wormhole NoC router with virtual channels
	AXI Components	Master/Slave Interfaces & bridges for AXI interconnect

Alternatively, in the following example, *dst* is a loop index and *src* is a N-element array indicating which input each output should be routed from. We refer to this as a a *dst-loop* implementation:

```
for(int dst=0;dst<LANES;++dst) {
  out[ dst ].write( in[ src[dst] ] ); }
```

Naively, both C++ implementations may appear to have identical behavior with a similar number of array accesses and complexity. However, HLS can generate significantly different RTL. The *src-loop* implementation requires priority decoders to control the generated multiplexers, since multiple inputs could be selecting the same output destination and the higher index would have priority. This design creates an undesirable dependency path from all *dst[src]* signals to all outputs. In comparison, the *dst-loop* implementation will only have a path from each *src[dst]* control input to each corresponding output port. Moreover, since the *dst-loop* implementation has fewer operations that must be scheduled after loop unrolling by HLS, significantly shorter compilation times and better scalability to larger N is observed.

Experimenting with a 32-lane 32-bit crossbar, we measured a 25% area penalty for the *src-loop* implementation over the *dst-loop* implementation in Catapult HLS. Although this example is trivial, we have observed similar behavior across many components from Table 2. Our results emphasize the advantage of encapsulating efficient C++ coding-style optimizations within MatchLib for achieving good QoR from HLS tools.

3 BACK-END DESIGN METHODOLOGY

After running through the front-end flow shown in Figure 1, gate-level netlists can be mapped to layout using standard back-end tool flows (floorplanning, automatic place-and-route, clock-tree syntheis, static timing analysis, and physical verification signoff). To manage the large design sizes and replication typical of today's large SoCs, unit-level netlists are grouped into a hierarchy of partitions. Partitions are composed of place-and-routed standard cells and islands of macro blocks such as SRAM for caches. Splitting large designs into smaller partitions can make back-end tool flows manageable, reduce runtime, enable physical reuse, and allow design teams to parallelize their work on physical design closure for each partition. However, with these productivity advantages come

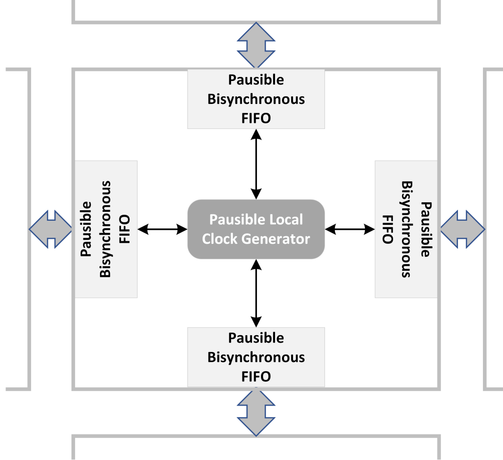

Figure 4: Fine-grained GALS pausible clocking

tradeoffs such as more top-level timing signoff on inter-partition interfaces and top-level clock distribution challenges.

3.1 Fine-grained GALS Clocking

To close timing on inter-partition interfaces, it is typical to use synchronous clocking from a single on-chip or off-chip clock source. This process involves both global clock distribution for sending a balanced clock to each partition with additional delay elements for fine tuning. Synchronous timing paths which span multiple partitions must close timing across many setup and hold corners, which can be challenging in the presence of on-chip variation.

As an alternative, we propose a novel clocking approach based on fine-grained GALS clocks, shown in Figure 4. Each partition has its own self-contained small local clock generators. Local adaptive clock generators are able to better track local power supply noise [7] to reduce design margin. All communication between partitions pass through Pausible Bisynchronous FIFOs [8]. These FIFOs allow low-latency, error-free clock domain crossings that work by integrating the synchronizers and clock generators. With this clocking scheme, the complexity of top-level clock distribution is completely eliminated from the SoC. Furthermore, correct-by-construction top-level timing can be achieved with asynchronous handshake signals passing between partitions, greatly simplifying timing closure effort compared to synchronous interfaces. In our implementation, all asynchronous interfaces are implemented as LI channels and can interface with Connections ports from HLS-generated RTL. Although we incur a small area penalty for local clock generators and pausible bisynchronous FIFOs, we estimate this overhead to be less than 3% for typical partition sizes.

4 CASE STUDY: PROTOTYPE SOC

To study the productivity benefits of our methodology on a real design, we developed an 87M-transistor Machine Learning (ML) prototype SoC using OOHLS and fine-grained GALS clocking. The design process helped drive the flow development and provided a testbed for experimenting with performance modeling, fine-grained GALS, and benefits to productivity. Table 3 lists the EDA tools used for the project. The design was implemented in a TSMC 16nm FinFET technology node with a signoff clock frequency of 1.1 GHz in a typical process corner, typical voltage, and high temperature.

Table 3: Design Tools

HLS compiler	Mentor Graphics Catapult HLS v10.0a
C++ Coverage tool	Testwell CTC++ 8.1
Verilog simulator	Synopsys VCS mx-2015.09
Logic Synthesis	Synopsys Design Compiler Graphical v2013.12
Place and Route	Synopsys ICC2 v2014.12

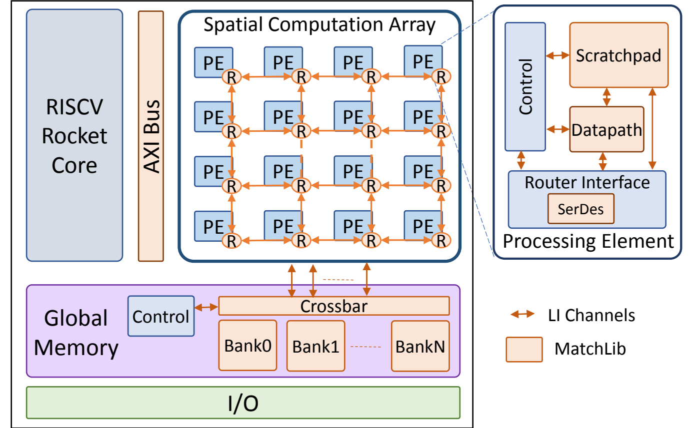

Figure 5: Prototype SoC

Figure 5 shows the overview of the prototype SoC. It includes a RISC-V processor [2], a ML accelerator similar to a more programmable Eyeriss [5], and a banked on-chip global memory. The programmable ML accelerator supports applications such as convolutional neural networks, K-means clustering, and other image processing workloads. It consists of a spatial array of processing elements (PEs), connected via a dedicated Network-on-Chip (NoC). Each PE contains a scratchpad, vector datapath unit, control unit, and router interface logic. Each PE is programmed to support execution of different compute kernels such as vector multiply, dot-product, and reduction. The RISC-V processor acts as a global controller, initiating the execution by configuring the control registers in PE and global memory and orchestrating the data transfer across different levels in memory hierarchy—(i) PE scratchpad, (ii) global memory, and off-chip. The prototype chip is attached to a daughtercard, which is connected to an off-the-shelf FPGA prototyping system attached via PCI to a PC for testing and demonstration.

OOHLS Design: Using the OOHLS methodology (Figure 1), we developed synthesizable SystemC architecture models for all components of the prototype SoC (except for the Chisel-generated RISC-V processor Verilog) and corresponding testbenches in SystemC. Many units of the prototype SoC were designed by directly instantiating the MatchLib components shown in orange in Figure 5, while Connections and various MatchLib helper functions and class containers were used throughout the design. In the PE, we used the MatchLib vector library to design the datapath unit, Serializer/Deserializer in the router interface, and arbitrated scratchpad from MatchLib to design the scratchpad memory. In the Global Memory, the different memory banks were designed using our abstract memory class, mem_array, and were connected to the multiple input/output ports using the MatchLib crossbar. For communication, we used the Connections implementation of LI channels to interconnect different units inside the PE and the Global Memory, and used WHVCRouter to design the NoC.

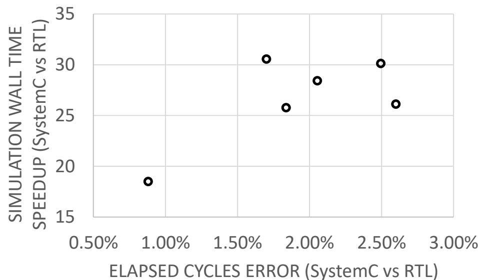

Figure 6: Performance accuracy of SoC-level tests

Back-end Design: For VLSI implementation, we split the design into five unique digital partitions: 15 replicated PEs, 1 left Global Memory, 1 right Global Memory, 1 RISC-V, and 1 I/O partition. Fine-grained GALS was implemented with a local clock generator and a NoC router per partition. Asynchronous router-to-router interfaces included a pausible bisynchronous FIFO for low-latency error-free crossing. With the small partition sizes and fine-grained GALS approach, we were able to implement a 12-hour RTL-to-layout turnaround time. This enabled dozens of daily iterations during the march-to-tapeout phase of the project to tweak floorplans, timing constraints, and tool settings until the design converged.

Verification: OOHLS benefited verification productivity in multiple ways. Reuse of pre-verified MatchLib components allowed us to focus our effort on integration and architectural verification. Stall injection mechanisms in Connections helped quickly find timing-interaction corner cases. C++ models and advanced debug support in MatchLib helper functions enabled fast debug by quickly locating bugs. Standard C++ code coverage tools were used to identify test coverage holes. RTL simulations were primarily used to test scaffolding around HLS-generated blocks such as clock domain crossing FIFOs and sanity-check full-system simulations. Design and verification effort was tracked closely during the project. We estimate that by leveraging OOHLS, we were able to achieve a productivity of between 2K-20K gates (NAND2 equivalents) per engineer-day on *unique* unit-level designs, estimated to be significantly higher than a baseline RTL-based design methodology. Future work is needed to study quantitative improvements in OOHLS design productivity versus contemporary approaches, since it is challenging to compare across different designs or engineering teams.

Performance Modeling: We primarily verified SoC performance using the sim-accurate model in Connections. This approach has several benefits: (i) it eliminates the need to maintain a separate architectural performance model, thereby reducing design effort; (ii) it accurately captures the effect of data and control flow dependencies on performance; and (iii) it enables significant simulation speed-up compared to cycle-accurate RTL. Figure 6 shows the accuracy of our performance model for six SoC-level tests. Tests were run on both SystemC and HLS-generated RTL models, comparing elapsed cycles and wall-clock runtime. For each test, SystemC speedup (with the sim-accurate LI channels model) versus RTL is shown on the Y axis. The X axis shows the relative difference in elapsed cycles. We observed a $20-30\times$ wall run time reduction when using the SystemC-based performance model with performance inaccuracy below 3%. We attribute the innacuracies to unit pipeline latencies not included in the SystemC models.

CONCLUSIONS

Two novel design methodology innovations for digital SoCs were described: An object-oriented HLS-based design flow and a fine-grained GALS clocking methodology. These innovations were integrated as part of an overall C++-to-layout end-to-end design flow and applied to an 87M-transistor prototype SoC for machine learning. As part of our future work, we intend to continue to improve and enhance OOHLS-based design and MatchLib and apply the SoC design flow to future SoC development projects.

ACKNOWLEDGMENTS

The authors thank additional members of the NVIDIA team who contributed to the architecture and testing of the RC17 testchip, including Bill Dally, Christopher Fletcher, Stephen Keckler, Angshuman Parashar, and Stephen Tell. We also thank Ben Keller, Ziyun Li, Antonio Puglielli, and Gopalakrishnan Srinivasan, who contributed to design methodology research during their internships at NVIDIA. We thank Bryan Bowyer and the Catapult engineering team at Mentor Graphics who helped provide useful feedback about LI channels and MatchLib. Finally, the authors would like to thank our collaborators at Harvard University and sponsors at DARPA for many insightful discussions and feedback about OOHLS and MatchLib under the CRAFT program.

REFERENCES

[1] AMBA ARM. 2010. Axi protocol specification (rev 2.0). (2010).
[2] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, and J. Koenig. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. University of California at Berkeley.
[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC'12.* 1212–1221.
[4] L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli. 1999. A methodology for correct-by-construction latency insensitive design. In *DAC'99.* 309–315.
[5] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. 2016. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *Proceedings of the International Solid State Circuits Conference.* 262–263.
[6] K.E. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J.S. Emer. 2012. Leveraging latency-insensitivity to ease multiple FPGA design. In *Proceedings of the ACM/SIGDA 20th International Symposium on Field Programmable Gate Arrays (FPGA).* 175–184.
[7] D.A. Kamakshi, M. Fojtik, B. Khailany, S. Kudva, Y. Zhou, and B. Calhoun. 2016. Modeling and Analysis of Power Supply Noise Tolerance with Fine-grained GALS Adaptive Clocks. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC).*
[8] B. Keller, M. Fojtik, and B. Khailany. 2015. A Pausible Bisynchronous FIFO for GALS Systems. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC).* 1–8.
[9] D. Lockhart, G. Zibrat, and C. Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Proceedings of the International Symposium on Microarchitecture (MICRO).* 280–292.
[10] M.C. McFarland, A.C. Parker, and R. Camposano. 1990. The high-level synthesis of digital systems. *Proc. IEEE* 78, 2 (1990), 301–318.
[11] R.S. Nikhil and K.R. Czeck. 2010. *BSV by Example: The Next-generation Language for Electronic System Design.* Bluespec.
[12] L. Piccolboni, P. Mantovani, G. Guglielmo, and L.P. Carloni. 2017. COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators. *ACM TECS* 16, 5s (2017), 150.
[13] O. Shacham, M. Wachs, A. Danowitz, S. Galal, J. Brunhaver, W. Qadeer, S. Sankaranarayanan, A. Vassiliev, S. Richardson, and M. Horowitz. 2012. Avoiding game over: Bringing design to the next level. In *DAC'12.* 623–629.
[14] SystemC Synthesis Working Group. 2016. *SystemC Synthesis Subset Standard v1.4.7.* Technical Report. Accellera.
[15] R. Zhao, G. Liu, S. Srinath, C. Batten, and Z. Zhang. 2016. Improving high-level synthesis with decoupled data structure optimization. In *DAC'16.* 137–142.