# EquivFusion: Unifying Hardware Equivalence Checking from Algorithms to Netlists via MLIR

Jiaying Zhu
The Chinese University of Hong Kong
National Center of Technology
Innovation for EDA

Baoqi Zhang
Mengxia Tao
National Center of Technology
Innovation for EDA

Kezhi Li
The Chinese University of Hong Kong

Hao Yan
Southeast University
National Center of Technology
Innovation for EDA

Qiang Xu
The Chinese University of Hong Kong

Min Li*
Southeast University
National Center of Technology
Innovation for EDA

## Abstract

Ensuring functional consistency between high-level algorithmic models and low-level hardware implementations is a critical challenge, particularly as modern design flows increasingly span heterogeneous abstractions—from deep learning frameworks to hardware netlists. In this paper, we present EquivFusion, an end-to-end equivalence checking tool tailored for multi-modal circuit designs. Unlike traditional flows that rely on siloed tools or ad-hoc translation, EquivFusion leverages a verification-oriented MLIR lowering pipeline to unify diverse entry points—including PyTorch, C/C++, Chisel, Verilog, and gate-level netlists—into a common intermediate representation. This architecture enables automated, pairwise equivalence checking across diverse abstraction levels by rigorously translating designs into standard formal verification formats, i.e., SMT-LIB, BTOR2, AIGER. We demonstrate EquivFusion's feasibility to bridge the semantic gap between software specifications and hardware realizations, showcasing its effectiveness in facilitating "shift-left" formal verification for datapath-intensive hardware designs. EquivFusion is available online[1].

## 1 Introduction

The complexity of modern hardware systems has necessitated a paradigm shift in design flows. To cope with the demands of domain-specific accelerators and data-intensive workloads, designers are moving beyond traditional Register-Transfer Level (RTL) entry points. The contemporary design spectrum is highly heterogeneous: algorithmic specifications are often defined in deep learning frameworks like PyTorch [20]; high-level synthesis (HLS) models [12] are written in C++; and hardware construction languages (HCLs) such as Chisel [4] are used to generate RTL, which is subsequently synthesized into gate-level netlists for implementation. This diversity enables rapid innovation but introduces a fundamental challenge: ensuring functional correctness across representations that differ drastically in semantics, granularity, and tooling support.

For example, verifying that a PyTorch model faithfully corresponds to its Verilog counterpart is critical in domains like mobile System-on-Chips (SoCs), where camera pipelines increasingly replace traditional Image Signal Processors (ISP) with end-to-end deep neural hardware [28]. Similarly, datapath-intensive accelerators for vision or language models often originate from high-level algorithmic intent but must be validated against low-level realizations [11, 12]. These scenarios highlight the need for robust equivalence checking across heterogeneous design abstractions [32].

Existing equivalence checking tools, however, remain largely siloed across specific abstraction boundaries and modalities. Open-source engines such as ABC [9] and Yosys EQY [33, 34] offer logic equivalence checking for RTL/netlist designs, while HW-CBMC [25] supports bounded checking between C and Verilog. Commercial solutions such as Synopsys VC Formal Datapath [29] and Cadence C2RTL [31] target C/C++-to-RTL datapath validation, and tools like Synopsys Formality [30] focus on RTL-to-gate equivalence for synthesis sign-off. Despite their effectiveness within each segment, none of these tools provides an end-to-end, extensible front-end that directly ingests modern algorithmic specifications (e.g., PyTorch) and hardware implementations (e.g., Verilog) and align them for equivalence checking. As a result, cross-modal validation often falls back to simulation-driven approaches [8, 19], requiring manual testbenches and are prone to missing subtle arithmetic discrepancies in datapath-intensive designs [7, 16].

The emergence of MLIR [2, 22] and CIRCT [1] offers a promising foundation for unification. Their multi-level design allows structural and semantic information to be captured explicitly across abstraction boundaries. However, existing MLIR workflows are primarily oriented toward compilation, lacking a verification-oriented pipeline capable of modeling equivalence relations or generating proof obligations. While recent work [15] has explored embedding logic equivalence checking (LEC) and a small subset of SystemVerilog Assertions (SVAs) [13] into MLIR, it does not address the broader challenge of cross-modal hardware equivalence checking. As verification shifts earlier in the design cycle, there is a critical need for a system that leverages MLIR's representational power while automating the generation of formal checks.

To this end, we introduce EquivFusion, a practical equivalence verification pipeline that unifies heterogeneous design modalities. We leverage MLIR not merely as a compilation infrastructure, but as the backbone of a rigorous verification workflow. EquivFusion advances the state of hardware verification through the following contributions:

- **Unified Multi-Modal Frontend:** EquivFusion acts as a semantic bridge, enabling automated pairwise equivalence checking

*Correspondence: min.li@seu.edu.cn
[1]Code: https://github.com/FORMiND-Lab/EquivFusion; Demo video: https://youtu.be/AdEeZHU54qA

between diverse abstraction levels. Users can formally verify a Chisel module against its C++ behavioral specification, or a synthesizable PyTorch model directly against its RTL implementation.

- **Verification-Oriented Lowering:** We implement a specialized MLIR pipeline that preserves high-level semantics (e.g., array operations, loop structures) where beneficial, while progressively lowering hardware specifics to a canonical representation. This avoids the semantic loss often associated with generic synthesis.
- **Flexible Multi-Engine Backend:** Rather than binding to a single engine, EquivFusion exports designs to standard formats including SMT-LIB [5], BTOR2 [27], and AIGER [6]. This flexibility allows users to leverage best-in-class open-source solvers (e.g., Z3 [14], Bitwuzla [26], kissat [17]) tailored to the specific logic depth of the problem.

Ultimately, EquivFusion bridges the semantic gap between specifications and implementations, enabling automated "shift-left" verification that traps functional errors early in the design cycle.

## 2 Related Work

### 2.1 MLIR-based Hardware Infrastructures

MLIR [2, 22] serves as a reusable compiler infrastructure designed to support domain-specific intermediate representations (IRs) through a unified dialect system. Its key innovation lies in enabling IRs at multiple abstraction levels to coexist and interact, facilitating progressive lowering and improving interoperability with downstream EDA tools. Built on MLIR, CIRCT [1] (Circuit IR Compilers and Tools) provides a collection of hardware-oriented dialects and tooling intended to form modular, reusable flows and to interoperate with downstream EDA tools. Recent work [15] implements basic logic checks and assertions into CIRCT, yet it is limited to small-scale reasoning and does not address the semantic gap between high-level algorithmic models and hardware implementations.

### 2.2 Hardware Equivalence Checking

Equivalence checking is a central technique in hardware verification for ensuring functional consistency across design transformations [21, 23]. Compared to standard flows where designs stabilize within a single domain, checking equivalence across abstraction levels-particularly between software reference models and hardware implementations-remains more challenging. It is commonly addressed through simulation-based verification or manually maintained reference models, approaches incurring significant engineering effort and scale poorly for datapath-intensive designs.

Existing tools focus on limited verification scenarios. Commercial solutions such as Synopsys Formality [30] and Cadence Conformal [10] target RTL-to-netlist equivalence, while Synopsys VC Formal DPV [29] and Cadence C2RTL [31] support C-to-RTL validation. Open-source tools including ABC [9], the Yosys EQY flow [33, 34], and HW-CBMC [25] provide scalable boolean or bounded reasoning, but are tightly coupled to fixed language pairs and RTL/netlist flows, lacking extensible front-end support for modern algorithmic specifications and hardware languages such as PyTorch or Chisel.

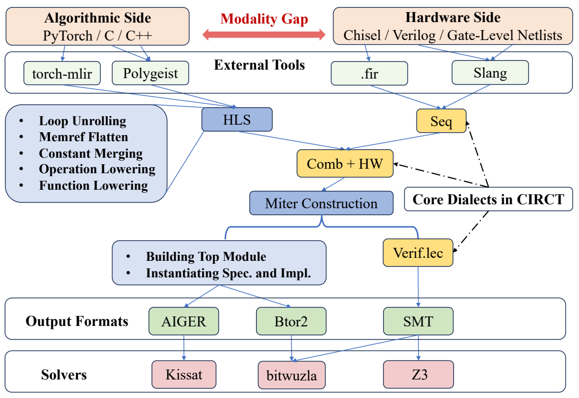

**Figure 1: Overview of EquivFusion.**

## 3 Overview of the Tool

### 3.1 Overview of Architecture

EquivFusion addresses cross-modal equivalence checking between algorithmic specifications and hardware implementations by unifying heterogeneous inputs into a common intermediate representation. To support "shift-left" verification, it adopts a verification-oriented lowering strategy that preserves reasoning-relevant semantics and delays bit-level refinement when possible. The tool is designed for datapath-intensive modules dominated by arithmetic and structured dataflow (e.g., kernels and accelerator datapaths), while control-dominant protocols with rich temporal behaviors are out of scope and are better served by hardware model checking [18].

As shown in Figure 1, EquivFusion accepts designs from diverse entry points, including PyTorch, C/C++, Chisel, Verilog, and gate-level netlists. Each input is first translated by existing front-ends into MLIR, bringing heterogeneous specifications into a common intermediate form. At the core, EquivFusion hosts a set of verification-oriented dialects on top of CIRCT: sequential designs are represented using a dedicated sequential abstraction and can be selectively unrolled to derive equivalent combinational views, while combinational and hardware-level structure is maintained through progressive lowering. On the unified IR, EquivFusion constructs a miter circuit by instantiating the specification and implementation under shared inputs and encoding output equivalence constraints. Finally, the miter is exported to standard solver formats (SMT-LIB, BTOR2, AIGER) and discharged to off-the-shelf SAT/SMT solvers. This modular design decouples language front-ends from verification back-ends, making it straightforward to extend EquivFusion with new input modalities or solvers.

### 3.2 Verification Scope

*3.2.1 Combinational Equivalence Checking.* EquivFusion supports combinational equivalence checking for designs that can be modeled as pure functions from inputs to outputs within a single cycle. This setting covers a wide range of modules extracted from larger systems, including arithmetic kernels, feed-forward blocks, and bit-precise datapath components, which may arise as standalone designs or as combinational regions from larger sequential systems.

*3.2.2 Transactional Equivalence Checking.* Beyond purely combinational settings, EquivFusion also targets equivalence checking for multi-cycle designs, where hardware behavior spans multiple clock cycles. In such scenarios, the two implementations may differ in cycle-level timing, latency, or internal pipelining, yet are expected to produce equivalent results over a complete transaction from inputs to outputs. Typical examples include validating an algorithmic reference model against a pipelined RTL implementation, or checking a high-level description with implicit scheduling against a multi-cycle hardware realization.

## 4 Implementation Details

### 4.1 User Interface and Workflow

The EquivFusion framework is accessed via its primary command-line interface (CLI), the `equiv_fusion` utility. A typical hardware equivalence verification workflow consists of five sequential steps, each invoked by a specific subcommand:

```
# Step 1: Specify input/output ports for algorithmic design
set_port <-input|-output> port-name
# Step 2: Read specification design
read_c|read_v|read_firrtl -spec -top <module-name> <input-file>
# Step 3: Read implementation design
read_c|read_v|read_firrtl -impl -top <module-name> <input-file>
# Step 4: Construct miter and generate output file
equiv_miter -specModule <module-name> -implModule <module-name>
-mitermode <smtlib|btor2|aiger> -o <miter-file>
# Step 5: Run solver for verification
solver_runner -solver <solver-name> -inputfile <miter-file>
```

**Listing 1: Execute Steps**

### 4.2 Unified Multi-Modal Frontend

To bridge the modality gap between heterogeneous abstraction levels, EquivFusion implements specialized translation flows built upon the CIRCT infrastructure. These flows map both algorithmic (Pytorch/C/C++) and hardware (Chisel/Verilog/netlists) descriptions onto a unified intermediate representation comprised of the CIRCT core dialects (HW, Comb, Seq, and Verif).

*4.2.1 Algorithmic Description Translation.* For high-level languages like C/C++ and PyTorch, EquivFusion targets a specific synthesizable subset conducive to High-Level Synthesis (HLS). EquivFusion leverages external compiler infrastructures to convert high-level specifications into the MLIR Affine dialect. Specifically, Polygeist [24] translates C/C++ programs, while torch-mlir [3] and upstream MLIR tooling are used to convert PyTorch models.
With the Affine dialect as the entry point, EquivFusion implements a custom HLS flow built upon the CIRCT infrastructure. This flow progressively lowers the Affine IR by applying a sequence of both existing CIRCT transformations and custom-implemented passes. The flow executes key operations including Loop Unrolling, Memref Flattening, Constant Merging, and Operation/Function Lowering, ultimately converting the high-level representation into an IR expressed in CIRCT's core dialects.

*4.2.2 Hardware Description Translation.* For Chisel designs, EquivFusion utilizes firtool (a utility within CIRCT) to parse the *.fir* file generated by the Chisel compiler and convert it into an IR expressed in CIRCT's core dialects. For Verilog/Netlist inputs, the tool employs *circt-verilog* (also within CIRCT), which leverages the Slang parser to process the source designs broad syntactic support. Both paths ultimately convert the hardware descriptions into the unified IR expressed in CIRCT's core dialects.

Notably, EquivFusion implements sequential unrolling to handle state-dependent logic. This technique expands the sequential circuitry over a specified number of clock cycles ($k$) to construct a cumulative combinational model. This model is behaviorally equivalent to the original design over the $k$ unrolled time-frames, thereby enabling transactional equivalence checking to rigorously verify state transitions across a finite temporal window.

### 4.3 Miter Construction Middle-End

The `equiv_miter` command orchestrates the miter circuit construction process, synthesizing a unified verification model from the distinct specification and implementation modules. The resulting miter circuit is subsequently exported into standard formats to facilitate formal equivalence checking.

*4.3.1 Input/Output Mapping.* The fundamental premise of the equivalence verification model in EquivFusion is that, given identical input stimuli, both the specification and implementation modules must produce identical outputs. Consequently, prior to constructing the miter model, EquivFusion establishes a one-to-one I/O correspondence between the *spec. module* and the *impl. module* via a port matching mechanism. EquivFusion establishes a one-to-one correspondence between specification and implementation ports by enforcing consistency in port count, names, and types. Upon successful validation, EquivFusion binds the ports by name to finalize the mapping for verification. Conversely, if any check fails, the tool reports a mismatch error and terminates the process.

*4.3.2 Construct Miter.* EquivFusion supports multiple modes for miter construction, including smtlib, btor2, and aiger. Depending on the selected mode, equivalence constraints are encoded either as word-level assertions or bit-level XOR checks, enabling compatibility across SAT and SMT solvers.

### 4.4 Solving Backends and Verification Results

The `solver_runner` command serves as the unified interface between EquivFusion's internal IR and external formal verification engines. It automates the invocation of backend solvers tailored to the specific verification format generated in the previous stage.

*4.4.1 Supported Solvers.* EquivFusion currently integrates three industry-standard solvers to cover different verification needs: Z3 [14] (for SMT-LIB), Bitwuzla [26] (optimized for BTOR2 word-level reasoning), and Kissat [17] (for AIGER bit-level verification). This flexibility allows users to select the most efficient engine based on the circuit's complexity and abstraction level.

*4.4.2 Verification Outcome and Debugging.* The solver's output provides a definitive conclusion regarding the functional consistency of the design pair: *UNSAT (Equivalent)* indicates that no input combination exists that causes the outputs to diverge, proving that the implementation is functionally equivalent to the specification; *SAT (Bugs Found)* indicates that the designs are not equivalent. Crucially, EquivFusion captures the *counterexample* provided by the solver — a specific assignment of input values that triggers the mismatch. This counterexample serves as a precise diagnostic trace, enabling designers to reproduce the error and pinpoint the logic bug within the hardware implementation.

## 5 Usage Scenarios and Case Studies

This section demonstrates the practical usage of EquivFusion through representative case studies.

## 5.1 Case Study 1: C++ *v.s.* Chisel

*5.1.1 Experimental Setup.* We verify a sorting module for unsigned integers to demonstrate equivalence checking across different abstraction levels. The setup contrasts a sequential software reference against a parallel hardware implementation:
**Specification (C++):** Bubble Sort algorithm (Listing 2)
**Implementation (Chisel):** Bitonic Sort algorithm (Listing 3)

```
#define N 8
extern "C" void Sort(unsigned char input[N], unsigned char
output[N]) {
    // Sorts the 'input' array, stores the result in 'output'
    ...
}
```

**Listing 2: Sort.cpp (Bubble Sort)**

```
class Sort(width: Int = 8) extends RawModule {
    val input = IO(Input(Vec(8, UInt(width.W))))
    val output = IO(Output(Vec(8, UInt(width.W))))
    // Sorts the 'io.input', drives the result to 'io.output'
    ...
}
```

**Listing 3: Sort.scala (Bitonic Sort)**

*5.1.2 Verification Process and Results.* We evaluate EquivFusion by conducting verification across two distinct scenarios.
**Experiment I: Consistent Sorting Order Verification.** In the baseline scenario, both the C++ specification and the Chisel implementation are configured to sort the input array in ascending order. The backend solver returns *UNSAT*, proving the Chisel implementation is functionally equivalent to the C++ specification, despite the fundamental algorithmic disparity (Bubble Sort *vs.* Bitonic Sort).
**Experiment II: Discrepancy Detection (Bug Injection).** Subsequently, to validate the tool's error detection capability, we modify the C++ specification to sort in descending order while leaving the Chisel implementation unchanged (ascending). The solver correctly returns *SAT*, successfully flagging the semantic mismatch and identifying the designs as non-equivalent.

## 5.2 Case Study 2: PyTorch *v.s.* Gate-Level Netlist

*5.2.1 Experimental Setup.* This case study targets the verification of a fundamental deep learning primitive: a dot product operator applied to two 2-dimensional, 8-bit integer vectors.
**Specification(Pytorch):** We define a high-level `torch.nn.Module` that encapsulates the standard `torch.dot` operation (Listing 4). To bridge the gap between Python-based deep learning frameworks and hardware verification, we leverage the `torch-mlir` compiler infrastructure. This pipeline automatically lowers the PyTorch module into the MLIR **Affine dialect** (Listing 5), preserving the loop structures and memory access patterns required for formal analysis.

```
class DotModule(torch.nn.Module):
    def mm(self, a, b):
        return torch.dot(a, b)

    def forward(self, a, b):
        return self.mm(a, b)
```

**Listing 4: dot.py**

```
module {
    func.func @dot(%arg0: memref<2xi8>, %arg1: memref<2xi8>) ->
    memref<i8> {
        ....
        affine.for %arg2 = 0 to 2 {
            %2 = affine.load %arg0[%arg2] : memref<2xi8>
            %3 = affine.load %arg1[%arg2] : memref<2xi8>
            %4 = arith.muli %2, %3 : i8
            affine.store %4, %alloc[%arg2] : memref<2xi8>
        }
        ...
        return %alloc_1 : memref<i8>
    }
}
```

**Listing 5: dot.mlir**

**Implementation (Netlist):** The hardware implementation begins as a hand-written Register-Transfer Level (RTL) Verilog design (Listing 6). To simulate a realistic backend flow, we employ the open-source synthesis suite Yosys to compile RTL into a flattened gate-level netlist. The synthesis targets *cmos_cells.lib*, a representative standard cell library containing fundamental combinational logic gates, resulting in the structural netlist shown in Listing 7.

```
module dot2_comb #(
    parameter N       = 2,   // Vector length
    parameter W       = 8,   // Bit-width of element
    parameter ACC_W   = 32,  // Bit-width of accumulator
    parameter O_W = 8        // Bit-width of Output
)(
    input  logic signed [N-1:0] [W-1:0] arg_0,
    input  logic signed [N-1:0] [W-1:0] arg_1,
    output logic signed [0:0][W-1:0] out_0
);
    logic signed [N-1:0] [(2*W)-1:0] prod;
    genvar i;
    generate
        for (i = 0; i < N; i = i + 1) begin : GEN_PROD
            assign prod[i] = $signed(arg_0[i]) *
            $signed(arg_1[i]);
        end
    endgenerate
    logic signed [ACC_W-1:0] sum_reg;
    ...   // Accumulate all products into sum_reg.
    assign out_0 = sum_reg[O_W-1:0];
endmodule
```

**Listing 6: dot.v**

```
module dot2_comb(arg_0, arg_1, out_0);
    wire _0000_;
    wire _0001_;
    ...
    NOT _0852_ (.A(arg_0[8]), .Y(_0307_));
    ...
endmodule
```

**Listing 7: netlist.v**

*5.2.2 Verification Process and Results.* We apply EquivFusion to verify the equivalence between the *Affine MLIR (Spec.)* and the synthesized *Netlist (Impl.)* across two distinct precision configurations:
**Experiment I: 8-bit Output Verification.** In the setup, both the PyTorch specification and the Verilog implementation are configured with 8-bit inputs and outputs. The solver returns *UNSAT*, proving the gate-level netlist is functionally equivalent to the high-level PyTorch specification within the constrained 8-bit domain.

**Experiment II: 32-bit Output Verification (Precision Mismatch).** We extend both designs to 32-bit outputs, upon which the solver returns *SAT*, indicating non-equivalence. EquivFusion pinpoints the root cause to the Verilog implementation's sign-extension of intermediate signals: the hardware design sign-extends intermediate values during the calculation, creating a subtle semantic divergence from the torch specification.

## 6 Conclusion

This paper presents EquivFusion, a unified equivalence checking framework that enables formal reasoning across heterogeneous design modalities, from high-level algorithmic specifications to hardware implementations. Built on MLIR and CIRCT, EquivFusion provides an automated, solver-agnostic pipeline that systematically lowers diverse inputs into a common logic representation, eliminating the need for manually maintained reference models. By supporting equivalence checking early in the design flow, the framework facilitates shift-left verification for datapath-intensive designs. As an open-source tool, EquivFusion establishes the first step towards cross-layer formal verification. We plan to integrate it into the CIRCT ecosystem to support standardized equivalence checking within MLIR-based hardware compilation flows.

## References

[1] 2025. CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/.

[2] 2025. MLIR: Multi-Level Intermediate Representation. https://mlir.llvm.org/.

[3] 2025. torch-mlir. https://github.com/llvm/torch-mlir.

[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference*. 1216–1225.

[5] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[6] Armin Biere. 2007. The AIGER and-inverter graph (AIG) format version 20071012. (2007).

[7] Manuel Blum and Hal Wasserman. 2002. Reflections on the Pentium division bug. *IEEE Trans. Comput.* 45, 4 (2002), 385–393.

[8] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. 2008. RTL-TLM equivalence checking based on simulation. In *Proceedings of IEEE East-West Design & Test Symposium (EWDTS'08)*. 214–217. doi:10.1109/EWDTS.2008.5580149

[9] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*. Springer, 24–40.

[10] Cadence Design Systems. 2025. Conformal Equivalence Checker Datasheet. https://www.cadence.com/en_US/home/resources/datasheets/conformal-equivalence-checker-ds.html.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[12] Philippe Coussy and Adam Morawiec. 2010. *High-level synthesis*. Vol. 1. Springer.

[13] Sayantan Das, Rizi Mohanty, Pallab Dasgupta, and Partha Pratim Chakrabarti. 2006. Synthesis of system verilog assertions. In *Proceedings of the Design Automation & Test in Europe Conference*, Vol. 2. IEEE, 1–6.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Amelia Dobis. 2024. *Formal Verification of Hardware using MLIR.* Master's thesis. ETH Zurich.

[16] Rolf Drechsler and Alireza Mahzoon. 2022. Polynomial formal verification: Ensuring correctness under resource constraints. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[17] ABKFM Fleury and Maximilian Heisinger. 2020. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *Sat Competition* 2020 (2020), 50.

[18] Alberto Griggio and Marco Roveri. 2015. Comparing different variants of the IC3 algorithm for hardware model checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 6 (2015), 1026–1039.

[19] Daniel Große, Markus Groß, Ulrich Kühne, and Rolf Drechsler. 2011. Simulation-based equivalence checking between SystemC models at different levels of abstraction. In *Proceedings of the 21st Edition of the Great Lakes Symposium on Great Lakes Symposium on VLSI*. Association for Computing Machinery, New York, NY, USA, 223–228. doi:10.1145/1973009.1973054

[20] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. 2021. PyTorch. In *Programming with TensorFlow: solution for edge computing applications*. Springer, 87–104.

[21] Andreas Kuehlmann and Cornelis A. J. van Eijk. 2002. *Combinational and Sequential Equivalence Checking*. Springer US, Boston, MA, 343–372.

[22] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[23] João P Marques-Silva and Karem A Sakallah. 2000. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*. 675–680.

[24] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[25] Rajdeep Mukherjee, Mitra Purandare, Raphael Polig, and Daniel Kroening. 2017. Formal Techniques for Effective Co-verification of Hardware/Software Co-designs. doi:10.1145/3061639.3062253

[26] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *International Conference on Computer Aided Verification*. Springer, 3–17.

[27] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. 2018. Btor2, btormc and boolector 3.0. In *International Conference on Computer Aided Verification*. Springer, 587–595.

[28] Eli Schwartz, Raja Giryes, and Alex M Bronstein. 2018. Deepisp: Toward learning an end-to-end image processing pipeline. *IEEE Transactions on Image Processing* 28, 2 (2018), 912–923.

[29] Synopsys. 2025. VC Formal Datapath Validation. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html.

[30] Synopsys. 2026. Formality: RTL-to-Gate Equivalence Checking. https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html. Accessed: 2026-01-04.

[31] Cadence Design Systems. 2025. Jasper C Formal Verification. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-c-formal-verification.html.

[32] Yanzhao Wang, Fei Xie, Zhenkun Yang, Pasquale Cocchini, and Jin Yang. 2023. An equivalence checking framework for agile hardware design. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. 26–32.

[33] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-A Free Verilog Synthesis Suite. https://api.semanticscholar.org/CorpusID:202611483

[34] YosysHQ. 2023. EQY: Equivalence Checking with Yosys. https://github.com/YosysHQ/eqy.

## A Walk through EquivFusion

A recorded walkthrough is available at https://youtu.be/AdEeZHU54qA.

### A.1 Installation

EquivFusion is compatible with Linux and macOS environments. The following steps(Listing 8) outline the procedure to clone the repository, configure the build system using CMake and Ninja, and install the required solvers (including AIGER, Bitwuzla and Kissat):

```
# Clone the repository
git clone git@github.com:FORMiND-Lab/EquivFusion.git
cd EquivFusion

# Configure and build the project
mkdir build && cd build
cmake -G Ninja ..

# Build
ninja

# Install solvers
ninja install_solvers

# Add 'EquivFusion/build/bin' to your PATH environment variable
export PATH="$PWD/bin/:$PATH"
```

**Listing 8: Commands to Build and Install EquivFusion**

Building EquivFusion requires git, ninja, python3, cmake, a C++ toolchain, and the readline library, along with the z3 solver in the system PATH.

### A.2 Docker

To facilitate reproducibility and ease of deployment, we provide a Docker-based environment. The Docker image can be built and run using the following commands(Listing 9):

```
$ docker build -t equivfusion .
$ docker run -it equivfusion
```

**Listing 9: Commands to Build and Run Docker**

### A.3 Run Examples

*A.3.1 C++ v.s. Chisel.* This appendix provides the complete source code and execution workflow for the C++ Specification vs Chisel Implementation case study presented in Section 5.1.

**Specification(C++):** The high-level specification is written in C++ (Listing 10), implementing the Bubble Sort algorithm:

```
// Sort.cpp
#include <cstdint>

#define N 8

extern "C" void Sort(unsigned char input[N], unsigned char
output[N]) {
    unsigned char temp[N];

    for (unsigned int i = 0; i < N; i++) {
        temp[i] = input[i];
    }

    for (unsigned int i = N - 1; i > 0; i--) {
        unsigned char high = temp[0];
        unsigned char low = 0;

        for (unsigned int j = 1; j < N; j++) {
            if (j <= i) {
                if (temp[j] > high) {
                    low = high;
                    high = temp[j];
                } else {
                    low = temp[j];
                }
            } else {
                low = temp[j - 1];
            }

            temp[j - 1] = low;
        }

        temp[i] = high;
    }

    for (unsigned int i = 0; i < N; i++) {
        output[i] = temp[i];
    }
}
```

**Listing 10: Sort.cpp**

This C++ source is lowered into the MLIR Affine dialect(Listing 11) using Polygeist:

```
// Sort.mlir
module {
  func.func @Sort(%arg0: memref<8xi8> {polygeist.param_name =
  "input"}, %arg1: memref<8xi8> {polygeist.param_name =
  "output"}) attributes {llvm.linkage = #llvm.linkage<external>}
  {
    %c8 = arith.constant 8 : index
    %alloca = memref.alloca() : memref<8xi8>
    affine.for %arg2 = 0 to 8 {
      %0 = affine.load %arg0[%arg2] : memref<8xi8>
      affine.store %0, %alloca[%arg2] : memref<8xi8>
    }
    affine.for %arg2 = 1 to 8 {
      %0 = arith.subi %c8, %arg2 : index
      %1 = arith.index_cast %0 : index to i32
      %2 = affine.load %alloca[0] : memref<8xi8>
      %3 = affine.for %arg3 = 1 to 8 iter_args(%arg4 = %2) ->
      (i8) {
        %4 = arith.index_cast %arg3 : index to i32
        %5 = arith.cmpi ule, %4, %1 : i32
        %6:2 = scf.if %5 -> (i8, i8) {
          %7 = affine.load %alloca[%arg3] : memref<8xi8>
          %8 = arith.extui %7 : i8 to i32
          %9 = arith.extui %arg4 : i8 to i32
          %10 = arith.cmpi sgt, %8, %9 : i32
          %11 = arith.select %10, %arg4, %7 : i8
          %12 = arith.select %10, %7, %arg4 : i8
          scf.yield %11, %12 : i8, i8
        } else {
          %7 = affine.load %alloca[%arg3 - 1] : memref<8xi8>
          scf.yield %7, %arg4 : i8, i8
        }
        affine.store %6#0, %alloca[%arg3 - 1] : memref<8xi8>
        affine.yield %6#1 : i8
      }
      affine.store %3, %alloca[-%arg2 + 8] : memref<8xi8>
    }
    affine.for %arg2 = 0 to 8 {
      %0 = affine.load %alloca[%arg2] : memref<8xi8>
      affine.store %0, %arg1[%arg2] : memref<8xi8>
    }
    return
  }
}
```

**Listing 11: Sort.mlir**

**Implementation(Chisel):** The hardware implementation is described in Chisel (Listing 12), implementing a Bitonic Sorter:

```
// Sort.scala
import chisel3._
import circt.stage.ChiselStage
```

```

class CompareAndSwap(width: Int) extends RawModule {
  val io = IO(new Bundle {
    val a = Input(UInt(width.W))
    val b = Input(UInt(width.W))
    val min = Output(UInt(width.W))
    val max = Output(UInt(width.W))
  })

  when(io.a <= io.b) {
    io.min := io.a
    io.max := io.b
  }.otherwise {
    io.min := io.b
    io.max := io.a
  }
}

class Sort(width: Int = 8) extends RawModule {
  val input = IO(Input(Vec(8, UInt(width.W))))
  val output = IO(Output(Vec(8, UInt(width.W))))

  def CAS(a: UInt, b: UInt): (UInt, UInt) = {
    val m = Module(new CompareAndSwap(width))
    m.io.a := a
    m.io.b := b
    (m.io.min, m.io.max)
  }

  // === Stage 1 ===
  val (s1_0, s1_1) = CAS(input(0), input(1))
  val (s1_3, s1_2) = CAS(input(2), input(3)) // Swap order for
  bitonic merge
  val (s1_4, s1_5) = CAS(input(4), input(5))
  val (s1_7, s1_6) = CAS(input(6), input(7)) // Swap order

  // === Stage 2 ===
  val (s2_0_t, s2_2_t) = CAS(s1_0, s1_2)
  val (s2_1_t, s2_3_t) = CAS(s1_1, s1_3)
  val (s2_0, s2_1) = CAS(s2_0_t, s2_1_t)
  val (s2_2, s2_3) = CAS(s2_2_t, s2_3_t)

  val (s2_6_t, s2_4_t) = CAS(s1_4, s1_6) // Descending merge
  val (s2_7_t, s2_5_t) = CAS(s1_5, s1_7)
  val (s2_5, s2_4) = CAS(s2_4_t, s2_5_t)
  val (s2_7, s2_6) = CAS(s2_6_t, s2_7_t)

  // === Stage 3 ===
  val (s3_0_t1, s3_4_t1) = CAS(s2_0, s2_4)
  val (s3_1_t1, s3_5_t1) = CAS(s2_1, s2_5)
  val (s3_2_t1, s3_6_t1) = CAS(s2_2, s2_6)
  val (s3_3_t1, s3_7_t1) = CAS(s2_3, s2_7)

  val (s3_0_t2, s3_2_t2) = CAS(s3_0_t1, s3_2_t1)
  val (s3_1_t2, s3_3_t2) = CAS(s3_1_t1, s3_3_t1)
  val (s3_4_t2, s3_6_t2) = CAS(s3_4_t1, s3_6_t1)
  val (s3_5_t2, s3_7_t2) = CAS(s3_5_t1, s3_7_t1)

  val (o0, o1) = CAS(s3_0_t2, s3_1_t2)
  val (o2, o3) = CAS(s3_2_t2, s3_3_t2)
  val (o4, o5) = CAS(s3_4_t2, s3_5_t2)
  val (o6, o7) = CAS(s3_6_t2, s3_7_t2)

  output(0) := o0
  output(1) := o1
  output(2) := o2
  output(3) := o3
  output(4) := o4
  output(5) := o5
  output(6) := o6
  output(7) := o7
}

object Sort extends App {
  (new ChiselStage).execute(args,
  Seq(chisel3.stage.ChiselGeneratorAnnotation(() => new
  Sort(8))))
}
```

**Listing 12: Sort.scala**

This design is compiled into the FIRRTL intermediate representation (Listing 13) using sbt:

```
FIRRTL version 3.3.0
circuit Sort :%[[
  {
    "class":"firrtl.transforms.DedupGroupAnnotation",
    "target":"~Sort|CompareAndSwap",
    "group":"CompareAndSwap"
  },

  ......

  {
    "class":"firrtl.transforms.DedupGroupAnnotation",
    "target":"~Sort|Sort",
    "group":"Sort"
  }
]]
  module CompareAndSwap :
    output io : { flip a : UInt<8>, flip b : UInt<8>, min :
    UInt<8>, max : UInt<8>}

    node _T = leq(io.a, io.b)
    when _T :
      connect io.min, io.a
      connect io.max, io.b
    else :
      connect io.min, io.b
      connect io.max, io.a

......

  module Sort :
    input input : UInt<8>[8]
    output output : UInt<8>[8]

    inst m of CompareAndSwap
    connect m.io.a, input[0]
    connect m.io.b, input[1]

  ......

    connect output[4], m_22.io.min
    connect output[5], m_22.io.max
    connect output[6], m_23.io.min
    connect output[7], m_23.io.max
```

**Listing 13: Sort.fir**

**Execution and Results:** To perform the equivalence check, the following commands(Listing 14) are executed in *equiv_fusion*:

```
read_c -spec -top Sort Sort.mlir
read_fir -impl -top Sort Sort.fir --scalarize-public-modules
false --scalarize-public-modules false --preserve-aggregate all
equiv_miter -specModule Sort -implModule Sort -mitermode aiger
-o miter.aiger
solver_runner --solver kissat --inputfile miter.aiger
```

**Listing 14: Commands to Run C++ *v.s.* Chisel Equivalence Checking**

The solver returns *UNSAT*, proving that the Chisel implementation is functionally equivalent to the C++ specification despite algorithmic differences.

*A.3.2 Pytorch v.s. Netlist.* This appendix details the complete source artifacts and verification workflow for the PyTorch Specification vs Gate-Level Netlist case study presented in Section5.2.
**Specification(Pytorch):** The high-level specification is defined as a PyTorch module (Listing 15) performing a dot product operation:

```
# dot.py
from typing import List
```

```python

import torch
import torch.nn as nn

from torch_mlir import fx

import os
import sys

class DotModule(torch.nn.Module):
  def mm(self, a, b):
    return torch.dot(a, b)

  def forward(self, a, b):
    return self.mm(a, b)

model = DotModule()

x = torch.randint(low=-128, high=128, size=(2,),
dtype=torch.int8)
y = torch.randint(low=-128, high=128, size=(2,),
dtype=torch.int8)

m = fx.export_and_import(model, x, y,
output_type="linalg-on-tensors", func_name="dot")

current_script_path = os.path.abspath(__file__)
current_script_dir = os.path.dirname(current_script_path)

with open(os.path.join(current_script_dir, "dot.mlir"), "w") as
f:
    f.write(str(m))
```

**Listing 15: dot.py**

This PyTorch model is lowered into the MLIR Affine dialect using the torch-mlir infrastructure combined with upstream MLIR tools. The resulting MLIR code (Listing 16) serves as the input specification for EquivFusion:

```
// dot.mlir
module {
  func.func @dot(%arg0: memref<2xi8>, %arg1: memref<2xi8>) ->
  memref<i8> {
    %c0_i64 = arith.constant 0 : i64
    %alloc = memref.alloc() {alignment = 64 : i64} : memref<2xi8>
    affine.for %arg2 = 0 to 2 {
      %2 = affine.load %arg0[%arg2] : memref<2xi8>
      %3 = affine.load %arg1[%arg2] : memref<2xi8>
      %4 = arith.muli %2, %3 : i8
      affine.store %4, %alloc[%arg2] : memref<2xi8>
    }
    %alloc_0 = memref.alloc() {alignment = 64 : i64} :
    memref<i64>
    affine.store %c0_i64, %alloc_0[] : memref<i64>
    affine.for %arg2 = 0 to 2 {
      %2 = affine.load %alloc[%arg2] : memref<2xi8>
      %3 = affine.load %alloc_0[] : memref<i64>
      %4 = arith.extsi %2 : i8 to i64
      %5 = arith.addi %4, %3 : i64
      affine.store %5, %alloc_0[] : memref<i64>
    }
    %alloc_1 = memref.alloc() {alignment = 64 : i64} : memref<i8>
    %0 = affine.load %alloc_0[] : memref<i64>
    %1 = arith.trunci %0 : i64 to i8
    affine.store %1, %alloc_1[] : memref<i8>
    return %alloc_1 : memref<i8>
  }
}
```

**Listing 16: dot.mlir**

**Implementation(Netlist):** The hardware implementation originates from a SystemVerilog design (Listing 17):

```verilog
// dot.v
module dot2_comb #(
    parameter N       = 2,   // Vector length
    parameter W       = 8,   // Bit-width of each element (signed)
    parameter ACC_W   = 32,    // Bit-width of accumulator
    parameter O_W = 8          // Bit-width of Output
)(
    // Two signed input vectors, each containing N elements of W
    bits
    input  logic signed [N-1:0] [W-1:0] arg_0,
    input  logic signed [N-1:0] [W-1:0] arg_1,

    // Signed output: dot product result (sum of element-wise
    products)
    output logic signed [0:0][W-1:0] out_0
);


    //============================================================
    // 1. Element-wise multiplications
    // Each pair a[i], b[i] is multiplied to produce a 2*W-bit
    product.

    //============================================================
    logic signed [N-1:0] [(2*W)-1:0] prod;

    genvar i;
    generate
        for (i = 0; i < N; i = i + 1) begin : GEN_PROD
            assign prod[i] = $signed(arg_0[i]) *
            $signed(arg_1[i]);
        end
    endgenerate


    //============================================================
    // 2. Summation of all products
    // This always block performs a combinational reduction (sum)
    // across all products. The result is stored in sum_reg.

    //============================================================
    integer k;
    logic signed [ACC_W-1:0] sum_reg;

    always @(*) begin
        sum_reg = '0; // Initialize accumulator to zero
        for (k = 0; k < N; k = k + 1) begin
            // Sign-extend each 2*W-bit product to ACC_W bits
            // before adding, to prevent overflow and preserve
            sign.
            sum_reg = sum_reg +
                {{(ACC_W-(2*W)){prod[k][(2*W)-1]}}, prod[k]};
        end
    end


    //============================================================
    // 3. Final output assignment
    // The dot product is simply the total accumulated sum.

    //============================================================
    assign out_0 = sum_reg[O_W-1:0];

endmodule
```

**Listing 17: dot.v**

To synthesize this design into a gate-level netlist, we utilize Yosys with a standard cell library *cmos_cells.lib*(Listing 18) provided in the Yosys examples. The corresponding Verilog simulation model for the cells is *cmos_cells.v*(Listing 19):

```
// cmos_cells.lib
library(demo) {
  cell(BUF) {
    area: 6;
    pin(A) { direction: input; }
    pin(Y) { direction: output;
            function: "A"; }
  }
  cell(NOT) {
    area: 3;
    pin(A) { direction: input; }
    pin(Y) { direction: output;
             function: "A'"; }
  }
  cell(NAND) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output;
            function: "(A*B)'"; }
  }
  cell(NOR) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output;
            function: "(A+B)'"; }
  }
  cell(DFF) {
    area: 18;
    ff(IQ, IQN) { clocked_on: C;
                next_state: D; }
    pin(C) { direction: input;
               clock: true; }
    pin(D) { direction: input; }
    pin(Q) { direction: output;
             function: "IQ"; }
  }
  cell(DFFSR) {
    area: 18;
    ff("IQ", "IQN") { clocked_on: C;
                next_state: D;
                    preset: S;
                     clear: R; }
    pin(C) { direction: input;
               clock: true; }
    pin(D) { direction: input; }
    pin(Q) { direction: output;
             function: "IQ"; }
    pin(S) { direction: input; }
    pin(R) { direction: input; }
    ; // empty statement
  }
}
```

**Listing 18: cmos_cells.lib**

```
// cmos_cells.v
module BUF(A, Y);
input A;
output Y;
assign Y = A;
endmodule

module NOT(A, Y);
input A;
output Y;
assign Y = ~A;
endmodule

module NAND(A, B, Y);
input A, B;
output Y;
assign Y = ~(A & B);
endmodule

module NOR(A, B, Y);
input A, B;
output Y;
assign Y = ~(A | B);
endmodule

module DFF(C, D, Q);
input C, D;
output reg Q;
always @(posedge C)
        Q <= D;
endmodule

module DFFSR(C, D, Q, S, R);
input C, D, S, R;
output reg Q;
always @(posedge C, posedge S, posedge R)
        if (S)
                Q <= 1'b1;
        else if (R)
                Q <= 1'b0;
        else
                Q <= D;
endmodule
```

**Listing 19: cmos_cells.v**

The synthesis is performed by executing the following Yosys passes(Listing 20):

```
read_verilog -sv dot.v
hierarchy -check -top top

proc; opt; fsm; opt; memory; opt

techmap; opt

dfflibmap -liberty cmos_cells.lib

abc -liberty cmos_cells.lib

opt_clean

write_verilog netlist.v
```

**Listing 20: Yosys Passes to Synthesized Netlist**

This process generates the gate-level netlist (Listing 21). A truncated snippet of the netlist is shown below:

```
// netlist.v
/* Generated by Yosys 0.53+98 (git sha1 50b63c648, g++
13.3.0-6ubuntu2~24.04 -Og -fPIC) */

(* dynports =  1  *)
(* top =  1  *)
(* src = "../../verilog/dot2_8/dot2.v:1.1-52.10" *)
module dot2_comb(arg_0, arg_1, out_0);
  wire _0000_;
  wire _0001_;
  wire _0002_;
  wire _0003_;
  wire _0004_;
  wire _0005_;
  wire _0006_;
  wire _0007_;
  wire _0008_;
  wire _0009_;
  wire _0010_;

  ......

  wire [31:0] sum_reg;
  NOT _0852_ (
    .A(arg_0[8]),
    .Y(_0307_)
  );
  NOT _0853_ (
    .A(arg_1[15]),
    .Y(_0318_)
  );
  NAND _0854_ (
    .A(arg_0[0]),
    .B(arg_1[0]),
    .Y(_0329_)
  );
  NAND _0855_ (
    .A(arg_0[8]),
    .B(arg_1[8]),
      .Y(_0340_)
    );

    ......

      NOT _1710_ (
      .A(_0840_),
      .Y(_0841_)
    );
    NOR _1711_ (
      .A(_0351_),
      .B(_0841_),
      .Y(out_0[0])
    );
    assign k = 32'd2;
    assign sum_reg[7:0] = out_0;
  endmodule
```

**Listing 21: netlist.v**

**Execution and Results:** To verify the equivalence between the generated MLIR specification and the synthesized netlist, the following commands(Listing 22) are executed in *equiv_fusion*:

```
read_c -spec -top dot dot.mlir
read_v -impl -top dot2_comb netlist.v cmos_cells.v
equiv_miter -specModule dot -implModule dot2_comb -mitermode
btor2 -o miter.btor2
solver_runner --solver bitwuzla --inputfile miter.btor2
```

**Listing 22: Commands to Run Pytorch *v.s.* Netlist Equivalence Checking**

The result *UNSAT* confirms functional equivalence under the 8-bit configuration.

## A.4 Debugging

To ensure robustness and facilitate development, EquivFusion provides a rigorous testing infrastructure and fine-grained debugging utilities:

**Continuous Integration (CI):** A GitHub Actions pipeline enforces code stability by automatically validating build success and executing a comprehensive suite of integration tests on every pull request, preventing functional regression.

**equivfusion-hls:** This utility isolates the HLS flow execution, enabling developers to inspect the intermediate MLIR IR generated after each transformation pass for deep analysis of the lowering process.

**equivfusion-opt:** Modeled after mlir-opt, this tool allows for the isolated invocation of individual EquivFusion passes, facilitating targeted debugging and the development of custom optimization logic.

## B Runtime Analysis

Table 1 details the execution time for each command in the verification workflows of the examples presented in Appendix A.3.

## C Synthesizable Subset and Constraints

For high-level languages like C/C++ and PyTorch, EquivFusion targets a specific synthesizable subset conducive to High-Level Synthesis (HLS), with the following specific constraints:

**Data Types:** Support is limited to integer arithmetic and fixed-width bit-vectors. Floating-point support is planned via future emulation library integration.

**Table 1: Execution Time(seconds)**

| Stage | bitwuzla smt | bitwuzla btor2 | kissat aiger |
|---|---|---|---|
| **Execution Time (UNSAT): Sort.scala vs Sort.cpp** | | | |
| **set_port** | 0.000006 | 0.000006 | 0.000006 |
| **read_c** | 0.017226 | 0.013448 | 0.013090 |
| **read_firrtl** | 0.060061 | 0.073934 | 0.067827 |
| **equiv_miter** | 0.010399 | 0.008282 | 0.031100 |
| **solver_runner** | 47.54907 | 55.82388 | 45.67969 |
| **Execution Time (SAT): Sort.scala vs Sort.cpp** | | | |
| **set_port** | 0.000005 | 0.000006 | 0.000005 |
| **read_c** | 0.013120 | 0.012417 | 0.012875 |
| **read_firrtl** | 0.055784 | 0.079103 | 0.052333 |
| **equiv_miter** | 0.006621 | 0.007322 | 0.027125 |
| **solver_runner** | 0.007440 | 0.013190 | 0.018456 |
| **Execution Time (UNSAT): Dot.py vs netlist.v** | | | |
| **read_c** | 0.003391 | 0.003289 | 0.003449 |
| **read_v** | 0.049022 | 0.049296 | 0.052890 |
| **equiv_miter** | 0.023631 | 0.013858 | 0.022719 |
| **solver_runner** | 2.776137 | 3.013583 | 1.800442 |
| **Execution Time (SAT): Dot.py vs netlist.v** | | | |
| **read_c** | 0.003769 | 0.003429 | 0.005344 |
| **read_v** | 0.251780 | 0.278213 | 0.262784 |
| **equiv_miter** | 0.058312 | 0.037827 | 0.066539 |
| **solver_runner** | 0.039862 | 0.041944 | 0.055321 |

**Control Flow (Branching):** Standard conditionals (e.g., if-else) are supported if statically resolvable or mappable to multiplexers.

**Control Flow (Loops):** Loop support is strictly limited to static affine *for* loops with compile-time determinable bounds and constant strides. Unstructured control flow (e.g., *break, continue*) is prohibited to ensure deterministic unrolling.

**Memory & Pointers:** Memory operations use statically allocated arrays. Pointers require fixed sizes and explicit indexing, and pointer reassignment within conditional scopes is prohibited to ensure Static Single Assignment (SSA) consistency.

## D More Cases

### D.1 Dot64 (C++ v.s. Chisel)

This case demonstrates the end-to-end EquivFusion workflow for verifying equivalence between a high-level algorithmic C++ model and a Chisel design, using the 64-element dot product (dot64) computation as a canonical benchmark.

**Specification (Chisel):** FIRRTL, derived from the Chisel design(Listing 23) via Chisel elaboration.

```
class Dot64 extends RawModule {
    val arg_0 = IO(Input(Vec(64, SInt(16.W))))
    val arg_1 = IO(Input(Vec(64, SInt(16.W))))
    val out_0 = IO(Output(SInt(64.W)))
    var sum = 0.S(64.W)
    for (i <- 0 until 64) {
        val product = arg_0(i) * arg_1(i)
        sum = sum + product
    }
    out_0 := sum
}
```

**Listing 23: Dot64.scala**

**Implementation (C++):** MLIR, derived from the high-level C++ algorithm via Polygeist.
`Dot64.cpp`(Listing 24): Fully consistent with Chisel specification.

```
extern "C" int64_t Dot64(const int16_t (&arg_0)[64], const
int16_t (&arg_1)[64]) {
    int64_t sum = 0;
    for (int i = 0; i < 64; ++i) {
        int32_t product = arg_0[i] * arg_1[i];
        sum += product;
    }
    return sum;
}
```

**Listing 24: Dot64.cpp**

`Dot64_truncation.cpp`(Listing 25): Contains a bug: product truncation to 16 bits.

```
        int16_t product = arg_0[i] * arg_1[i];
```

**Listing 25: Dot64_truncation.cpp**

**Verification Results:**

- **Dot64.scala vs Dot64.cpp**
  The solver returns UNSAT, proving functional equivalence.
- **Dot64.scala vs Dot64_truncation.cpp**
  The solver returns SAT, indicating non-equivalence.

## E Future Directions

### E.1 Intelligent Solver Orchestration

Building on the foundational `solver_runner` module—which currently orchestrates Z3, Bitwuzla, and Kissat—future efforts will focus on evolving the orchestration engine from a static dispatcher into an adaptive, intelligent decision-making layer. We target three key enhancements:
**Expanded Verification Engine Integration:** We aim to incorporate a broader spectrum of domain-specific engines to cover diverse verification needs. This includes integrating **ABC** for optimized combinational equivalence checking and **CVC5** for handling more complex theories.
**Adaptive Solver Selection and Configuration:** Different solvers and configurations exhibit vastly different performance characteristics depending on the circuit structure (e.g., arithmetic-heavy vs. control-heavy). To optimize verification throughput, we plan to implement automated selection mechanisms: **Rule-Based Heuristics:** Leveraging static analysis of the IR to extract circuit features (e.g., bit-width distribution, logic depth, operator types). Based on these signatures, the system will apply expert rules to dispatch tasks (e.g., routing purely boolean logic to AIGER-based SAT solvers like Kissat, while directing complex word-level arithmetic to Bitwuzla).
**AI-Driven Optimization:** We explore employing Machine Learning (ML) models, such as Graph Neural Networks (GNNs), to embed circuit netlists and predict the most efficient solver for a given instance. Furthermore, we intend to utilize Bayesian Optimization or Reinforcement Learning to automatically tune solver hyperparameters (e.g., restart strategies, decision heuristics) dynamically, minimizing solving time without manual intervention.
**Parallel Portfolio Execution:** Beyond sequential invocation, we will implement a parallel portfolio strategy. This involves launching multiple solvers (or the same solver with distinct random seeds/configurations) concurrently on multi-core systems. The orchestration layer will terminate all processes as soon as the fastest solver returns a result, thereby reducing the latency of the "straggler" effect in hard verification instances.

### E.2 Floating-Point Verification via Soft-Float Emulation

While the current iteration of EquivFusion focuses on integer arithmetic, extending verification capabilities to floating-point domains is a key objective. We propose integrating a canonical floating-point emulation library (e.g., SoftFloat) to bridge this gap. The envisaged approach involves lowering high-level floating-point IR operations (e.g., `arith.addf`, `arith.mulf`) into function calls that invoke verified software implementations of IEEE-754 compliant operators. By replacing native floating-point instructions with bit-precise integer logic from the emulation library, EquivFusion will enable rigorous bit-level equivalence checking between algorithmic specifications and hardware FPUs. This strategy effectively bypasses the complexity of floating-point theories, allowing standard bit-vector solvers to reason about floating-point semantics.

### E.3 Supporting Hand-Written Assumes and Lemmas

Although EquivFusion targets fully automated equivalence checking, the semantic gap between high-level algorithms and low-level RTL can sometimes exceed the capabilities of automatic inference—particularly when dealing with complex loop unrolling, retiming, or aggressive synthesis optimizations. To address these challenges, a promising future direction is to support **user-guided verification** through hand-written *assumes* and *lemmas*. **Assumes** define critical environmental constraints or input invariants (e.g., valid control signal ranges or loop bounds) under which equivalence is expected to hold, effectively pruning the solver's search space. *Lemmas* capture intermediate semantic properties, such as algorithmic invariants or correspondence points between pipeline stages. These can serve as "checkpoints" to guide the solver in state alignment.
This proposed workflow draws inspiration from the methodology employed in Synopsys Hector [29]. Similar to Hector's approach of leveraging user insights to decompose monolithic proofs, EquivFusion aims to translate these user-provided annotations into solver constraints and proof sub-goals. By incorporating verified lemmas to guide equivalence partitioning and cross-level state alignment, we seek to combine the scalability of automated solving with the precision of manual guidance, without altering the core verification pipeline.