Illinium: Accelerator for RTL Emulation

Peter Lee *wonjong3* Reet Sinha *reet2* Kyle Jones *kylej6* Thomas Gornet *tgornet2* Yehya Sleiman Tellawi *yehyas2* Steffen Brown *sbrown16*

***Abstract*—Illinium is a simplified processor-based RTL emulation accelerator that explores a middle ground between CPU simulation and FPGA prototyping. Rather than mapping logic directly into reconfigurable fabric, Illinium maps synthesized RTL into instruction streams executed by many small Boolean processing cores connected through a lightweight on-chip communication network.**

I. Problem Statement

RTL emulation is a fundamental step in digital design verification, yet it remains constrained by a tradeoff between flexibility and performance. CPU-based approaches enable fast compilation and iterative debug, but struggle to efficiently exploit the fine-grained parallelism inherent in Boolean computation. FPGA-based solutions improve throughput through spatial execution, but introduce long compile times and reduced iteration agility.

As design sizes and simulation lengths continue to grow, these limitations become increasingly significant, particularly for cycle-based workloads with complex dependency patterns and frequent state synchronization. This gap highlights the need for compute models that can better expose fine-grained parallelism while maintaining rapid iteration, motivating exploration of architectural approaches that move beyond traditional CPU and FPGA paradigms.

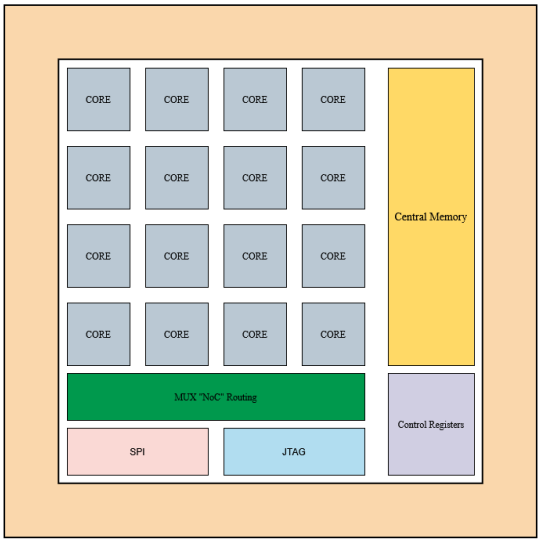

Fig. 1. Chip Conceptual Design

II. Architecture

At a system level, Illinium consists of an array of Boolean processor cores, small core clusters, a hierarchical communication fabric, a centralized SRAM structure, and a host-facing control/debug interface. The compiler maps a synthesized RTL design onto this substrate by partitioning the netlist, assigning logic to cores, and generating scheduled instruction streams and routing metadata. During execution, the cores evaluate combinational logic, exchange intermediate values through the interconnect, and update sequential state in a cycle-accurate manner.

A. Boolean Processor Cores

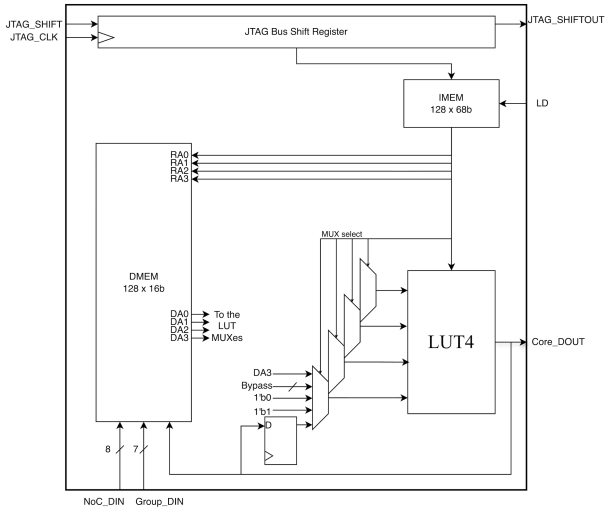

Fig. 2. Single Boolean Core

Each Boolean processor core is a compact, instruction-driven compute element built around a LUT4 execution unit, local instruction memory, and local data memory. The instruction memory stores the statically scheduled operations assigned to the core, while the data memory stores Boolean state, intermediate values, and communicated operands. For each operation, the core reads selected data memory entries, uses instruction-controlled bit selection to extract the required input bits, and routes them into the LUT4 to evaluate the desired Boolean function. This maps to the LUT4 netlists produced by the compiler flow.

This organization enables time multiplexing of logic operations rather than dedicating hardware to every node in the DUT, a single core steps through its instruction stream and reuses the same LUT4 and local storage across cycles to execute many scheduled operations. To support parallel execution across many cores, the compiler may insert delay slots so communicated values arrive at the correct time and

dependent operations remain synchronized. The local data memory accepts inputs from both the NoC and the core group, allowing produced values to be written into local state and consumed by later operations. In this way, the core can communicate with neighboring and remote cores while maintaining a simple local execution model. The JTAG interface is used at startup to program the instruction memory and initialize the core before execution begins.

B. Interconnect

Illinium employs a mux-based interconnect to route signals between processor cores, local data memories, and a centralized SRAM. Instead of relying on a dynamic NoC, the architecture uses a deterministic, statically compiled communication scheme.

The system is organized as a hierarchial communication fabric. Processor cores are grouped into a small clusters of 8 cores, where communication is handled through local sharing or broadcast paths. Communication between clusters is performed via a higher level mux network. As inter-cluster communication is based on deep pipelined muxes across longer distances, which results in higher hop latency and bandwidth overhead/constraints.

In a 32-core chip configuration, there are 4 clusters, each producing 8 bits of output every cycle. This results in total of 32 bits of output per cycle across the chip. Each cluster then selects (1) one bit each from the 8-bit output of 3 other clusters, and (2) one bit from the 8-bit output of the centralized memory. This hierarchical organization supports scaling to higher cluster counts.

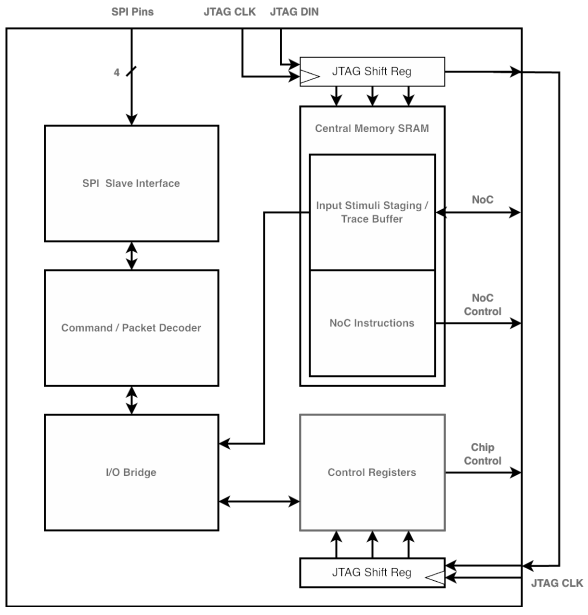

Fig. 3. Central Memory and Control I/O

C. I/O Interface & Initialization

Illinium's I/O interface utilizes the Joint Test Action Group (JTAG) and Serial Peripheral Interface (SPI) protocols to enable communication between itself and a host device. The interface serves five purposes: System configuration, instruction/data initialization, debugging, and runtime communication.

JTAG is Illinium's debug interface, giving the user complete access to all control registers, as well as the per core and central on-chip SRAM. In addition to live debugging, JTAG will enable the deployment of designs to the chip. During the programming of Illinium, the JTAG interface will load the instructions for each core into the local SRAM and load initial stimulus and routing instructions into Illinium's central memory. During initial silicon bring up, JTAG will be used to apply test cases and input vectors into the design, collecting results, and communicating these with a host for comparison.

SPI will enable runtime communication with Illinium by a user. As opposed to JTAG, SPI will provide a higher speed interconnect between the host and the chip. This communication link will be used for offloading results of the emulation to the host, as well as runtime configuration changes. SPI will be provided with a set of memory mapped registers for read and write operations to configure chip settings, as well as full read and partial write access to Illinium's central memory.

D. Compilation / Static Scheduling

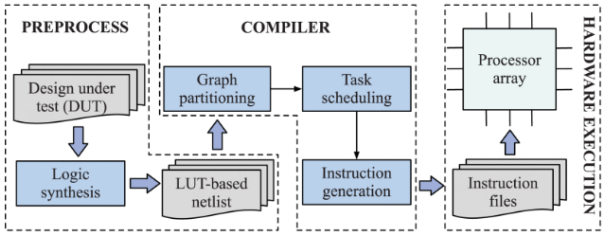

Fig. 4. Overall boolean processor based emulation flow, (from [4]).

Largely, the overall flow of the processor-based emulation system, like Illinium, is similar to the flow of an FPGA. One major difference is that there is no place-and-route stage that physically maps synthesized logic into the FPGA fabric, as physical wires are replaced with data movement among the processors. The typical flow is illustrated in figure 3.

We plan to use Yosys [2] to synthesize a SystemVerilog design into a LUT4 netlist. The netlist is then processed and partitioned using an existing partitioner like Mt-KaHyPar [5]. The partitioner will treat the netlist as a directed-acyclic hypergraph (DAH) and split the DUT netlist into subgraphs while optimizing for the number of cut edges to minimize communication across remote processor groups.

The scheduler takes the partitioned netlists and creates a stream of instructions to emulate the DUT. As emulation performance is largely dependent on the slowest combinational path (i.e., the critical path), the scheduler must prioritize the critical path for optimality. Another thing to consider is that the results of some LUT4 operations are used much later in the emulation step, meaning their communication latency can still be hidden if we defer it to much later. Hence, utilizing

ASAP and ALAP scheduling techniques is also very important in achieving good performance.

Currently, we have a compiler that takes in any arbitrary combinational/sequential SystemVerilog module and emits instruction streams for each processor. The compiler is aware of the exact constraints of the system (number of processors, bandwidth, and latency from source to destination). The system configuration for the compiler is parameterizable, so we expect the compiler to still be functional when the number of processors, bandwidth of ingress and egress, and latency between hops change as we further develop the architecture and design throughout the project. The scheduler is also verified through logical-equivalence checking using a simple simulator of the system emulating many designs (e.g., 16-bit counter, pipelined ALU, and multicycle RISC-V CPU).

III. Roadmap

Illinium's pre-silicon development will proceed in three largely sequential but overlapping phases, with significant time devoted to architectural exploration and simulation due to the need for iterative refinement in a novel small-scale chip design.

A. Phase I - Architecture (May - Jun.)

In Phase I, we will finalize the high-level architecture, including the processor microarchitecture, memory organization, interconnect structure, and host interface. During this stage we will evaluate key architectural parameters such as the number of cores, the size of local and central memories, and the latency/bandwidth tradeoffs of the communication fabric. We will also finalize the cluster-level organization of the Boolean processors so that the compiler, memory model, and routing fabric are all defined against a consistent hardware hierarchy.

B. Phase II - RTL, Simulation, DV (Jul. - Sept.)

In Phase II, we will implement the RTL for the selected architecture and develop the supporting verification environment. This phase includes unit-level verification of the processor core, memory structures, and interconnect, followed by subsystem integration and full-chip simulation. We will validate correctness using directed tests and end-to-end compiler-generated workloads, with emphasis on cycle accuracy and the handling of small inferred memories.

C. Phase III - PD, Compiler/Embedded (Oct. - Nov.)

In Phase III, we will close the loop between hardware and software by preparing the design for physical implementation, while maturing the compiler and host-side tooling. This includes synthesis, place-and-route, timing closure, GDSII file generation.

IV. Team Composition

Team members with diverse VLSI backgrounds are assigned default task areas, but responsibilities flex across categories as project needs evolve.

Category	Coordinators
Architecture/RTL	Steffen, Yehya, Peter
Physical Design	Kyle, Reet, Peter
Design Verification	Thomas, Reet
Compiler & Embedded	Steffen, Peter, Kyle
Hardware Tooling	Thomas, Peter

Below is a roster of our team, detailing applicable qualification for this project. We anticipate that a majority of our members will continue on this project in the spring to see through the bring-up process.

- **Peter** - Senior in Computer Engineering
  - Relevant Coursework: ECE 411, ECE 425, CS 533
  - Academic Experience: ECE 385 CA, ECE 411 CA, Ugrad. researcher with Prof. Rakesh Kumar
  - Industry Experience: Multi-chiplet NPU bringup, SoC NoC verification, CPU DV at Apple
  - Technical Strength: Verification, RTL, Physical Des., Prototyping
- **Reet** - Junior in Electrical Engineering
  - Relevant Coursework: ECE 411, ECE 425, ECE 342
  - Academic Experience: ECE 385 CA
  - Industry Experience: GPU DV Intern at Apple
  - Technical Strength: Verification, RTL, Physical Des.
- **Kyle** - 2nd-Year M.S. ECE
  - Relevant Coursework: ECE 411, ECE 425, ECE 511
  - Academic Experience: Advised by Prof. N.S. Kim
  - Industry Experience: Firmware Des. Engr Intern at Intel, EE Intern at Aeronix/Peraton/L3Harris Tech
  - Technical Strength: Physical Design, RTL, PCB Des.
- **Thomas** - Senior in Computer Engineering
  - Relevant Coursework: ECE 411
  - Industry Experience: GPU DV Intern at Apple
  - Technical Strength: Verification, RTL, Architecture
- **Yehya** - 2nd-Year M.S. ECE
  - Relevant Coursework: ECE 411, ECE 511
  - Academic Experience: Advised by Prof. Kumar, ECE120 head TA
  - Technical Strength: RTL, Architecture
- **Steffen** - 1st-Year M.S. ECE
  - Relevant Coursework: ECE 411
  - Academic Experience: ECE 411 Course Staff
  - Industry Experience: HW Validation Intern at Annapurna Labs (AWS), Embedded Engr. Intern at Textron System
  - Technical Strength: Architecture, RTL, Embedded

References

[1] W. Feng, Y. Zhang, Z. Wang, Z. Wang, Y. Wang, P. Ma, and N. Xu, "CCSS: Hardware-accelerated RTL simulation with fast combinational logic computing and sequential logic synchronization," *arXiv preprint arXiv:2507.08406*, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.08406

[2] YosysHQ, "Yosys Open SYnthesis Suite," GitHub repository, Mar. 25, 2026. [Online]. Available: https://github.com/YosysHQ/yosys

[3] R. Pu, Y. Sun, P.-H. Ho, F. Yang, L. Shang, and X. Zeng, "Sphinx: A Hybrid Boolean Processor-FPGA Hardware Emulation System," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Oct. 2023, pp. 1–9. doi: 10.1109/ICCAD57390.2023.10323694

[4] Z. Wang *et al.*, "ParSGCN: Bridging the Gap Between Emulation Partitioning and Scheduling," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 44, no. 3, pp. 1180–1192, Mar. 2025. doi: 10.1109/TCAD.2024.3453199

[5] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag, "Scalable Shared-Memory Hypergraph Partitioning," *arXiv preprint arXiv:2010.10272*, Oct. 2020. doi: 10.48550/arXiv.2010.10272