INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2025, VOL. 71, NO. 4, PP. 1–7
Manuscript received May 13, 2025; revised October 2025. doi: 10.24425/ijet.2025.156531

PyHLS: Intermediate Representation for Versatile High-Level Synthesis

Radoslaw Cieszewski

***Abstract*—High-Level Synthesis (HLS) has become an established methodology to accelerate the development of FPGA-based systems by allowing algorithms to be written in high-level languages (HLLs) such as C/C++ or Python. Yet, for real-time physics experiments—including fusion plasma diagnostics, high-energy physics (HEP) detectors, and rare-event astrophysical triggers—conventional HLS still falls short in three essential aspects: *determinism*, *portability*, and *auditability*. Pragmas embedded in HLL code blur the separation between algorithmic intent and implementation details, coupling scientific software to a particular device or compiler version. This is particularly problematic in long-lived scientific projects such as ITER or the Pierre Auger Observatory, where systems must remain functional and maintainable over decades [4]–[6].**

To address these challenges, we propose an Intermediate Representation (IR)-centric HLS flow—PyHLS—that explicitly introduces an abstraction layer between algorithm and Register-Transfer Level (RTL) design. The IR centralizes all performance-critical aspects: timing contracts (initiation interval, latency, jitter), concurrency (loop unrolling, pipelining), memory layout (banking, tiling, port allocation), and resource binding (DSPs, BRAMs, AI tiles). In this model, the algorithm is expressed in clean, testable Python code [1], [2], while device-specific optimizations are described in a structured IR graph. This IR is then lowered into a reusable VHDL *microinstruction library* [3], which serves as a portable middle layer across devices. By versioning and auditing IR graphs and instruction streams, PyHLS ensures reproducibility and traceability—critical properties in scientific computing where results must be verifiable years after deployment.

The methodology builds upon earlier work in Python-based high-level synthesis, parameterizable metamodels, and algorithmic synthesis with multi-level compilers [8], [9], [11]. It incorporates systematic design space exploration (DSE), allowing parameter sweeps over IR attributes and early feasibility checks. The flow is complemented by a cycle-accurate microinstruction emulator, which validates both functionality and timing contracts before vendor toolchains are invoked, reducing iteration time and catching infeasible designs early.

We demonstrate the motivation and applicability of this approach in two demanding domains. First, in plasma diagnostics at JET/ITER, where spectrometer and data acquisition systems must combine high bandwidth with deterministic latency [6]. Second, in trigger pipelines for astroparticle physics, where artificial neural networks (ANNs) and fuzzy-logic algorithms have been implemented directly in FPGA logic to discriminate rare events from large backgrounds [4], [5]. These use-cases highlight the need for explicit IR-level contracts and modularity: the same high-level algorithm must be portable across device generations, yet adapted to exploit specialized hardware resources such as DSP slices, systolic AI engines, or high-bandwidth memories.

The contribution of this work is therefore threefold:

1) **We formalize the role of an explicit IR in HLS, decoupling algorithms from implementation decisions and introducing contract-driven determinism.**
2) **We present a reusable VHDL microinstruction library and emulator that stabilize implementation and provide auditable artifacts.**
3) **We show how PyHLS extends naturally to heterogeneous FPGAs, mapping operators to emerging AI/ML blocks while maintaining scientific reproducibility and portability across decades.**

By unifying algorithmic specification, IR-based parameterization, and reusable microinstructions, PyHLS establishes a sustainable methodology for real-time physics experiments and beyond. In short: *write the science once, retarget the hardware many times*.**High-Level Synthesis (HLS) has become an established methodology to accelerate the development of FPGA-based systems by allowing algorithms to be written in high-level languages (HLLs) such as C/C++ or Python. Yet, for real-time physics experiments—including fusion plasma diagnostics, high-energy physics (HEP) detectors, and rare-event astrophysical triggers—conventional HLS still falls short in three essential aspects: *determinism*, *portability*, and *auditability*. Pragmas embedded in HLL code blur the separation between algorithmic intent and implementation details, coupling scientific software to a particular device or compiler version. This is particularly problematic in long-lived scientific projects such as ITER or the Pierre Auger Observatory, where systems must remain functional and maintainable over decades [4]–[6].**

To address these challenges, we propose an Intermediate Representation (IR)-centric HLS flow—PyHLS—that explicitly introduces an abstraction layer between algorithm and Register-Transfer Level (RTL) design. The IR centralizes all performance-critical aspects: timing contracts (initiation interval, latency, jitter), concurrency (loop unrolling, pipelining), memory layout (banking, tiling, port allocation), and resource binding (DSPs, BRAMs, AI tiles). In this model, the algorithm is expressed in clean, testable Python code [1], [2], while device-specific optimizations are described in a structured IR graph. This IR is then lowered into a reusable VHDL *microinstruction library* [3], which serves as a portable middle layer across devices. By versioning and auditing IR graphs and instruction streams, PyHLS ensures reproducibility and traceability—critical properties in scientific computing where results must be verifiable years after deployment.

The methodology builds upon earlier work in Python-based high-level synthesis, parameterizable metamodels, and algorith-

This work was supported by the grant No.504/05033/1033/43.022504 "Research and Development of Metamodeling and High-Level Synthesis (HLS) Methods: Design of Low-Latency Physics Triggers Based on Neural Networks Implemented in FPGAs", financed from the internal science fund of Warsaw University of Technology.

R. Cieszewski is with Faculty of Electronics and Information Technology, Institute of Electronics Systems, Warsaw University of Technology, Poland (e-mail: radoslaw.cieszewski@pw.edu.pl).

mic synthesis with multi-level compilers [8], [9], [11]. It incorporates systematic design space exploration (DSE), allowing parameter sweeps over IR attributes and early feasibility checks. The flow is complemented by a cycle-accurate microinstruction emulator, which validates both functionality and timing contracts before vendor toolchains are invoked, reducing iteration time and catching infeasible designs early.

We demonstrate the motivation and applicability of this approach in two demanding domains. First, in plasma diagnostics at JET/ITER, where spectrometer and data acquisition systems must combine high bandwidth with deterministic latency [6]. Second, in trigger pipelines for astroparticle physics, where artificial neural networks (ANNs) and fuzzy-logic algorithms have been implemented directly in FPGA logic to discriminate rare events from large backgrounds [4], [5]. These use-cases highlight the need for explicit IR-level contracts and modularity: the same high-level algorithm must be portable across device generations, yet adapted to exploit specialized hardware resources such as DSP slices, systolic AI engines, or high-bandwidth memories.

The contribution of this work is therefore threefold:

1) **We formalize the role of an explicit IR in HLS, decoupling algorithms from implementation decisions and introducing contract-driven determinism.**
2) **We present a reusable VHDL microinstruction library and emulator that stabilize implementation and provide auditable artifacts.**
3) **We show how PyHLS extends naturally to heterogeneous FPGAs, mapping operators to emerging AI/ML blocks while maintaining scientific reproducibility and portability across decades.**

By unifying algorithmic specification, IR-based parameterization, and reusable microinstructions, PyHLS establishes a sustainable methodology for real-time physics experiments and beyond. In short: *write the science once, retarget the hardware many times*.H

***Keywords*—High-Level Synthesis; intermediate representation; FPGA; Real-Time Systems; triggers, microinstructions; modularity; AI tiles**

I. Introduction

THE last two decades have seen a rapid growth in the use of reconfigurable hardware in physics experiments, embedded systems, and data acquisition pipelines. Field-Programmable Gate Arrays (FPGAs) now form the backbone of many real-time systems where latency and determinism are as important as throughput and energy efficiency. Unlike CPUs and GPUs, which rely on instruction-level execution and often variable latency, FPGAs offer fully customized datapaths and memory fabrics that can be tailored to a specific application domain. This property makes them indispensable in areas such as plasma diagnostics for fusion devices, high-energy physics triggers, network packet inspection, and advanced control systems.

However, FPGA development has traditionally relied on hardware description languages (HDLs) such as VHDL or Verilog. While these approaches guarantee full control over timing and resources, they are labor-intensive and error-prone, particularly in large, long-lived scientific projects. As the complexity of physics experiments grows, so does the demand for higher productivity design flows that allow scientists and engineers to focus on the algorithms rather than the intricacies of low-level implementation.

High-Level Synthesis and Its Limitations

High-Level Synthesis (HLS) tools emerged to address this productivity gap by allowing designers to specify algorithms in C, C++, SystemC, or Python, which are then automatically compiled into RTL. This abstraction promised rapid development and reuse of high-level code. Yet, in practice, HLS has revealed significant limitations for real-time applications. The most critical shortcoming is the heavy reliance on pragmas—compiler directives embedded in the high-level language—which dictate pipelining, loop unrolling, and memory partitioning strategies. While effective in the short term, pragmas couple algorithmic code tightly to specific compiler versions and FPGA families, undermining both portability and reproducibility.

In addition, timing predictability remains a challenge. In classical HLS flows, performance emerges as a side-effect of tool heuristics rather than as an explicit contract. This unpredictability is problematic in scientific domains where stable latency and bounded jitter are not just desirable but mandatory. As highlighted in earlier reviews of FPGA parallelism [7], the interaction between concurrency, memory bandwidth, and scheduling is highly complex. Without a structured way to represent and reason about these factors, HLS often produces designs that are difficult to validate or port.

Scientific Motivation: ITER and JET

A concrete illustration of these issues comes from plasma diagnostics in fusion experiments. The JET tokamak, and its successor ITER, require x-ray spectrometry systems and other diagnostic pipelines to capture and analyze plasma behavior in real time [6]. These systems must operate with deterministic latency, often below microseconds, while handling very high data rates. At the same time, they must remain maintainable across decades of experimental operation, during which FPGA families and vendor tools will inevitably evolve. A design flow that entangles algorithms with low-level pragmas cannot meet these long-term requirements. What is needed instead is a methodology that keeps the algorithmic layer stable while allowing implementation choices to evolve.

Beyond Fusion: Triggers and Rare-Event Detection

Similar constraints arise in high-energy physics (HEP) and astrophysics. First-level triggers in experiments such as the Pierre Auger Observatory or CERN-based detectors must filter rare physical events—such as neutrino-induced showers or exotic particle interactions—from extremely large backgrounds. Latency budgets are tight, and decisions must be taken deterministically within fixed windows. The use of artificial neural networks (ANNs) [4] and fuzzy-logic discriminators [5] implemented directly in FPGA logic shows the field's drive toward more sophisticated, algorithmically rich triggers. However, these advances also highlight the inadequacy of classical HLS approaches: as algorithms grow in complexity, embedding performance directives directly in the HLL code becomes unsustainable.

The Case for an Intermediate Representation (IR)

The limitations of existing HLS methodologies point toward the need for a structured Intermediate Representation (IR). An IR provides a middle layer between the high-level algorithm and the low-level RTL, explicitly capturing timing, concurrency, and resource-binding decisions. In contrast to pragmas, which are scattered directives, an IR forms a coherent, analyzable model of the design. This model can be versioned, audited, and systematically explored. Crucially, it provides a stable interface: the algorithm remains unchanged in the HLL, while IR attributes evolve as hardware generations change.

Our proposal, PyHLS, embodies this philosophy. Written in Python both at the algorithmic and compiler levels, PyHLS decouples scientific algorithms from hardware-specific optimizations. The IR serves as the single source of truth for performance decisions, while a reusable microinstruction library ensures that implementations are modular, portable, and auditable. This separation is particularly well suited for scientific projects where reproducibility and long-term maintainability are as important as raw performance.

Contribution of This Paper

This paper introduces an IR-centric HLS methodology tailored for real-time scientific applications. Our contributions are:

- We motivate the need for explicit IR contracts in fusion diagnostics and HEP triggers, where determinism and portability are critical [4]–[6].
- We build on prior studies of FPGA parallelism and HLS parameterization [7], [8], extending them into a structured metamodel framework.
- We introduce a reusable VHDL microinstruction library [3], combined with a cycle-accurate emulator, to ensure correctness and reproducibility.
- We show how capability-aware IR operators map algorithms to heterogeneous FPGA resources, including emerging AI/ML tiles.

By placing an explicit IR between algorithms and RTL, PyHLS creates a flow where algorithms are written once and retargeted many times, bridging the gap between productivity and determinism. This introduction sets the stage for a deeper exploration of motivation, metamodel design, microinstructions, heterogeneous FPGA mapping, and ultimately the conclusions of this work.

II. Motivation

Despite their success in raising the abstraction level of FPGA design, existing High-Level Synthesis (HLS) methodologies reveal fundamental shortcomings when applied to real-time scientific systems. The most pressing challenge arises from the reliance on *pragma-driven optimization*. Pragmas, inserted directly into the high-level language (HLL) code, dictate loop unrolling, pipelining, and memory partitioning. Although pragmatic in appearance, this approach tightly couples algorithmic descriptions to specific toolchains and device families. Over time, such coupling creates designs that are brittle, difficult to port, and hard to reproduce in long-lived projects.

Problems with pragma-driven HLS

Pragmas blur the separation between *algorithmic intent* and *implementation detail*. The scientific algorithm, which should remain a clean mathematical description, becomes polluted with annotations tied to a particular vendor tool. Moreover, pragmas are inherently local: they optimize individual loops or arrays but fail to provide a coherent, system-level view of timing, resource budgets, or latency contracts. As a result, the overall performance of the design emerges as a side effect of tool heuristics rather than as an explicit, analyzable property.

This unpredictability is unacceptable in experiments where every microsecond matters. For example, in fusion diagnostics or rare-event triggers, deadlines must be met deterministically. Silent degradation—where the compiler silently alters initiation intervals (II) or fails to meet a latency bound—is not tolerable. Without a higher-level representation of performance contracts, reproducibility and scientific auditability are undermined.

Parameterization as a solution

To overcome these limitations, we argue for systematic parameterization of the synthesis process. Instead of embedding performance hints inside the HLL, parameters should be first-class entities in an explicit *Intermediate Representation (IR)*. This allows timing (II, deadlines, jitter), concurrency (pipelining, unrolling), memory layout (banking, tiling), and resource ceilings (DSP, BRAM, LUT) to be expressed declaratively. Parameters can then be varied systematically in *design space exploration* (DSE), making it possible to navigate the trade-offs between performance, area, and power in a controlled fashion.

Earlier work on widely parameterizable HLS demonstrated the value of exposing a multidimensional parameter space for synthesis [8]. By modeling parameters as ranges, feasibility checks and cost models can prune the search space before full RTL synthesis. This ensures that only realistic and resource-compliant designs are considered, reducing wasted compilation cycles and accelerating design convergence.

The multi-level compiler concept

Parameterization is most effective when coupled with a multi-level compiler architecture. In this model, the HLL encodes only the algorithm, while intermediate layers—most importantly the IR—capture performance decisions. Back-ends then translate the IR into device-specific RTL. This philosophy was formalized in the multi-level compiler concept for HLS [9], which emphasized that abstraction should not stop at the algorithm level but extend through multiple layers of representation. Each layer serves as a boundary: algorithms remain pure, IR captures time and resources, and microinstructions materialize the design in reusable VHDL modules.

The multi-level approach provides several benefits. First, it isolates scientific code from toolchain volatility: as devices evolve, only IR presets and back-ends need to change. Second, it enables explicit reasoning about performance: contracts and resource budgets are part of the IR rather than scattered across pragmas. Third, it fosters modularity: subsystems can be

compiled independently and integrated via IR-level contracts, improving maintainability.

Practical implications

The motivation for our work is thus twofold. On one hand, we must escape the limitations of pragma-driven flows that compromise determinism, portability, and reproducibility. On the other, we must embrace parameterization and multi-level abstraction as the foundation of a sustainable methodology. By placing an explicit IR at the heart of the synthesis process, PyHLS transforms performance from an incidental property into a contract, while enabling systematic exploration of the parameter space. This approach provides the predictability required by real-time physics experiments and the flexibility demanded by decades-long scientific collaborations.

III. IR-Centric HLS

The central thesis of this work is that a dedicated *Intermediate Representation (IR)* layer transforms High-Level Synthesis (HLS) from a pragma-driven heuristic into a structured, contract-based methodology. Rather than embedding optimization hints in the high-level language (HLL), PyHLS moves all performance-critical decisions into the IR. This separation allows scientific algorithms to remain pure and portable while ensuring that timing, concurrency, and memory policies are explicit, analyzable, and auditable.

Rationale and Background

The need for an intermediate layer arises from the shortcomings of classical HLS. As shown in prior studies on algorithmic synthesis using Python compilers [11] and RPython-based flows [10], embedding optimization at the source code level leads to fragile designs that depend on specific compiler interpretations. By contrast, an IR-centric approach creates a stable abstraction boundary:

- **HLL (Python)** encodes only the algorithmic intent and test vectors.
- **IR** encodes performance contracts: deadlines, initiation intervals (II), loop unroll factors, memory banks, and resource ceilings.
- **Microinstructions (VHDL)** materialize the IR as reusable units, links, and memories [3].
- **Back-ends** translate the same IR to different FPGA families, exploiting vendor-specific optimizations without altering the algorithm.

This separation reflects a broader trend in compiler design: the introduction of multiple intermediate layers, each responsible for progressively lowering abstraction while preserving correctness. In HLS, however, such layering has historically been weak, with pragmas serving as a poor substitute for an explicit IR. PyHLS addresses this gap by introducing IR as a first-class citizen.

Modularity at Different Levels

The IR-centric flow promotes modularity across four levels:

1) **Algorithmic Level.** The high-level algorithm is written in Python. It remains unchanged when retargeting to new devices or adopting new optimization strategies. Functional correctness and scientific meaning are preserved.
2) **IR Level.** The IR captures timing, concurrency, and memory attributes. Each block (dataflow, control, memory) is represented as an independent module with explicit contracts. This makes the design space explicit and analyzable.
3) **Microinstruction Library.** A set of reusable VHDL modules defines datapath primitives, memory fabrics, and controllers [3]. Algorithmic IR operators are mapped to these microinstructions, ensuring reusability and uniformity across projects.
4) **Back-End Level.** Vendor-specific back-ends lower the same IR into RTL for different FPGA families (Intel, Xilinx, Lattice) or even ASIC targets. Device heterogeneity is handled at this level without contaminating the algorithm.

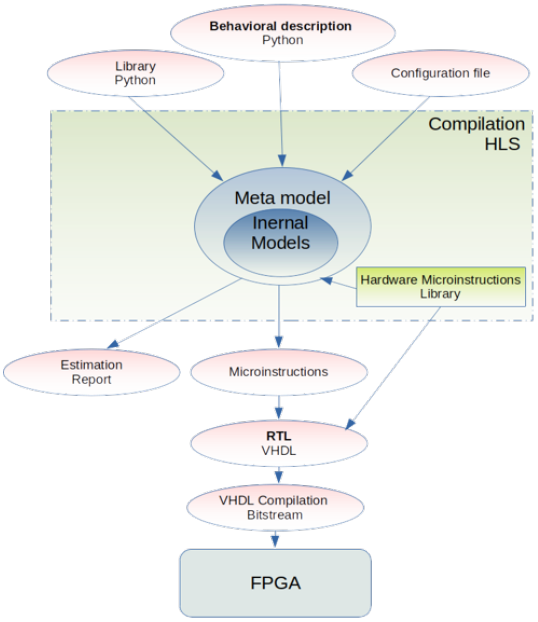

Fig. 1. Compilation and optimization model in PyHLS. The IR separates algorithmic description (Python) from reusable microinstructions (VHDL) and device-specific back-ends. Modularity is preserved at algorithm, IR, microinstruction, and back-end levels.

IR as a Parameterized Contract Space

In PyHLS, the IR is not just an intermediate syntax tree. It is a parameterized graph where each operation and channel carries attributes and contracts. These parameters cover:

- **Representation.** Numeric types (fixed/floating), bit-widths, quantization and rounding, vector layouts.
- **Transformation.** Loop unrolling, pipelining, tiling, strip-mining, memory banking.

- **Estimation.** Operator latency models (DSP vs LUT), area usage, simple power models.
- **Constraints.** Deadlines, jitter bounds, throughput/II targets, ceilings on DSP/BRAM/LUT/FF.
- **Heuristics.** Scheduler/binder biases, exploration strategies.

This taxonomy is consistent with the vision of widely parameterizable HLS flows [8], but here embedded explicitly in IR. Parameters may be fixed values or ranges to be swept during design space exploration (DSE). Conflicts, such as more memory reads than ports available at a given II, are reported deterministically with minimal counterexamples, ensuring that no design silently fails.

Lowering Path: From Algorithm to Hardware

The lowering process from HLL to IR and then to microinstructions proceeds in structured stages:

1) **Algorithm Analysis.** The Python AST is parsed; loops, dependences, and memory accesses are summarized.
2) **IR Construction.** Operations and channels are represented with timing and resource attributes; contracts are attached.
3) **Legality Checks.** Loop-carried dependences and memory conflicts are validated against requested II and unroll factors.
4) **Scheduling.** A timestamped dataflow graph is generated, ensuring that deadlines and ceilings are respected.
5) **Binding.** Operations are mapped to DSPs, LUTs, or AI tiles; arrays are mapped to banks or replicated RAMs with explicit port usage.
6) **Layout.** Multiplexers and FIFOs are inserted to respect cycle budgets and meet timing closure.
7) **Control Generation.** Guard logic, counters, and handshake signals are synthesized, keeping datapaths regular.
8) **Microinstruction Emission.** The design is encoded as a stream of microinstructions that drive the reusable VHDL library [3].

This disciplined lowering path stabilizes the synthesis process. Instead of ad hoc pragma tweaking, each decision is explicit, inspectable, and auditable.

Microinstructions as the Executable Form of IR

A distinctive feature of PyHLS is that IR is not only analyzed but also executed. By lowering IR into a microinstruction stream, PyHLS creates an executable artifact that can be run in a cycle-accurate emulator. Each instruction carries timing metadata, and the emulator checks contracts such as deadlines, II, and jitter. This approach pushes failures early—designs that cannot meet contracts fail before vendor synthesis. The dual artifact (HDL + microinstructions) provides clarity about what was built and why it satisfies performance targets.

Comparison with Prior HLS Approaches

Earlier Python-based HLS efforts demonstrated the potential of using high-level languages for hardware synthesis [10], [11], but lacked a stable intermediate abstraction. Vendor tools such as Xilinx Vivado HLS or Intel HLS Compiler offer pragma-driven optimization but leave determinism implicit. Open-source flows such as LegUp and Bambu emphasize productivity but often lack explicit timing contracts. PyHLS differs in its insistence that all timing and resource properties be first-class IR attributes, enforced by an emulator and realized through reusable microinstructions. This positions IR as the single source of truth.

Implications for Scientific and Engineering Projects

By introducing a dedicated IR layer, PyHLS provides the determinism, portability, and auditability required for long-lived scientific facilities. In ITER and JET, algorithms for spectroscopy or diagnostics can be written once and retargeted across FPGA generations by evolving IR presets. In high-energy astrophysics, trigger algorithms based on neural networks or fuzzy logic [4], [5] can be mapped to AI tiles or DSPs by altering IR binding policies, without rewriting the algorithm. The IR-centric methodology thus bridges the gap between scientific intent and hardware realization, ensuring that FPGA accelerators remain both performant and sustainable.

IV. Microinstructions and Emulation

A defining feature of PyHLS is the use of a reusable *microinstruction library* as the executable carrier of the Intermediate Representation (IR). Rather than emitting plain RTL code directly from the IR, the compiler lowers algorithmic operators, memory actions, and control flow into streams of parameterized microinstructions. These microinstructions instantiate reusable VHDL primitives—datapath units, controllers, and memory fabrics—providing a stable middle layer between the IR and vendor back-ends [3].

Microinstruction Classes

The microinstruction set is compact but expressive. Each instruction encodes an operation, its arguments, attributes, and optional timing metadata:

```
INST { opcode, args[], attrs{}, timing{} }
```

The instruction classes include:

- **Declaration/configuration:** introduce units, ports, buffers, and clock domains.
- **Data movement:** memory loads/stores, buffer-to-unit transfers, scatter/gather patterns.
- **Arithmetic/logic:** DSP and ALU operations (e.g., add, multiply, MAC), with latency and pipeline attributes.
- **Control:** conditional branches, loop counters, and synchronization barriers.
- **Synchronization:** valid/ready signals, deadline markers, epoch tags.

By representing datapath and control as streams of such instructions, PyHLS ensures modularity. A convolution kernel, a matrix multiply, or a trigger filter can be expressed as a parameterized sequence of microinstructions, independent of vendor or device family.

Advantages of a Microinstruction Layer

The use of microinstructions provides several practical benefits:

- **Reusability.** Instructions map to a library of verified VHDL modules that can be reused across projects and devices.
- **Auditability.** Instruction streams are human-readable and diffable; changes in pipeline depth or unroll factor are visible as small edits.
- **Portability.** Device-specific back-ends interpret the same instruction stream differently, binding operators to DSPs, LUTs, or AI tiles.
- **Determinism.** Timing metadata embedded in instructions allows explicit contracts (II, deadlines, jitter) to be enforced by the controller.

This approach mirrors earlier ideas of modular DSP building blocks for FPGA-based accelerators, but extends them into a unified HLS flow [12]. Instead of ad hoc instantiation, every operator in PyHLS flows through a common instruction abstraction.

Cycle-Accurate Emulation

To validate designs before vendor synthesis, PyHLS provides a cycle-accurate emulator that executes microinstruction streams. The emulator models:

- **Primitive units:** registers, FIFOs, RAMs (with ports and latencies), DSP/ALU units.
- **Control:** counters, guards, handshakes.
- **Channels:** typed links with valid/ready semantics and backpressure.

At each cycle, the emulator:

1) Fetches instructions ready to execute.
2) Checks port availability; if a port is oversubscribed, it raises a contention violation.
3) Advances units according to their latency models.
4) Commits results to registers or memories in the correct order.
5) Updates control state and validates deadlines, II, and jitter.

Outputs include waveforms of selected signals, FIFO occupancies, resource utilization, and timing summaries. Importantly, the emulator provides *early failure*: infeasible designs are rejected before synthesis, saving time and avoiding misleading results.

Application to AI and Real-Time Systems

The microinstruction layer is especially powerful when targeting heterogeneous FPGAs with AI accelerators. For example, a matrix multiply operator may be expressed as a stream of multiply-accumulate instructions. On one device, these instructions bind to DSP blocks; on another, to dedicated AI tiles with systolic scheduling. The IR remains unchanged, and the algorithmic code in Python is untouched. This capability aligns with earlier demonstrations of reconfigurable computing accelerating AI workloads [12], but extends them with a systematic, contract-driven methodology.

In real-time physics experiments, such as fusion diagnostics or cosmic-ray triggers, emulation provides confidence that latency and determinism are met before deployment. ANN-based triggers for inclined showers [4] or fuzzy-logic triggers for neutrino signatures [5] can thus be validated in a cycle-accurate environment before being committed to hardware.

Summary

The combination of a microinstruction library and a cycle-accurate emulator bridges the gap between abstract IR and concrete FPGA implementations. Microinstructions ensure modularity, portability, and reproducibility; emulation ensures correctness and determinism. Together, they form the practical backbone of PyHLS, enabling scientific algorithms to be "written once" and "retargeted many times" without sacrificing the stringent requirements of real-time systems.

V. Future Work

While the present work establishes the foundations of an IR-centric HLS flow, the most compelling opportunities for extension arise from *trigger systems* in large-scale physics experiments. These systems, which must discriminate rare signals from overwhelming backgrounds, place the strictest demands on determinism, latency, and reproducibility.

Neural and Fuzzy Logic Triggers

Past efforts have shown that artificial neural networks (ANNs) can successfully identify signatures of very inclined air showers when implemented in FPGA hardware [4]. Similarly, fuzzy-logic triggers have been proposed for distinguishing neutrino-induced events in the Pierre Auger Observatory [5]. Both approaches highlight the need for low-latency, high-throughput accelerators that are also flexible enough to adapt to evolving physics requirements. Extending PyHLS with specialized IR operators for ANN layers (dense, convolutional, pooling) and fuzzy classifiers would directly address this need.

CNNs and Pattern-Based Methods

Convolutional Neural Networks (CNNs) represent a natural extension of ANN-based triggers. Their ability to detect spatial and temporal patterns makes them suitable for analyzing waveforms, images, or segmented detector data. Implementing CNN layers efficiently on FPGA requires careful handling of convolutions, data reuse, and memory banking. By extending the IR with CNN-specific capability tags (e.g., `conv1d`, `conv2d`, `pooling`) and binding policies (DSP vs AI tile), PyHLS can target future heterogeneous FPGAs that integrate AI blocks. Beyond neural models, pattern-based algorithms—such as matched filtering or template correlation—could also be formalized as IR primitives. These algorithms are widely used in physics for feature extraction and can benefit from the same systematic parameterization.

Extending the Compiler and IR

Supporting these trigger-oriented algorithms will require several compiler and IR enhancements:

- **Specialized operators.** IR extensions for ANN and CNN layers, pattern-matching kernels, and fuzzy inference rules.
- **Capability-aware binding.** Mapping dense linear algebra to AI tiles or DSP blocks, depending on device capabilities.
- **Contract integration.** Expressing latency, throughput, and memory bandwidth requirements of neural triggers as explicit IR contracts.
- **Reusable microinstructions.** Enriching the microinstruction library with optimized primitives for convolutions, activation functions, and pattern recognition.

Scientific Relevance

Trigger systems are often the first and most critical stage of data acquisition. In fusion diagnostics, they decide which plasma events warrant deeper analysis; in astroparticle physics, they determine whether candidate neutrino or cosmic-ray events are preserved for offline study. By extending PyHLS toward ANN, CNN, and pattern-based triggers, we enable experiments to leverage heterogeneous FPGA platforms with confidence that their timing constraints will be met. This direction aligns naturally with the long-term demands of facilities such as ITER, JET, and the Pierre Auger Observatory.

Outlook

The ultimate goal of this line of research is to create a unified, IR-centric toolchain where physics triggers—whether based on ANN, CNN, fuzzy logic, or pattern matching—are expressed once in Python and then retargeted across FPGA generations. With IR carrying the performance contracts and the microinstruction library enforcing modularity, such triggers can remain both scientifically valid and technically portable for decades.

VI. Conclusions

This paper has presented a comprehensive argument for an *Intermediate Representation (IR)-centric* approach to High-Level Synthesis (HLS), with specific focus on the demands of real-time physics experiments such as fusion diagnostics and astroparticle triggers. By introducing a dedicated IR layer between high-level algorithmic description and hardware-specific realization, PyHLS establishes a contract-driven methodology that ensures determinism, portability, and auditability—properties that are indispensable for long-lived scientific facilities.

Transfer of Abstraction

The most important conceptual shift advocated in this work is the deliberate transfer of performance-related concerns out of the high-level language (HLL) and into the IR. While the HLL (Python) remains the reference point for algorithmic intent and functional validation, all decisions regarding timing, concurrency, resource binding, and memory layout are expressed in the IR. This separation eliminates the fragility of pragma-driven flows, where small changes in compiler heuristics or vendor tool versions can lead to unpredictable variations in performance. Instead, the IR provides a stable, inspectable, and versioned record of performance contracts that must be met for an implementation to be considered correct.

Determinism by Explicit Contracts

A central advantage of the IR is that it elevates performance targets to first-class contracts. Initiation intervals (II), end-to-end deadlines, jitter bounds, and resource ceilings are not implicit side effects of scheduling but explicit properties attached to IR operators and channels. The compiler must either satisfy these contracts or report minimal counterexamples that pinpoint infeasible assumptions. This contract-driven model transforms latency from a heuristic goal into a predictable guarantee. For experiments that must respond to signals within microseconds—such as plasma diagnostics at ITER [6] or rare-event triggers in the Pierre Auger Observatory [4], [5]—this determinism is not optional but mission-critical.

Portability Across Device Generations

Another major benefit of the IR-centric approach is portability. FPGA devices evolve rapidly, with each new generation offering different mixes of DSP slices, logic, memory resources, and increasingly specialized AI tiles. Vendor toolchains also change, sometimes in backward-incompatible ways. Scientific projects, however, often span decades, during which algorithms must remain valid even as the underlying hardware shifts. By maintaining the algorithm in pure Python and expressing performance in the IR, PyHLS allows designs to be retargeted across devices by adjusting IR presets and back-end mappings. This ensures that the same scientific code can be replayed on legacy platforms or extended to exploit next-generation heterogeneous architectures without rewriting the algorithm.

Auditability and Reproducibility

Scientific work demands transparency and reproducibility. In PyHLS, IR graphs, parameters, and contracts are human-readable and version-controlled. A change in unroll factor or memory banking strategy is visible as a small diff in the IR, making design decisions traceable and reviewable. Moreover, by lowering IR into both HDL and microinstruction streams [3], PyHLS generates dual artifacts: the implementation itself and a structured record of the decisions that produced it. This duality enhances reproducibility: accelerator designs can be reconstructed and validated years later by replaying the same microinstruction streams under the same contracts.

Microinstructions and Emulation as Practical Tools

The microinstruction library acts as the executable form of the IR. Rather than producing opaque RTL directly, PyHLS

lowers designs into streams of parameterized microinstructions that instantiate reusable VHDL modules. This provides three advantages: (i) modularity—operators and memories are described by composable building blocks; (ii) portability—back-ends interpret the same stream differently depending on device capabilities; and (iii) auditability—streams are human-readable, versioned, and diffable.

Complementing this is a cycle-accurate emulator that executes microinstruction streams before vendor synthesis. The emulator validates functionality, checks latency and initiation interval contracts, and detects hazards such as port contention. By moving failures earlier in the flow, the emulator shortens iteration time and stabilizes continuous integration pipelines. In domains where missed deadlines cannot be tolerated, such as real-time DAQ in plasma diagnostics [6] or online pattern recognition in cosmic-ray experiments [4], [5], this ability to prune infeasible designs early is invaluable.

Alignment with Parameterizable and Multi-Level HLS

The PyHLS philosophy resonates with prior efforts in widely parameterizable HLS [8] and multi-level compiler design [9]. By embedding a structured parameter space in the IR—including representation, transformation, estimation, constraints, and heuristics—PyHLS allows systematic design space exploration. Instead of rewriting HLL code, designers sweep IR parameters such as unroll factors, pipeline depths, and memory banking schemes. This makes design optimization a reproducible, configuration-driven process rather than an ad hoc art. Similarly, the multi-level philosophy [10], [11] is realized here in practice: from algorithm in Python, to IR with explicit contracts, to microinstructions, to back-end-specific RTL.

Heterogeneous FPGA Devices and Specialized Blocks

The rise of heterogeneous FPGAs reinforces the need for an explicit IR layer. Modern devices integrate not only general-purpose logic and DSP slices but also specialized AI engines (matrix multipliers, systolic arrays), high-bandwidth memory (HBM), and on-chip networks. Exploiting these resources requires capability-aware binding. In PyHLS, IR operators carry capability tags (e.g., `matmul_int8`, `conv1d_fp16`) and binding policies (`prefer: AI_tile; fallback: DSP; legal: LUT`). Back-ends interpret these tags to select the most efficient implementation for the target device. This mechanism allows the same IR to map to DSP-based designs on legacy devices and AI-tile-based designs on newer ones [12]. The algorithmic code remains unchanged, and the IR contracts remain intact.

Triggers as a Driving Use-Case

Trigger systems in physics experiments exemplify the need for IR-centric HLS. They must recognize rare and subtle patterns—whether plasma instabilities, neutrino-induced showers, or cosmic-ray events—while rejecting overwhelming background. ANN-based triggers for very inclined showers [4] and fuzzy-logic triggers for neutrino discrimination [5] have demonstrated the feasibility of advanced algorithms on FPGA hardware. Looking forward, CNNs and pattern-based methods represent natural extensions: convolutions can capture spatial or temporal correlations, while template-matching kernels provide deterministic pattern recognition. By extending the IR with operators for dense layers, convolutions, pooling, and pattern filters, PyHLS can directly support such triggers. These operators would carry explicit contracts for latency and throughput, ensuring that physics requirements are met.

Scientific and Practical Implications

The implications of the IR-centric methodology extend beyond academic elegance. For large facilities such as ITER, JET, or the Pierre Auger Observatory, the cost of revalidating algorithms after toolchain changes is immense. By freezing the HLL and managing performance exclusively via IR and presets, PyHLS reduces this cost. For engineers, the availability of a microinstruction library and emulator provides practical tools for debugging and optimization. For scientists, the auditability of IR ensures that algorithms remain reproducible and trustworthy even decades after deployment.

Outlook

Future work will extend PyHLS along several lines:

- **Trigger-oriented operators.** Adding IR primitives for ANN layers, CNN convolutions, fuzzy inference, and pattern matching, building directly on prior trigger research [4], [5].
- **Capability-aware compilation.** Refining back-ends to exploit AI tiles, HBM, and NoC fabrics on heterogeneous FPGAs while preserving IR contracts.
- **Extended emulation.** Enhancing the microinstruction emulator to cover more device models and to integrate energy estimation for sustainable design.

Final Takeaway

The trajectory of FPGA devices is clear: greater heterogeneity, stronger AI acceleration, and tighter integration with large-scale experiments. The trajectory of scientific facilities is also clear: longer lifespans, stricter reproducibility demands, and more complex real-time triggers. The IR-centric approach of PyHLS addresses both trajectories. By keeping algorithms pure in Python, moving performance into explicit IR contracts, and enforcing them through reusable microinstructions and emulation, PyHLS creates a future-proof methodology. It is a methodology in which algorithms are written once, retargeted many times, and trusted always—a foundation for the next generation of real-time scientific computing.

REFERENCES

[1] R. Cieszewski and K. Poźniak, "Synteza Wysokiego Poziomu z wykorzystaniem jezyka Python," *Elektronika - konstrukcje, technologie, zastosowania*, vol. 58, no. 8, pp. 31–35, 2017. https://doi.org/10.15199/13.2017.8.7

[2] R. Cieszewski, K. Poźniak, and R. Romaniuk, "Python Cased High-Level Synthesis Compiler," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 9290, pp. 92903A-1–92903A-8, 2014. https://doi.org/10.1117/12.2075988

[3] K. Poźniak, "VHDL-based universal programmable process for FPGA," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments*, vol. 12476, 124760X, 2022. https://doi.org/10.1117/12.2659617

[4] Z. Szadkowski and K. Pytel, "Artificial Neural Network as a FPGA Trigger for a Detection of Very Inclined Air Showers," in *Proc. IEEE Real Time Conference*, Nara, Japan, May 25–30, 2014, pp. 1–8. Available: https://doi.org/10.48550/arXiv.1406.1903

[5] K. Pytel and Z. Szadkowski, "Proposal of the Fuzzy Trigger for the Surface Detector of the Pierre Auger Observatory," *IEEE Transactions on Nuclear Science*, vol. 71, no. 6, pp. 1281–1291, 2024. https://doi.org/10.1109/TNS.2024.3386217

[6] A. E. Shumack, A. Byszuk, R. Cieszewski, et al., "X-ray Crystal Spectrometer Upgrade for ITER-like Wall Experiments at JET," *Review of Scientific Instruments*, vol. 85, no. 11, pp. 11E425, 2014. https://doi.org/10.1063/1.4891182

[7] R. Cieszewski, M. G. Linczuk, K. Poźniak, and R. Romaniuk, "Review of Parallel Computing Methods and Tools for FPGA Technology," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 8903, pp. 890321–13, 2013. https://doi.org/10.1117/12.2035385

[8] R. Cieszewski, K. Poźniak, and M. G. Linczuk, "Widely parameterizable high-level synthesis," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 10808, 2018. https://doi.org/10.1117/12.2502153

[9] R. Cieszewski, R. Romaniuk, and K. Poźniak, "Multi-level compiler concept for high-level synthesis," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 12476, pp. 1–7, 2022. https://doi.org/10.1117/12.2659459

[10] R. Cieszewski and M. G. Linczuk, "RPython high-level synthesis," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 10031, pp. 100314O-1–6, 2016. https://doi.org/10.1117/12.2249143

[11] R. Cieszewski, R. Romaniuk, K. Poźniak, and M. G. Linczuk, "Algorithmic synthesis using Python compiler," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 9662, pp. 96623J-1–8, 2015. https://doi.org/10.1117/12.2205609

[12] R. Cieszewski, "Accelerating Artificial Intelligence with Reconfigurable Computing," in *Proc. SPIE Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, vol. 8454, pp. 84541L-1–8, 2012. https://doi.org/10.1117/12.2000098