Assassyn: A Unified Abstraction for Architectural Simulation and Implementation

Jian Weng*
KAUST
Thuwal, Saudi Arabia
jian.weng@kaust.edu.sa

Boyang Han†
Hong Kong University
Hong Kong, Hong Kong
yqszxx@gmail.com

Derui Gao
KAUST
Thuwal, Saudi Arabia
derui.gao@kaust.edu.sa

Ruijie Gao†
University of Michigan
Ann Anbor, USA
ruijieg@umich.edu

Wanning Zhang
KAUST
Thuwal, Saudi Arabia
wanning.zhang@kaust.edu.sa

An Zhong
KAUST
Thuwal, Saudi Arabia
an.zhong@kaust.edu.sa

Ceyu Xu
HKUST
Hong Kong, Hong Kong
eeentropy@ust.hk

Jihao Xin
KAUST
Thuwal, Saudi Arabia
jihao.xin@kaust.edu.sa

Yangzhixin Luo
KAUST
Thuwal, Saudi Arabia
yangzhixin.luo@kaust.edu.sa

Lisa Wu Wills
Duke University
Durham, USA
lisa@cs.duke.edu

Marco Canini
KAUST
Thuwal, Saudi Arabia
marco@kaust.edu.sa

Abstract

The continuous growth of on-chip transistors driven by technology scaling urges architecture developers to design and implement novel architectures to effectively utilize the excessive on-chip resources. Due to the challenges of programming in register-transfer level (RTL) languages, performance modeling based on simulation is typically developed alongside hardware implementation, allowing the exploration of high-level design decisions before dealing with the error-prone, low-level RTL details. However, this approach also introduces new challenges in coordinating across multiple teams to align implementation details separate codebases.

In this paper, we address this issue by presenting Assassyn, a *unified*, *high-level*, and *general-purpose* programming framework for architectural simulation and implementation. By taking advantage of the concept of asynchronous event handling, a widely existing behavior in both hardware design and implementation and software engineering, a *general-purpose*, and *high-level* programming abstraction is proposed to mitigate the difficulties of RTL programming. Moreover, the *unified* programming interface naturally enables an accurate and faithful alignment between the simulation-based performance modeling and RTL implementation.

Our evaluation demonstrates that Assassyn's high-level programming interface is sufficiently expressive to implement a wide range of architectures, from architectural components, and application-specific accelerators, to designs as complicated as out-of-order CPUs. All the generated simulators perfectly align with the generated RTL behavior, while achieving 2.2-8.1× simulation speedup, and requiring 70% lines of code. The generated RTL achieves comparable perf/area compared to handcrafted RTL, and 6× perf/area compared to high-level synthesis generated RTL code by introducing by mean 1.26× lines of code overhead.

CCS Concepts

• **Hardware** → **Hardware description languages and compilation**; • **Computer systems organization** → *High-level language architectures*; *Data flow architectures*; *Pipeline computing*.

Keywords

Performance Modeling and Simulation, Open-source Hardware, High-level Hardware Description Language

ACM Reference Format:
Jian Weng, Boyang Han, Derui Gao, Ruijie Gao, Wanning Zhang, An Zhong, Ceyu Xu, Jihao Xin, Yangzhixin Luo, Lisa Wu Wills, and Marco Canini. 2025. Assassyn: A Unified Abstraction for Architectural Simulation and Implementation. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3695053.3731004

*Serve as both the first and correspondence author.
†Both participated in this work at KAUST as research interns.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731004

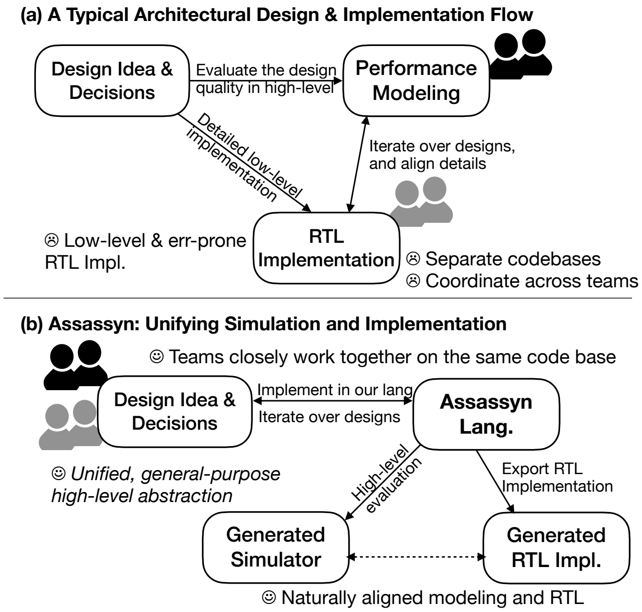


Figure 1: A typical flow of architectural design and implementation compared to our goal, designing and implementing in a *unified, high-level, and general-purpose* interface.

1 Introduction

Technology scaling continues to provide abundant on-chip transistors, urging developers to create innovative architectures to effectively utilize these resources. These architectural innovations have attracted significant attention from both industry and academia for their potential to deliver remarkable performance gains and energy efficiency. However, realizing these innovations is no small feat; as shown in Figure 1(a), transforming an architectural design into a hardware implementation demands a lengthy and complex process involving collaboration across multiple teams. The excessive low-level details exposed by Register-Transfer Level (RTL) language significantly hinder the design space and decision exploration of an architecture. To address this, simulation-based performance models are often developed alongside hardware implementations, so that high-level design ideas and decisions can be evaluated before confronting the challenges of error-prone RTL language.

However, this two-pronged approach introduces a significant challenge: maintaining and aligning simulation-based modeling and RTL implementation of the target architecture imposes substantial coordination burdens across multiple teams. This difficulty stems from using different codebases and programming languages for performance modeling and RTL implementation, leading to potential inconsistencies and increased development time.

Prior works [5, 7, 10, 11, 13, 16, 19, 21, 23–26, 29, 31, 33–35, 37, 40, 41, 45, 49, 52–54, 60, 62–64, 66, 69, 70, 75, 76, 79, 81] [1] follow this design and implementation paradigm, reporting performance numbers from simulation-based modeling, and the power/area evaluations from separate synthesized RTL implementations. However, the alignment between the simulation and hardware implementation is often overlooked, highlighting a persistent challenge of this two-pronged approach.

Our Goal (cf. Figure 1(b)): Recognizing these limitations in current practices, we propose that an ideal architectural design and implementation workflow shall have a *unified, general-purpose, and high-level* language to describe the architecture design to generate both cycle-accurate simulation and RTL implementation. This approach enables multiple teams to collaborate closely, iterating on high-level design decisions and implementation details together, while seamlessly maintaining alignment between performance modeling and RTL implementation. The generated RTL implementation should have minimized overhead caused by high-level abstraction. Prior works [27, 65, 68, 72] automate architecture design and implementation by adopting *high-level* programming interfaces that *unify* the software development and hardware generation; however, they all failed to be *general-purpose* — their target application domain or underlying hardware is limited within a designated scope.

Our Approach: In light of these limitations, we propose Assassyn[2], a fundamentally different approach that *unifies* the *general-purpose* hardware design and implementation in a *high-level* programming abstraction. Our approach is enabled by generalizing the hardware's behaviors: Pipelined designs are widely adopted to improve the performance and frequency in modern architectures, and our insight is that both simulating and implementing such pipelined architectures can be abstracted as *asynchronous event handling*. By leveraging several widely adopted concepts in functional programming (high-order functions and bind) and software engineering (async event handling), a *high-level and general* language for pipelined hardware description is developed to better manage the error-prone and tedious efforts in RTL programming, and explore the high-level design parameters decoupled from the architectural designs. The key contributions of Assassyn are:

- Recognizing a generalized paradigm for pipelined architectures' design and implementation.
- Based on this generalized paradigm, a *unified, high-level, and general-purpose* programming language is proposed for hardware simulation and implementation. To the best of our knowledge, this is the **first** RTL generator that is both *high-level, and general-purpose.*
- By integrating a fully open-source workflow, from frontend language to synthesis tool, and technique library, an end-to-end flow is presented to maximize the reproducibility of architectural research.

Our evaluation shows that Assassyn's programming abstraction is: 1. highly productive to easily target architecture designs and implementations while requiring only 70% of lines of code; 2. sufficiently expressive to target diverse prior designs, from design components to end-to-end accelerators and CPUs; 3. efficient to generate RTL implementation with comparable quality compared to handcrafted designs.

The rest of the paper is organized as follows: the backgrounds on the existing approach on hardware design and implementation will be overviewed in Section 2 to motivate our approach; the technical

[1] All these papers are from a single conference proceedings, ISCA 2024.

[2] Assassyn is the acronym for asynchronous semantics for architectural simulation and synthesis.

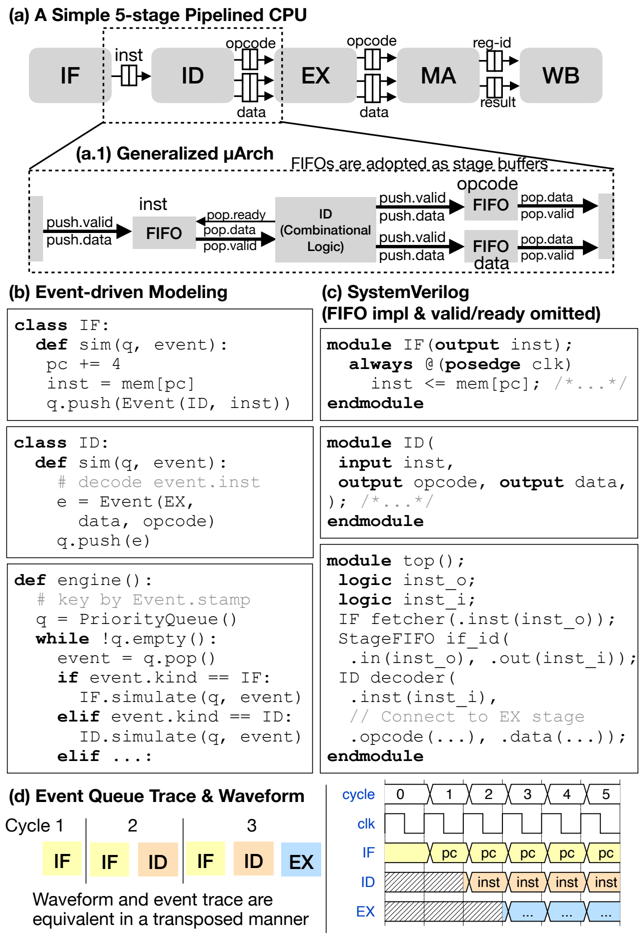


Figure 2: A 5-stage pipeline evaluated in both an event-driven simulator, and an RTL simulator.

details of Assassyn will be explained in in Section 3, 4, and 5, including the programming abstraction, compiler transformations, and the runtime and μ-architectural support. Finally, the experiment setup and the evaluation will be presented in Section 6 and 7 so that we can discuss and conclude this work in Section 8.

2 Background & Motivation

In this section, we first overview the existing technologies for hardware modeling and implementation, highlighting their challenges and opportunities, to motivate our proposed *unified, high-level, and general-purpose* programming interface for both simulation-based modeling, and pipeline description.

2.1 Pipeline Design & Implementation

Pipelining is widely adopted in modern architectures to improve the performance by separating interdependent logic into multiple stages, so more transistors remain active simultaneously. While promising, this architectural paradigm imposes additional implementation challenges in managing the communications and timing among pipeline stages. To mitigate these challenges, before engaging with these low-level and error-prone details exposed in the RTL

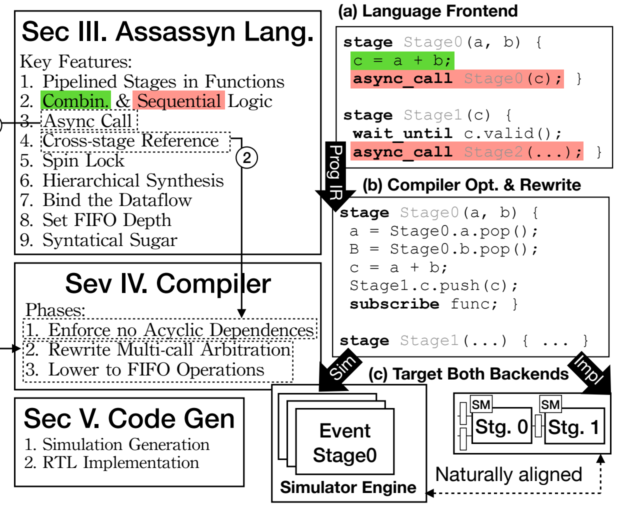


Figure 3: An overview to Assassyn.

language, simulation-based performance models are often built alongside to explore the high-level design decisions.

Simulation-based Hardware Modeling: Many prior works' performances were modeled by either implementing in-house [11, 19, 23, 29, 62] or extending existing open-source [9, 14, 30, 36, 48, 61] simulators. These *simulation-based* models provide detailed performance insight into the architectural pipeline by tracking the state and behaviors of each pipeline stage in each cycle.

As illustrated in Figure 2(b), event-driven simulation is a typical approach of simulating a 5-stage CPU pipeline by maintaining an event queue. Each stage's functionality is simulated by popping and processing the corresponding event. Notably, each stage triggers the next stages by pushing an event instance associated with its inputs to the queue, which is similar to *asynchronously* invoking a function. For example, the `IF` stage enqueues an event for the `ID` stage with fetched instruction associated, while `ID`'s simulation will not be occur until the next cycle, which is akin to invoking `ID.sim` *asynchronously*.

The event engine terminates once the queue is empty, or it reaches a predefined cycle threshold. This simulation-based approach allows developers to rapidly explore design ideas and decisions at a high level, but unrealistic assumptions may significantly compromise the accuracy of simulation results [51]. In addition, the fundamental differences between the programming models cause a nontrivial effort to bridge the simulated designs to actual RTL implementations.

Hardware Implementation: To mitigate the difficulties in programming low-level RTL language, prior works automate architecture design and implementation by trading off the generality. For instance, with a designated architectural paradigm, highly specialized designs can be generated for the given sets of the target applications, by tuning the parameters and adjusting the topology of the template architecture [39, 56, 72]. Some other works [27, 65, 68] focus on particular application domains or constructs, such as perfect loop nests, allowing these frameworks to achieve performance comparable to or even surpassing hand-crafted designs. However, when an application or architecture design falls outside the framework scope, developers must either extend the framework to accommodate new designs or revert to manual implementations.

A RTL language, such as SystemVerilog, offers full control over the underlying hardware implementation, from the circuit structure to the cycle timing. While this fine-grain control is necessary for many optimal designs, exposing excessive low-level details also make the programming process complex and error-prone. Commercial and open-source RTL projects, like Bluespec Verilog [1], and Chisel [6], attempted to alleviate these difficulties by offering syntactic sugar and encapsulated APIs, but still, these tools adhere closely to the RTL's programming and execution model, leaving the fundamental challenges on dealing with timing, concurrency, and cycle-carried state machines unresolved.

Challenges and Opportunities: In contrast to event-driven simulation, which *pushes* data forward through pipeline stages, RTL programming follows an event-listen paradigm where stages *pull* in data to process their state machines. Figure 2(c) shows the `IF` stage listens to `clk` signal to fetch instructions, and the `ID` stage listens to fetched `inst` from `IF`. This *pull/push* mismatch creates a significant gap between architecture design and implementation. However, Figure 2(d) suggests that there exists a fundamental correspondence between simulation traces and waveforms when viewed in a transposed manner. This correspondence presents an opportunity to create a unified abstraction to bridge the simulation and RTL implementation.

2.2 Event-driven Programming

Based on the correspondence between the event trace and the RTL waveform, accompanied with the asynchronous event-driven nature in hardware simulation, we motivate an asynchronous event-driven programming abstraction to bridge the architectural simulation and implementation. Before introducing this programming abstraction in the next section, we first formalize a generalized architecture pipeline model and relate this model to event-driven programming.

Asynchronous Event-handling & Pipeline Stages: An asynchronous event is executed when its corresponding entry in a bookkeeping FIFO is scheduled. Pipeline stage activation can be abstracted as asynchronous event handling in a similar mechanism: When a pipeline stage is called, its bookkeeping state machine is *subscribed*. This *subscription* remains until the stage is activated, at which point it is cleared. Moreover, the boundary between synchronous and asynchronous execution naturally defines the scope of timing.

Cycle-bound Timing: In Assassyn, each pipeline stage naturally defines a cycle-bound scope. Within each pipeline stage, everything is done in the "current cycle", and the asynchronously invoked stage will be executed no earlier than the "next cycle". For example, Figure 2(d) shows that each pipeline stage is executed in a cycle-bound manner: each pipeline stage is finished within the current cycle, and the enqueued new events will not be executed until next cycle. Moreover, the transposed event-waveform correspondence revealed in this figure also implies a perfect cycle alignment between simulation-based modeling and RTL simulation.

Dataflow across Stages: In architecture design, data flows from one stage to another, which is analogous to a function call: a stage (callee) takes inputs from upstream (caller) and produces outputs for downstream stages (recursively invoke another callee). However, a key distinction between software programming and hardware design is that data for callee in software are typically all from a single caller. In contrast, a pipeline stage can accept data from multiple upstream stages. This multi-source data flow can be abstracted as function bind (see Listing 1), which is is also known as `functools.partial` in Python. This language feature fixes a subset of the function arguments, and creates a new function with fewer arguments.

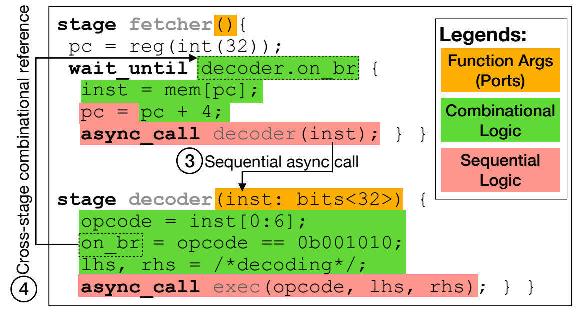


Figure 4: Pipeline stages, IF, and ID, programmed in Assassyn.

```
from functools import partial
def foo(x, y): return x + y
goo = partial(foo, 5) # Fix foo.x = 5
goo(10) # Equivalent to foo(5, 10)
```

Listing 1: A function bind example.

Note, this `bind` approach is **not** a syntactical sugar. This abstraction significantly improves the expressiveness. Refer to Figure 5(b) and explanations in Section 3.7 for more details on how this is used for dataflow abstraction.

Generalized Architectural Pipeline: As it is shown in Figure 2(a.1), an architectural pipeline can be generalized as follows: reading data from the upstream stage buffers, processing them through the intra-stage combinational logic, and then pushing the results to downstream stage buffers. This generalized flow captures the essence of a pipelined architecture. After decades of research and evolution, we believe that the communication protocols and signal-handling mechanisms between pipeline stages have converged toward several mature and optimal design patterns. In our design, we adopt a simple FIFO structure for stage buffers/stage registers, chosen for its balance between generality, efficiency, and moderate on-chip area requirements. For more details on the microarchitectural support and automatic code generation, please refer to Section 5.

All the technical aspects of Assassyn, as well as the synergies among them, are overviewed in Figure 3. In the following three sections, we will in detail explain each of them.

3 Assassyn: Abstraction & Syntax

In this section, we will in detail explain Assassyn's abstraction to describe a pipelined architecture in a *high-level, and general-purpose* programming interface for both cycle-accurate simulation, and RTL generation by sticking to the examples shown in Figure 4 and 5.

3.1 Program pipeline stages in functions

Functions serve as the most fundamental building block of Assassyn to construct each stage of a pipelined architecture. Each function consists of a unique identifier to reference this function itself, an argument list for the stage inputs, and the combinational and sequential logic within its body. Only arithmetic operations, and conditional statements (i.e. if&select) are supported in the function bodies. No loops are supported in function bodies. This restriction ensures that there are no cyclic dependences among the combinational logic within each function to maintain the necessary acyclic nature for proper hardware synthesis. Each function corresponds to dedicated on-chip recourses when synthesized, establishing a direct mapping from our high-level language constructs to physical hardware components.

3.2 Separate combinational & sequential logic

The scope of a function naturally establishes a clear boundary between combinational and sequential logic: all arithmetic computations performed synchronously within this function are considered combinational, which can be completed within one cycle, while any side-effect operations (such as register updates, and memory write) are sequential. These side-effect operations will take effect in the next cycle, which reflects the inherent timing behavior. As shown in ① of Figure 4 activating a downstream stage is sequential, because it involves stage register, state machine writings. The μ-architectural support for the stage registers and state machines will be in detail explained in Section 5.2.

3.3 Invoke stages asynchronously

As discussed in Section 2.2, asynchronous function calls naturally capture the hardware behavior of stage activation — each stage operates on the output of the previous stage in the subsequent clock cycle. As it is shown in Figure 4, the `fetcher` activates the `decoder` stage by calling `decoder` asynchronously, and `decoder` in turn calls `executor` with the decoded results. This chain of asynchronous calls models the flow of data through the pipeline stages, with each stage processing its input in one clock cycle and passing the results to the next stage to be processed in the following cycle. As overviewed in ① of Figure 3, Section 4.3 will elaborate on how the compiler lowers asynchronous function calls for code generation.

3.4 Reference cross-stage logic

In contrast to conventional RTL syntax, where exposing a value to an external module requires verbose module instantiation and explicit pin connections, our language supports a straightforward cross-stage value reference by directly accessing a variable within another function body. As shown in○ of Figure 4, a combinational logic `decoder.on_br` is directly accessed to immediately determine either to fetch a new instruction or stall. The compiler will automatically determine the type of logic of the cross-stage referenced value. See Section 3.7 for an example of cross-stage sequential reference. As overviewed in ② of Figure 3, Section 4.1 provides more details on how the compiler enforces acyclic dependencies among combinational logic.

3.5 Wait until the spin is unlocked

To provide fine-grained control over the execution of invoked functions, we introduce the `wait_until` statement. This statement primitive allows a callee function to postpone its execution until a specified condition is satisfied. When a caller asynchronously invokes a callee, the callee maintains a bookkeeping mechanism to track incoming asynchronous calls. If there is at least one pending call, the callee function continuously checks its wait-until condition to determine whether it should execute, which works like a *spin lock*. If the condition evaluates to true, the function executes and clears the bookkeeping; if not, it defers execution, retaining the count of pending calls until the condition becomes true. All the unused data will be buffered in stage registers, and multiple invocations will be managed by compiler-generated arbiter. Refer to Section 5.2 and 4 for more details. For example, in Figure 4, the fetcher stage must wait to fetch new instructions when the decoder detects a branch instruction, preventing the pipeline from fetching incorrect instructions and ensuring correct control flow.

Takeaway: These basic primitives discussed above already enable productive pipeline construction. Next, we will introduce several advanced language features to further improve the expressiveness, code reusability, and design modularity.

3.6 High-order function for duplication

In RTL programming, it is normal to duplicate code for similar modules, typically achieved through parameterized hierarchical synthesis. Since we already use functions to program pipeline stages, it is intuitive to employ *higher-order functions* to parameterize functions. *Higher-order functions* are functions that return parameterized functions by giving different arguments. Consider the Python example below. `foo` will return a two-argument function, which sums up these two arguments and adds another constant `delta`, and this constant `delta` can be parameterized.

```
def foo(delta):
  return lambda x, y: x + y + delta
goo = foo(5); goo(1, 2) # 1 + 2 + 5
```

Listing 2: An higher-order function example.

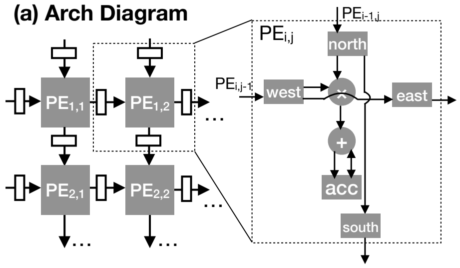
Similarly, in Assassyn, an additional argument list is introduced to instantiate functions with the given argument list which defines the signature of the instantiated function. As shown in Figure 5(b), each systolic processing element is constructed from top-left (1,1) to lower-right (n,n) by supplying its neighbor PEs (south, and east) as parameters. To fully explain this example, the semantics of the `bind` keyword will be discussed in the next section.

3.7 Bind the dataflow

As mentioned, unlike software function calls where all arguments for a callee are provided by a single caller, a pipeline stage often receives data flowed from multiple upstream stages. For example, in the systolic array shown in Figure 5(a), each (PE) receives inputs recursively from its northern and western neighbors–no single PE can asynchronously invoke another with all the necessary data. To address this challenge, we introduce function `bind`, similar to the `functools.partial` in discussed above in Listing 1, which fixes certain arguments of a function in advance, effectively managing partial dataflow from multiple sources.

Figure 5(b) shows how `bind` can be applied to asynchronous function calls, when constructing the entire systolic array. For $\mathrm{PE}_{i,j}$, its southern neighbor's, $\mathrm{PE}_{i+1,j}$, northern input is bound (Figure 5(b), line 8). As illustrated in Figure 5(c), recursively, the eastern $\mathrm{PE}_{i,j+1}$'s northern input is already bound on iteration $i-1$, j. Using the cross-stage access feature discussed in Section 3.4, we can access this bound handle and asynchronously call it (Figure 5(b) line 7) to activate the downstream stage. This approach not only streamlines

(a) Arch Diagram



(b) Higher-order Stage Function

Legend: Higher-order Arguments

```
[#static_timing]
stage PE
   (east:PE,south:bind)
   (west:int<32>,north:int<32>) {
 acc = reg(int(32));
 delta = west * north;
 acc = acc + delta;
 async_call east(west=west);
 bound = bind south(north=north); }
```

(c) Top Function Instantiation

```
1 # Assuming 1-based array, and we
2 # have sentinels for row5 & col5
3 sa = [PE() array of 5x5];
4 # Recursively connect each PE
5 for i in 1..4: for j in 1..4:
6  sa[i][j] = PE.build(sa[i+1][j],
7                      sa[i][j+1].bound)
8  sa[I][j].fifo_depth(west=1,north=1)
```

Figure 5: A systolic array example for hierarchical synthesis, and function bind.

the construction of the systolic array but also simplifies the management of complex *dataflows* among PEs. For more details on the `.build` method, refer to the next section.

3.8 Drive the testbench

To support a testbench function, we reserve the `driver` identifier. When present, the pipeline stage corresponding to this function will be unconditionally activated every cycle, so that this stage can serve as the testbench to generate the signal that *drive*s the execution of the entire pipelined architecture.

3.9 Set the FIFO depth

As discussed in Section 2, to keep the generality of the language, we use FIFOs as our stage registers. Currently, we leave the responsibility of tuning the sizes of the stage FIFOs to the developers through the `fifo_depth` API, as demonstrated in the line 8 of Figure 5(c).

3.10 Syntactical Sugar

Decoupled declaration and implementation: To further explain the systolic array example, we decouple the declaration and implementation of each PE. In complex designs, cyclic dependences among stages may prevent developers from finding a linear order to instantiate each stage. By decoupling declaration and implementation, we can first declare the functions without immediately defining their body logic. Since the function signatures are already determined, only the signatures are needed when constructing the top function. We can instantiate all the functions without concerning for the order of their implementation. Line 3 of Figure 5(c) declares an array of `PE` objects. The internal behavior of each PE is determined by calling the method `.build`, supplying parameters for the higher-order function argument list.

Takeaway: Decoupled declaration and implementation simplifies the management of complex interdependences among the stages.

```
struct Entry {
 valid: bits(1), payload: bits(32) }
# a, b are identical
a, b = reg(Entry), reg(bits(32))
# equivalent to a[0:0] ? a[1:33] : 0
c = a.valid ? a.payload : 0;
# use b like a
d = entry.create_view(a)
e = b.valid ? b.payload : 0;
```

Figure 6: An example for the "struct" syntactical sugar.

Struct slicing: In hardware design, it is normal to pack multiple fields into a bit vector. To simplify access to these packed fields, we introduce a syntactic sugar that allows developers to reinterpret a bit vector as a struct and access its fields through implicit slicing. Figure 6 shows the usage of this syntactical sugar, this struct type can be used to directly declare a data array or create a "view" of an existing bit vector, enabling convenient field-wise access without manual bit manipulation.

4 Compiler

The programmed hardware design undergoes an elaboration process to generate an intermediate representation (IR), a data structure that captures all the aspects of the design. This IR is fed to our compiler to first **enforce** the hardware synthesis constraints. If all the constraints are satisfied, our compiler rewrites the IR for some hardware-specific **transformations**. Finally, our compiler **lowers** the IR to a format ready for code generation.

In the rest of this section, we will discuss these three phases: analysis, transformation, and lowering, respectively.

4.1 Analysis

As mentioned above in Section 3.1 and Section 3.4, cyclic dependences among the combinational logic are prohibited. Since loops are not supported, cyclic combinational logic within a single stage is inherently prevented. Therefore, the compiler focuses on detecting cyclic dependencies in the inter-stage combinational logic.

The compiler inspects all the cross-stage references. For each instance where a combinational expression in one stage references a combinational expression from another stage, the compiler adds a dependency edge from the referencing stage to the referenced stage to construct a dependency graph. For example, the arithmetic operations in Figure 4 and Figure 5 are considered combinational, and corresponds to an edge in the graph. However, the `async_call` and `bind` expressions are considered sequential, which will **not** be added to the dependency graph.

Once the graph construction is done, the compiler performs a topological sort: iteratively, it selects and removes a vertex (stage) with no incoming edges from the graph, along with all its outgoing edges, until the graph is empty. If at a certain iteration no such vertex can be found, it indicates that cyclic combinational dependences exist among the stages, and the compiler reports an error.

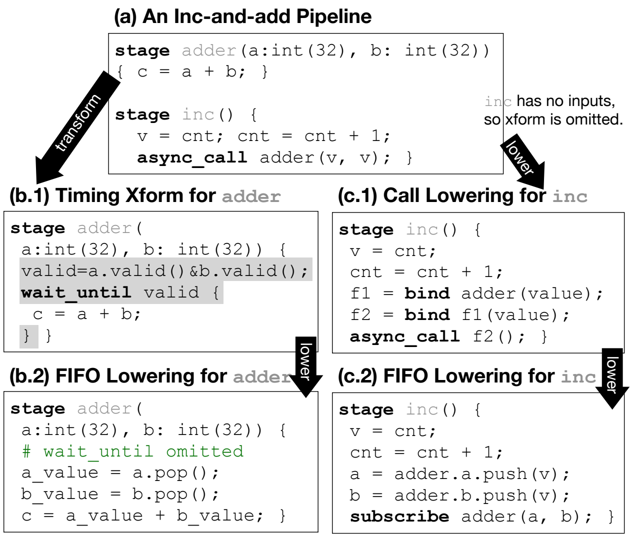


Figure 7: An example for compiler timing transformation and FIFO lowering.

Once the topological sort is done, this order is preserved for further simulator generation (see Section 5 for more details).

4.2 Transformation

We developed two specialized transformations for Assassyn that are especially useful to abstract away the low-level hardware implementation details.

Timing Control: Unlike software programming, where the data are all immediately available, hardware requires developers to manually manage the timing of data arrival. To hide this low-level detail, our compiler will by default wrap all the function bodies with a `wait_until` statement. As shown in Figure 7(b.1), the `wait_until` statement checks if all the operands in the stage buffer are valid. If any operand is invalid, this `adder` event cannot proceed. Developers can also use a `#static_timing` tag on stages to disable the transformation, as shown in Figure 5(b).

Arbiter Generation: There are two key constraints that differentiate hardware design from software programming:

1. **Resource allocation:** Each function/stage can only be called/activated once in each cycle, because dedicated on-chip resources are allocated to each pipeline stage;
2. **Register write:** A register can only be written once in each cycle, and the written value will not take effect until the very end of the current cycle.

Therefore, to evade these issues, the compiler will automatically detect the functions with multiple callers and generate an arbiter for them. As it is shown in Figure 8, both the `EX` and `MA` activate the `WB` stage to commit the results. However, these activations are not guaranteed to be mutually exclusive within a single cycle. If they both activate stage `WB` in a same cycle, they will not only activate `WB` twice, but also write to `WB`'s stage buffer twice, which leads to an undefined behavior. Therefore, an arbiter is generated among these three stages so that two upstream stages write data to separate sets of stage registers, and the state machine determines which value to commit. We currently support `#round_robin` and `#priority_arbiter` tags, allowing developers to decide strategies.

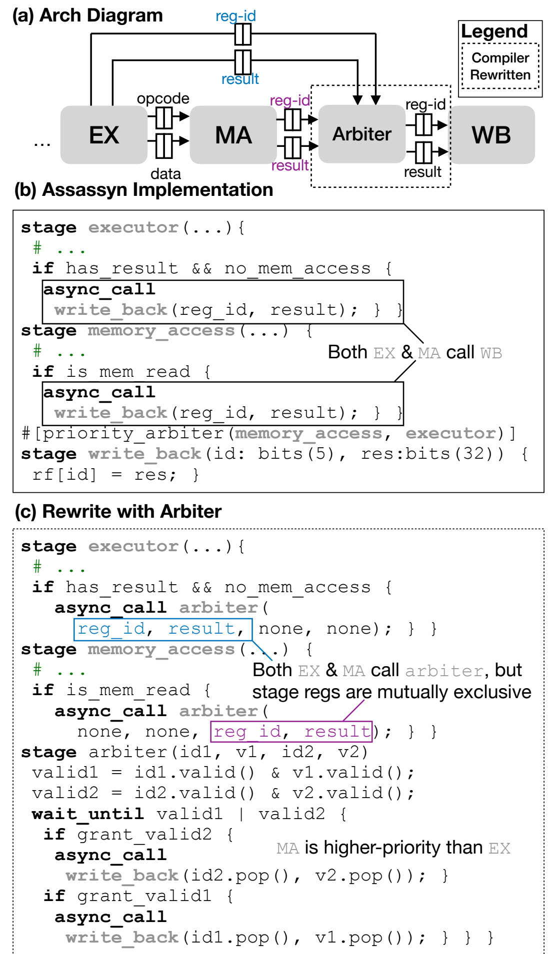


Figure 8: An example for compiler-generated arbiter.

In this case, in an in-order CPU, `MA` executes an earlier instruction, so it has a higher priority than `EX`.

4.3 Lowering

Recall that, as discussed in Section 2.2, FIFOs are adopted as our stage buffers. Therefore, we need to explicitly represent FIFO push and pop in our IR for code generation.

Function Binds to Pushes. To maintain a unified interface for compiler implementation, our compiler first rewrites all the multi-argument function calls and binds to single operand binds, and then replaces the function call itself with the bound handles. As lowering rewriting shown from Figure 7(a) to to Figure 7(c.1), the `async_call` statement is later replaced by two `bind`s, `f1` and `f2`, and then call the fully bound handle, `f2`. After this step, all the function binds will be replaced by FIFO pushes, and the function call will be replaced by event `subscription`s, as shown in Figure 7(c.2).

Function Pops. When activating a function or stage, the values in the FIFO buffers should be popped so that the next set of inputs is available at the head of the FIFOs for the next cycle. As shown in Figure 7(b.2), for the functions that implicitly use all their operands, our compiler will inject FIFO pop at the beginning of the `wait_until` body. The FIFO pop statements do not necessarily pop all the input arguments of a function. For example, as shown in

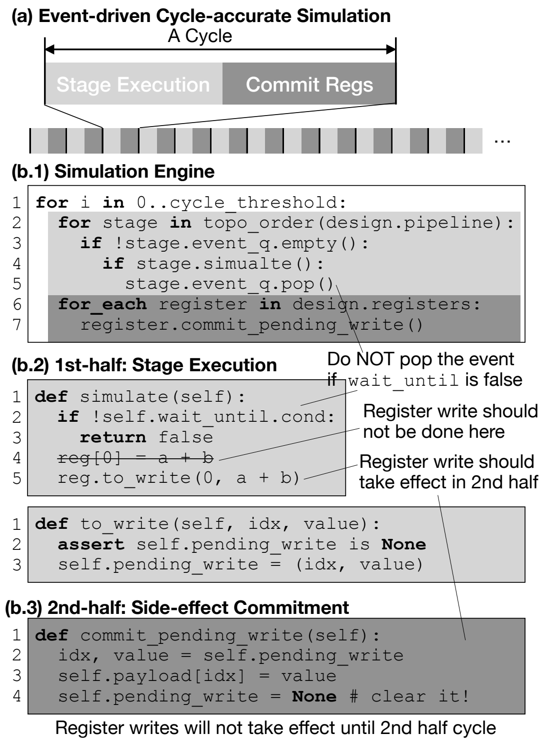


Figure 9: Assassyn-generated cycle-accurate simulator

8(c), when the generated arbiter grants execution, only the subset of the involved operands are popped.

5 Code Generation

After transforming and lowering the IR, the final step is to generate code for both the simulator and the RTL implementation. Translating the logic within each stage is straightforward, as our high-level functional programming interface naturally maps to the intended program behavior. Therefore, this section primarily focuses on the runtime support for hardware simulation, and the microarchitectural support to connect pipeline stages and generate the RTL implementation.

5.1 Simulator Generation

Figure 9(a) overviews the structure of an Assassyn-generated cycle-accurate simulator: A simulator engine drives the stage executions as well as register commitments that occur in each cycle.

Cycle-Accurate Event-Driven Simulation: As illustrated in Figure 9(b.1), in each cycle, the simulation engine traverses all the pipeline stages of the hardware design in the topological order discussed in Section 4.1. This ensures that all the cross-stage combinational references access well-defined values.

For every cycle, the simulation is divided into two phases: *stage execution* for simulating the behaviors of each stage, and *register commitment* to update the values of the registers.

Stage Execution: The first phase simulates the combinational logic within each stage. Figure 9(b.1) shows that the simulation engine traverses the `event_q` of each stage, and invokes the behavioral simulation function. The return value of each simulation function is determined by the `wait_until` condition. If true, this event will be cleared, and vice versa.

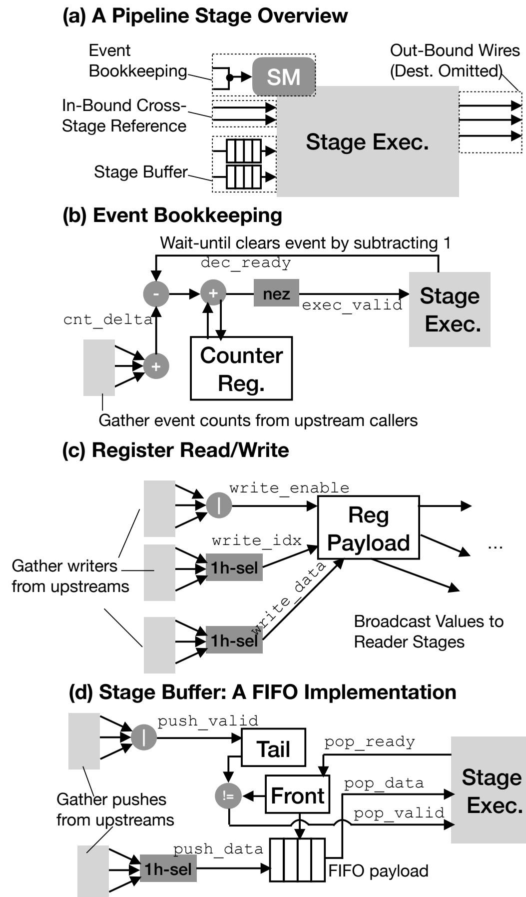


Figure 10: The μ-architectural support for RTL generation.

Register Commitment: As it is shown in Figure 9(b.2) line 4, during the stage execution phase, values written to registers cannot take effects immediately. Instead, it writes to each register's `pending_write` bookkeeping by invoking the `to_write` runtime API. This `to_write` API also enforces that each register can only be written once in each cycle. If a register is written more than once, an error will be thrown to terminate the simulation, indicating a mistake in the architectural design. All these pending writes will be committed to the registers in the second phase of a cycle, and be cleared for the subsequent cycles' simulation.

Randomization: Ideally, the order of executing stages in each cycle should not affect the simulation result. On the other hand, serialized event simulation cannot capture the full details of hardware's concurrency. Therefore, we provide a runtime flag to optionally shuffle the order of the stages without breaking the topological dependences to emulate the non-determinism.

5.2 RTL Generation

Generating RTL implementations essentially maps each aspect of the high-level abstraction to highly efficient micro-architectural components, as overviewed in Figure 10(a).

Combinational Logic: As discussed in Section 3.1, each stage's function body is composed of acyclic arithmetic operations, so it is simple to map each operation to `wire`d combinational logic in RTL.

Table 1: Manual Designs

Target Design	Reference
Priority Q [12]	Manual Impl.
In-order CPU	Sodor [3]
Out-of-order CPU	
Systolic Array	Gemmini [28]

Table 2: HLS-generated Designs from MachSuite [58]

Application	Data Size
ellpack	n=494,m=10
stencil-2d	img=128^2,f=3^2
radix-sort	n=2048,m=16
kmp	n=32000,m=4
merge-sort	n=2048

All the cross-stage combinational references can be mapped to pin ports that connect input/output values among `modules`.

Event Bookkeeping & Wait-until: We use a counter-based state machine associated to each pipeline stage to enable the conditional stage activation and event clearance. Specifically, as shown in Figure 10(b), each stage's wait-until condition is wired into this state machine to *decrease* the counter, and all the signals from upstream callers to this stage will be gathered by *addition*, rather than *or*, to ensure no event is missed and increase the counter.

Register Read/Write: The value of register writes will not take effect until the next cycle, so we use non-block assignment (<=) to write the registers. Moreover, as discussed above, a register can be read any number of times in each cycle, while only one write is allowed. Therefore, we use an or-operation to gather all the writers' write-enable signal, and conduct a one-hot selection to determine which value is written to the register. This similar technique is also applied to FIFO pushes as shown in Figure 10(d).

FIFO as Stage Registers: As discussed in Section 2.2, to ensure the generality of our language, we adopt FIFOs as our stage register. These FIFOs can be parameterized by data types determined by the data enqueued and buffer sizes determined by the `fifo_depth` API mentioned above, allowing developers to adjust the data arrival timing, consumption rate, and buffering. We implemented a penetrable FIFO as our template design, and this FIFO will fall back to a stage register when `fifo_depth(1)` is given.

See **Q4** in Section 7 for more details on the area overhead of these components discussed above.

6 Evaluation Methodology

Implementation: The current Assassyn frontend is embedded in Python by overloading the operators. By tracing the Python program execution, the hardware intermediate representation is recorded in an abstract syntax tree (AST). Then this AST is fed to our backend (implemented in Rust) for compiler transformations and code generation for both simulators and RTL implementations. The generated cycle-accurate simulators are in Rust, and the generated RTL implementations are in SystemVerilog.

Designs:[3] We select 8 representative reference designs to stress Assassyn. 3 of them are handcrafted RTL implementations, 5 of them are HLS-generated from MachSuite [58] as shown in Tables 1 and 2. We implemented a priority queue in SystemVerilog; the reference design of CPU and systolic array are from the latest Chipyard GitHub release [59]; all the HLS workloads are from Bambu [55]'s.

Software Platform: All the Rust codes, both the Assassyn compiler backend, and the Assassyn-generated simulators, are compiled by Rust 1.81.0. To simulate the Verilog, we use Verilator `v5.027` to compile the SystemVerilog, and the generated C++ simulators are compiled by GNU GCC-11.4.0.

[3]All the Assassyn-related infrastructures are are available at https://github.com/Synthesys-Lab/assassyn

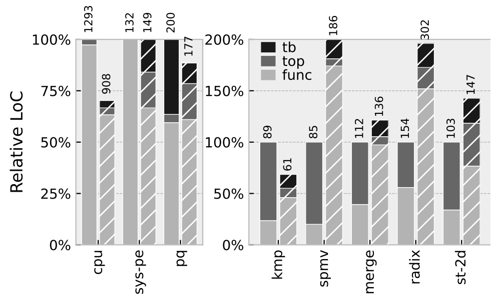


Figure 11: Lines of code breakdowns compared to reference designs, with absolute LoC above. Unhatched is the reference design, and hatched is Assassyn. (Q2)

We use Bambu [55] as our HLS baseline. All the HLS designs are generated by their `bambu-v0.9.7.AppImage` release, and simulated by Verilator. To make a fair comparison, we assume these HLS-generated designs can make fully pipelined exclusive scalar memory read/write with one-cycle latency.

To estimate the chip area of the given RTL implementation, including manual, HLS-generated, and Assassyn-generated, we use Yosys [74], accompanied with ASAP7 [22] technology library, to synthesize the RTL implementations, with all the memory-related modules excluded by a `(*blackbox*)` directive.

All the lines of code (LoC) are counted by feeding related files to `cloc` [2]. Then we manually characterize the semantics and functionality of each part of the code into 3 categories: module, top, and testbenches.

Hardware Platform: All the simulator performance reported below are single-thread, running on an AMD EPYC-7763 CPU.

Takeaway: All the chosen third-party tools and libraries are open-source to maximize our reproducibility. We plan to submit for artifact evaluation release our open-sourced infrastructure upon acceptance.

7 Evaluation

We evaluate three main aspects of Assassyn: the expressiveness of its abstraction, the quality of the generated RTL implementation, and the fidelity and performance of the generated simulator. The key results are:

- Assassyn's abstraction is expressive enough to program a wide range of hardware designs, from component modules, to application-specific accelerators and an end-to-end CPU.
- The generated simulators perfectly align with the RTL simulation results, while achieving 2.2-8.1× speedup.
- The generated RTL achieves near-handcrafted quality in terms of area, while requiring only 70% of lines of codes.

Next, we will in detail explain these key results by comparing Assassyn with both hand-crafted RTL, and HLS-generated designs on several representative workloads.

Q1. How expressive is Assassyn's abstraction?

Tables 1 and 2 show the diverse target designs we stress Assassyn to compare with both handcrafted, and HLS-generated designs. Each design showcases unique design and implementation challenges to highlight the effectiveness of Assassyn's abstraction.

For example, CPU is nearly a linear pipeline with complex inter-stage controls. Sequential communications between stages are expressed in asynchronous function invocations, and the inter-stage controls are in combinational cross-stage references.

Systolic array, exemplifying a dataflow architecture, presents a unique challenge in that each PE gathers data from multiple sources. Our *function bind* abstraction effectively enables expressing this architectural paradigm.

Notably, when manually mapping imperative C code to Assassyn for HLS comparison, we observe an interesting pattern: each Assassyn function resembles a basic block in a control flow graph, while `async_call` acts as a branching mechanism between these blocks. This correspondence not only highlights how our abstraction bridges the imperative programming model with hardware design and implementation, but also unveils the potential of leveraging Assassyn as a novel HLS backend.

Q2. How well does Assassyn mitigate the difficulties of hardware design and implementation?

Figure 11 presents a comparison of lines of code (LoC) for each workload between designs implemented with Assassyn and their corresponding reference implementations, which are either handcrafted RTL or HLS-generated code.

Handcrafted RTL: Assassyn requires only 70% of the LoC of the reference RTL when the design is as complicated as a single-issue CPU; for simple components, the LoC is comparable to Chisel RTL. We excluded all the highly overengineered common modules in Chipyard-related reference designs for Sodor CPU, and Gemmini systolic array, or the LoC comparison will be badly skewed. For example, Chipyard overengineered a unified testbench for all the CPU-based designs in the `generator` directory, and Sodor adopts a comprehensive implementation of state control registers (CSR) from `roketchip`. These LoC savings are primarily attributed to the language abstraction provided by Assassyn. Traditional RTL implementations involve redundant code for pin declarations and connections across hierarchical synthesis, whereas Assassyn simplifies these aspects through function calls and cross-module references.

HLS-generated Design: Our Assassyn-programmed workloads require, on average, only 1.26× the LoC of the MachSuite C code, with testbench `harness` included, to implement these application-specific accelerators. Two outliers, `spmv` and `merge` require more than 2× LoC in total. `spmv`'s kernel alone even demands 11× LoC. `spmv`'s kernel complication stems from three memory operations, 2 loads and 1 write, in its loop body. The exclusive memory read-/write constraints necessitates careful state machine management to schedule the memory accesses. `merge` suffers from a similar situation. Nevertheless, Assassyn's full control over the underlying hardware's micro-architecture and pipeline stages enable better performance and area tuning. As a result, as shown in Figure 12, we achieve an order of magnitude improvement in area-normalized performance over HLS.

Q3. What is the quality of Assassyn-implemented hardware?

As shown in Figure 12, Assassyn-programmed designs achieve comparable performance per area to handcrafted designs, while delivering up to 32× and by mean 6× over HLS-generated designs. Below, we analyze these results by examining both performance and area characteristics.

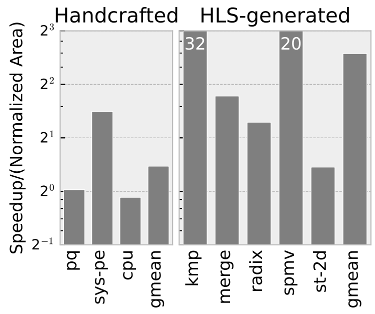


Figure 12: Area normalized performance (Q3)

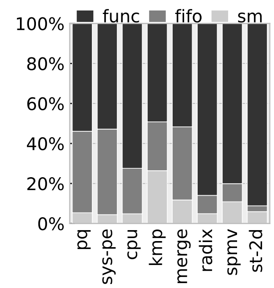


Figure 13: Area breakdown (Q4)

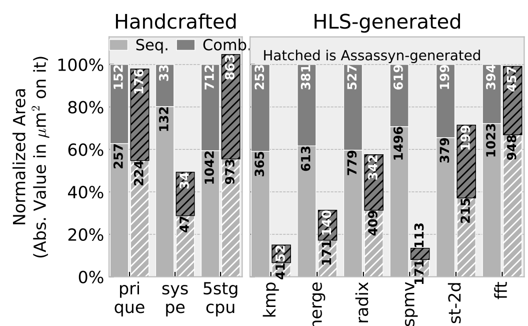


Figure 14: Area compared with reference designs (Q3)

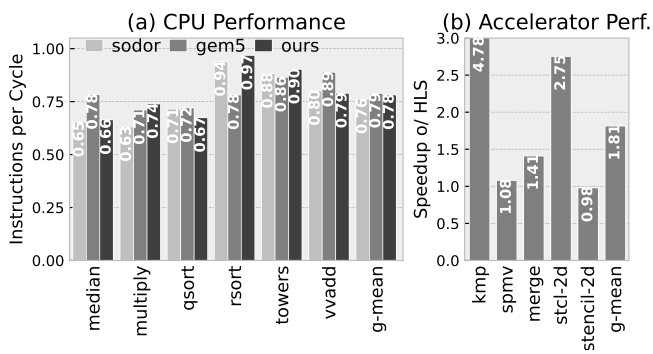


Figure 15: CPU/HLS performance comparison (Q3)

Performance: The performance of the priority queue and the systolic array are estimated by their pipeline initial interval (II). Because handcrafted references and the Assassyn-programmed ones achieve the identical desired II, we assume they have the same performances. Therefore, we only simulated the Sodor CPU to evaluate the performance, by running six workloads from the Sodor project folder. Our implementation achieves slightly better performance, 2.6% higher IPC, than Sodor CPU as shown in Figure 15(a), because we implemented an always-take branch prediction, which introduces around 3% area overhead as shown in Figure 14.

As shown in Figure 15(b), when comparing with equal provisioned HLS code, Assassyn-programmed designs achieve, on average, 1.8× speedup over HLS-generated designs. The speedup primarily stems from two factors: 1) our full control over the underlying pipeline stages and micro-architecture, allowing for more aggressive pipeline scheduling compared to HLS, and 2) smarter, human-driven optimizations. In `kmp`, the length of the pattern string is only 4, so we implemented a brute force string match to avoid excessive memory access, and complicated control. In `radix_sort`, instead of allocating the radix brackets in SRAM, we use 16 registers as our radix brackets. This design decision trades off chip area and the number of iterations required to scan the entire array, while also eliminating two memory accesses for bracket accumulation. This simplification reduces the complexity of the state machine that manages exclusive memory reads and writes in the innermost loop body. In `merge_sort`, our implementation adopts an infinite sentinel to maintain a unified interface of popping merged elements when one side is exhausted, which simplifies the state transition.

Area Cost: The area comparison is shown in Figure 14. Our Assassyn-implemented designs achieve comparable area to handcrafted designs with similar functionality (CPU and priority queue). Systolic PE is an outlier, because Gemmini's [28] systolic array is highly flexible for both output and weight stationary models, which introduces extra combinational area to control the state machine inside the PE. On the other hand, HLS inevitably suffer from area overhead caused by HLS tools, and our Assassyn-programmed designs achieve an average area savings of 70%.

Q4. What is the language abstraction overhead of Assassyn?

To better understand our language abstraction overhead, we characterize the area of two main Assassyn-generated architectural components, the FIFOs serving as stage registers, and the counter stage machine serving as stage execution bookkeeping. The area breakdown shown in Figure 10 are counted by synthesizing each standalone component, and sum them up, because the synthesis tool flattens all the architectural hierarchies for more aggressive synthesis optimizations, which loses the hierarchical information for components. This approach may slightly enlarge the area of each component. Their area breakdowns are shown in Figure 13. For `spmv`, and `stencil-2d`, the functionality area is mainly occupied by the multiplication and accumulation unit, while for `radix`, the functionality area is mainly spent on the bracket registers.

The stage registers are inevitable when designing and implementing pipelined architectures. Their areas can be easily tuned by calling the `fifo_depth` API. When area 1-depth is given, the FIFO will fall back to a single stage register. These stage buffers occupy around 20%-40% of the area in control-heavy designs, e.g. CPU, priority queue, and merge sort.

In most of the workloads, another language-generated component, the counter state machine is a modest overhead (consumes less than 5% of the total area). An outlier, `kmp`, is too simple to dwarf this component — as shown in Figure 14, the absolute area is less than $100\mu m^2$. Though redundant, in many static-timing designs, e.g. systolic array, such state machines are still generated to maintain a unified interface for both simulation and RTL generation, which can be eliminated in our future version.

Q5. What is the quality of the generated simulator?

We evaluate three key aspects of our Assassyn-generated simulators, the alignment, the performance, and debugging.

Alignment: As discussed in Section 2.2, the transposed correspondence between the event simulation traces and the waveform activation enables a perfect alignment between the simulation-based modeling and the RTL implementation. Across all our target designs, all the cycles counts from Verilator simulated RTL exactly match our Assassyn-generated Rust simulator. In contrast, aligning

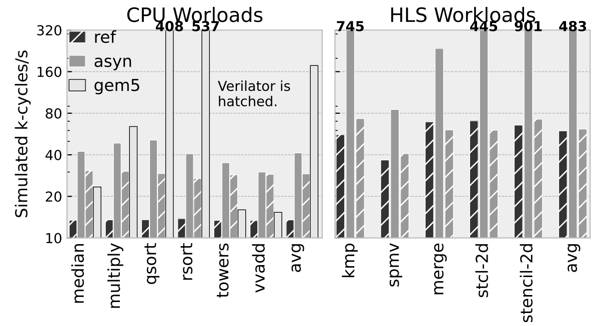


Figure 16: Assassyn-generated cycle-accurate simulator compared to Verilator-generated Verilog simulation, and simulation-based modeling (Q5)

gem5-simulated results with an actual RTL implementation proves challenging even in a design as simple as a single-issue CPU.

To demonstrate, we configured a minimized in-order, single-issue, and one-cycle memory access CPU in gem5 23.0. As shown in Figure 15, while the three implementations — sodor, gem5-simulated, and ours — achieve similar mean performance, gem5's results show significant fluctuations across different benchmarks. This suggests that the similar mean performance is merely coincidental, resulting from offsetting variations rather than consistent behavioral alignment with RTL. Our detailed analysis of execution traces revealed specific sources of misalignment. In `median` and `vvadd`, gem5 CPU outperforms ours because its fetch stage can access branch execution results within the same cycle — an design that would lengthen the combinational critical path in actual hardware. Conversely, gem5 underperforms on `rsort` due to a missed bypassing opportunity: when instruction A is decoded and depends on instruction B in writeback, B's result remains invisible to A until next cycle while bypass registers already have been flushed and occupied by other newer instructions in EX and MA. Such subtle discrepancies can hardly be discovered through extensive coordination between design and implementation teams, while it is naturally aligned in our framework.

Performance: As shown in Figure 16, our Assassyn-generated cycle-accurate simulator implemented in Rust is 2.2× faster than the Verilator-generated simulator on CPU simulation, and has 8.1× speedup over HLS ASIC simulation. This speedup comes from the domain knowledge of simulating pipeline stages, which significantly simplifies the architectural simulation model. A generic SystemVerilog simulation/modeling typically involves a rather complicated process to 1. maintain the event queue of each timescale; 2. determine the active and inactive code regions in a fine-grained style; 3. compute the logics; 4. cleanup each cycle and move time forward [4]. Our two-phase model discussed in Section 5.1 can rapidly determine the active and inactive code region in the granularity of each pipeline stage to save simulation time.

For workloads fewer than 10k cycles like `vvadd`, `tower`, and `median`, gem5's initialization overhead hinders its performance compared to both Assassyn and even Verilator-generated simulator. However, for longer-running workloads like `qsort` and `rsort`, gem5 achieves an order of magnitude speedup once this overhead is amortized. Gem5 excels in raw simulation speed, but our Assassyn-generated simulators offer a unique cycle-exact correspondence

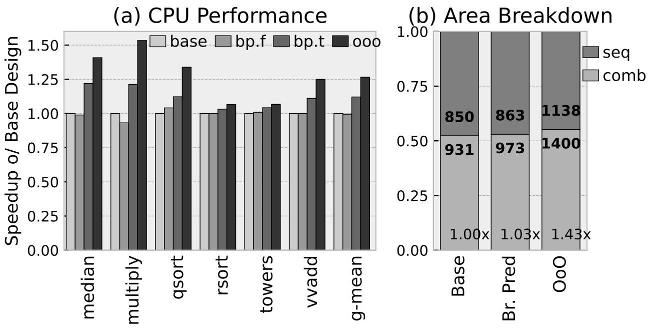


Figure 17: CPU performance by incrementally enabling branch prediction, and out-of-order. `bp.f` is always not taken, and `bp.t` is always taken.

with RTL implementation, which will enable precise debugging and facilitate seamless transition from design to implementation.

Debugging: Conventional Verilog simulation typically involves massive concurrency and non-determinism, making it hard to locate the mistake, while the serialized event-driven simulation, with operations within the same stage tightly coupled, significantly simplifies tracing execution and analyzing expected behaviors. Most behavioral bugs can be easily found at this phase.

Q6. How could Assassyn facilitate a seamless architecture design and implementation?

A key benefit of Assassyn is enabling seamless transition from architectural design to RTL implementation. Design decisions can be rapidly evaluated for both performance impact and hardware cost. We demonstrate this advantage through an progressive CPU design case study.

We incrementally developed several CPU variants. Starting from a fully interlocked 5-stage single-issue in-order base design, we extended it with branch prediction and ultimately implemented an out-of-order (OoO) version with branch prediction. For the branch prediction study, we implemented two simple strategies: always-taken (bp.t) and always-not-taken (bp.f). The performance impact of each mechanism is shown in Figure 17(a), and the generated RTL can be immediately synthesized to inspect the cost, as shown in Figure 17(b). The always-not-taken predictor shows limited performance improvement because most branches in these workloads are taken, particularly loop branches that jump back to the loop header. This behavior is quantified by the success rate of the always-taken predictor shown below:

median	mul	qsort	rsort	towers	vvadd
59.4%	90.6%	64.9%	76.2%	85.7%	71.8%

Both branch predictors require very near on-chip area, so we only report the always-taken predictor area, which improves the performance by 1.12× and introduces around 3% area overhead.

OoO Execution: Moreover, our framework scales to designs as complicated as an out-of-order CPU with always-taken branch prediction. It achieves 1.26× speedup over the base design, and introduces 1.43× area overhead. To understand this performance gain, we profiled each workload execution. Instructions are dispatched to reservation station in almost every cycle. An outlier is `qsort`, which introduces 2.1% dispatch idle, because of the limited size of the reservation buffer size. Instructions can retire after 3 cycles when they are on the correct code path, and issuance unit only is only idle for 5.4% of the cycles, which is mostly caused by branch misprediction. In more than 99% of misprediction, there is at most one instruction mistakenly dispatched by the always-taken branch prediction, because we prioritize the branch instruction execution on the reservation stations. To sum up, all the profiling above suggests OoO effectively exploits the CPU pipeline utilization.

When developing, Assassyn illustrates a key advantage: Analogous to software application development, which follows a "algorithm+data structure" paradigm, our Assassyn-implemented out-of-order CPU follows a "pipeline logic+bookkeeping" approach. This abstraction naturally separates the core pipeline functionality from the state management required for out-of-order execution, making the code base more maintainable.

8 Discussion

The fundamental challenge in architectural design and implementation stems from the complexity of RTL programming, which led to the separation of simulation and implementation codebases, while Assassyn addresses this challenge by reproposing a programming paradigm to unify the simulation and implementation.

8.1 Related Works

Prior hardware modeling works [14, 15, 17, 43, 80] still remain highly disconnected between the simulation and implementation. The prior work closest to our goal should be PyMTL [46] and Gem5+RTL [47]. PyMTL aims at offering a unified simulation-based modeling framework to integrate components with different level of implementation, from functional to RTL, but this still relies on developers to manually implement of a same component multiple times. gem5+RTL allows developers to integrate their extended components written in RTL to a full-system simulation. In contrast to the framework discussed above, the primary goal of Assassyn is even more aggressive, unifying the cycle-accurate simulation and the RTL implementation.

Meanwhile, high-level RTL generators target only a limited subset of architectures, including but not limited to general-purpose CPUs [17, 77, 78] or domain-specific accelerators [27, 39, 68, 72], or address individual challenges in hardware description — such as placement [67] *or* timing [50]. In contrast, Assassyn seeks an *all-in-one* approach for design and implementation by carefully rethinking abstraction through the lens of software language evolution.

Beyond the scope of this programming paradigm, our work reveals two boarder insights on hardware description language and architectural design compared to prior related works.

Analogous to software programming, we characterize conventional SystemVerilog as assembly code, and Chisel [6] serves as a intrinsic wrapper (or syntactical sugar) that encapsulates many common uses of RTL programming, targeting an open-source IR infrastructure [32], CIRCT (a.k.a. FIRRTL), to mitigate the programming difficulties. These prior works still adhere to the circuit graph abstraction, and pin connections. Meanwhile, software languages evolved beyond assembly by carefully trading-off the unnecessary expressiveness. A famous example is the deprecation of the `goto-statement` in modern languages: by disabling the excessive flexibility of branching across the basic blocks, programmers' productivity and code quality were significantly improved. The compiler can also make stronger assumptions to apply more aggressive optimizations. **Assassyn represents a first step toward a "C-like" abstraction for hardware design and implementation.** Similarly, by carefully constraining the programming model to event-driven patterns, and binding the dataflow across the pipeline stages, a higher level of abstraction that retains the expressiveness to implement many practical architectures is achieved.

Moreover, though many prior works [20, 42, 44, 56, 72, 73] already demonstrated that domain-specific accelerators can be composed by connecting spatial processing elements, our work suggests an even more fundamental principle: **Spatial processing element is all you need** for pipelined architectural construction. Each pipeline stage can be viewed as a spatial processing element, and its functionality can be programmed within an Assassyn function. A pipelined architecture can be viewed as a spatial arrangement of these processing elements connected by dataflows. This principle becomes particularly striking when considering seemingly sequential architectures like CPUs — unlike spatial dataflow accelerators that typically employ an array of homogeneous processing elements, CPU pipelines are constructed by connecting multiple heterogeneous stages, yet still fundamentally adhere to this spatial arrangement paradigm. This principle may lead to a more agile design, implement, and evaluation flow for new architectures.

8.2 Future Works

Frontend: When implementing the `radix_sort` and `merge_sort`, we found it particularly challenging to manually manage the state of execution and transition across the different phases of the algorithm. It will be highly desirable to have better abstraction to program different code regions that share the same inputs but execute under different conditions, and the transitions across these conditions can be easily and clearly describe like imperative programming.

Backend: As a programming language, Assassyn occupies a unique position, which is both *domain-specific* in its focus on hardware design and implementation, and *general-purpose* in its expressiveness of the architectural construction. *"Domain-specific"* typically implies that additional domain knowledge enables more aggressive automated optimizations [8, 18, 38, 57, 71, 82] on the programmed *"general-purpose"* architecture designs. As discussed above, our language naturally encodes: 1. the clear boundary of each pipeline stage, and 2. the clear separation between the combinational and sequential logic. This domain-specific knowledge opens up promising optimization opportunities to automatically 1. find the critical path of a design before synthesis; 2. verify the intra-stage and inter-stage stage machine footprint for complicated designs.

Integration: Assassyn-generated RTL maintains clear correspondence to high-level design intentions, making it more readable than conventional HLS-generated RTL for both human and machines. This quality positions Assassyn as a valuable tool for generating high-quality training datasets for AI for hardware design.

9 Conclusion

We introduced Assassyn, a *unified*, *general-purpose*, and *high-level* programming framework for hardware design and implementation, offering a fresh perspective on describing hardware pipelines. Our evaluation highlights the framework's advantages in terms of expressiveness, productivity, and the quality of hardware generation, demonstrating its potential as a transformative tool in the hardware design space. More broadly, this work establishes a foundation for a new paradigm in hardware description languages, bridging the disconnection between the high-level design and the low-level implementation. In doing so, it opens the door to future research in automated design optimizations and reimagines the way we approach hardware design and development.

Acknowledgments

We thank all the anonymous reviewers' insightful and constructive comments on this work. This work is fully supported by the baseline funding offered by King Abdullah University of Science and Technology (KAUST).

References

[1] [n. d.]. BlueSpec Verilog. https://bluespec.com/

[2] [n. d.]. cloc counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc

[3] [n. d.]. Educational Microarchitectures for RISC-V ISA. https://github.com/ucb-bar/riscv-sodor

[4] 2024. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (2024), 1–1354. doi:10.1109/IEEESTD.2024.10458102

[5] Sam Ainsworth and Lev Mukhanov. 2024. Triangel: A High-Performance, Accurate, Timely On-Chip Temporal Prefetcher. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1202–1216. doi:10.1109/ISCA59077.2024.00090

[6] 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 *49th DAC*.

[7] Daehyeon Baek, Soojin Hwang, and Jaehyuk Huh. 2024. pSyncPIM: Partially Synchronous Execution of Sparse Matrix Operations for All-Bank PIM Architectures. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 354–367. doi:10.1109/ISCA59077.2024.00034

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[9] A Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. 163–174.

[10] Mohammad Bakhshalipour and Phillip B. Gibbons. 2024. Tartan: Microarchitecting a Robotic Processor. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 548–565. doi:10.1109/ISCA59077.2024.00047

[11] Rahul Bera, Adithya Ranganathan, Joydeep Rakshit, Sujit Mahto, Anant V. Nori, Jayesh Gaur, Ataberk Olgun, Konstantinos Kanellopoulos, Mohammad Sadrosadati, Sreenivas Subramoney, and Onur Mutlu. 2024. Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 88–102. doi:10.1109/ISCA59077.2024.00017

[12] R. Bhagwan and B. Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, Vol. 2. 538–547 vol.2. doi:10.1109/INFCOM.2000.832227

[13] Anubhav Bhatla, Navneet, and Biswabandan Panda. 2024. The Maya Cache: A Storage-efficient and Secure Fully-associative Last-level Cache. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 32–44. doi:10.1109/ISCA59077.2024.00013

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).

[15] B. Black and J.P. Shen. 1998. Calibration of microprocessor performance models. *Computer* 31, 5 (1998), 59–65. doi:10.1109/2.675637

[16] Ishita Chaturvedi, Bhargav Reddy Godala, Yucan Wu, Ziyang Xu, Konstantinos Iliakis, Panagiotis-Eleftherios Eleftherakis, Sotirios Xydis, Dimitrios Soudris, Tyler Sorensen, Simone Campanoni, Tor M. Aamodt, and David I. August. 2024. GhOST: a GPU Out-of-Order Scheduling Technique for Stall Reduction. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1–16. doi:10.1109/ISCA59077.2024.00011

[17] Odysseas Chatzopoulos, George-Marios Fragkoulis, George Papadimitriou, and Dimitris Gizopoulos. 2021. Towards Accurate Performance Modeling of RISC-V Designs. arXiv:2106.09991 [cs.AR] https://arxiv.org/abs/2106.09991

[18] 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 OSDI*.

[19] Hao-Wei Chiang, Chin-Fu Nien, Hsiang-Yun Cheng, and Kuei-Po Huang. 2024. ReAIM: A ReRAM-based Adaptive Ising Machine for Solving Combinatorial Optimization Problems. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 58–72. doi:10.1109/ISCA59077.2024.00015

[20] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *28th ASAP*.

[21] Md Hafizul Islam Chowdhuryy, Hao Zheng, and Fan Yao. 2024. MetaLeak: Uncovering Side Channels in Secure Processor Architectures Exploiting Metadata. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 693–707. doi:10.1109/ISCA59077.2024.00056

[22] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. doi:10.1016/j.mejo.2016.04.006

[23] Cansu Demirkiran, Guowei Yang, Darius Bunandar, and Ajay Joshi. 2024. Mirage: An RNS-Based Photonic Accelerator for DNN Training. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 73–87. doi:10.1109/ISCA59077.2024.00016

[24] Aniket Deshmukh, Lingzhe Chester Cai, and Yale N. Patt. 2024. Alternate Path Fetch. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1217–1229. doi:10.1109/ISCA59077.2024.00091

[25] Quang Duong, Akanksha Jain, and Calvin Lin. 2024. A New Formulation of Neural Data Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1173–1187. doi:10.1109/ISCA59077.2024.00088

[26] Yuan Feng, Seonjin Na, Hyesoon Kim, and Hyeran Jeon. 2024. Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 834–847. doi:10.1109/ISCA59077.2024.00065

[27] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. 2012. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media.

[28] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. doi:10.1109/DAC18074.2021.9586216

[29] Nika Mansouri Ghiasi, Mohammad Sadrosadati, Harun Mustafa, Arvid Gollwitzer, Can Firtina, Julien Eudine, Haiyu Mao, Joël Lindegger, Meryem Banu Cavlak, Mohammed Alser, Jisung Park, and Onur Mutlu. 2024. MegIS: High-Performance, Energy-Efficient, and Low-Cost Metagenomic Analysis with In-Storage Processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 660–677. doi:10.1109/ISCA59077.2024.00054

[30] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jiménez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR]

[31] Seunghee Han, Seungjae Moon, Teokkyu Suh, JaeHoon Heo, and Joo-Young Kim. 2024. BLESS: Bandwidth and Locality Enhanced SMEM Seeding Acceleration for DNA Sequencing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 582–596. doi:10.1109/ISCA59077.2024.00049

[32] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. doi:10.1109/ICCAD.2017.8203780

[33] Aamer Jaleel, Gururaj Saileshwar, Stephen W. Keckler, and Moinuddin Qureshi. 2024. PrIDE: Achieving Secure Rowhammer Mitigation with Low-Cost In-DRAM Trackers. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1157–1172. doi:10.1109/ISCA59077.2024.00087

[34] Aditya K Kamath and Simon Peter. 2024. (MC)2: Lazy MemCopy at the Memory Controller. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1112–1128. doi:10.1109/ISCA59077.2024.00084

[35] Nikos Karystinos, Odysseas Chatzopoulos, George-Marios Fragkoulis, George Papadimitriou, Dimitris Gizopoulos, and Sudhanva Gurumurthi. 2024. Harpocrates: Breaking the Silence of CPU Faults through Hardware-in-the-Loop Program Generation. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 516–531. doi:10.1109/ISCA59077.2024.00045

[36] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. doi:10.1109/ISCA45697.2020.00047

[37] Tae Hoon Kim, David Rudo, Kaiyang Zhao, Zirui Neil Zhao, and Dimitrios Skarlatos. 2024. Perspective: A Principled Framework for Pliable and Secure Speculation in Operating Systems. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 739–755. doi:10.1109/ISCA59077.2024.00059

[38] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 77.

[39] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 296–311.

[40] Apostolos Kokolis, Antonis Psistakis, Benjamin Reidys, Jian Huang, and Josep Torrellas. 2024. HADES: Hardware-Assisted Distributed Transactions in the Age of Fast Networks and SmartNICs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 785–800. doi:10.1109/ISCA59077.2024.00062

[41] Weihao Kong, Yifan Hao, Qi Guo, Yongwei Zhao, Xinkai Song, Xiaqing Li, Mo Zou, Zidong Du, Rui Zhang, Chang Liu, Yuanbo Wen, Pengwei Jin, Xing Hu, Wei Li, Zhiwei Xu, and Tianshi Chen. 2024. Cambricon-D: Full-Network Differential Acceleration for Diffusion Models. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 903–914. doi:10.1109/ISCA59077.2024.00070

[42] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. 2022. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embed. Comput. Syst.* (apr 2022).

[43] Zhijing Li, Yuwei Ye, Stephen Neuendorffer, and Adrian Sampson. 2022. Compiler-Driven Simulation of Reconfigurable Hardware Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 619–632. doi:10.1109/HPCA53966.2022.00052

[44] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. 2022. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation . In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 35–56. doi:10.1109/MICRO56248.2022.00018

[45] Yunzhe Liu, Xinyu Li, Tingting Zhang, Tianyi Liu, Qi Guo, Fuxin Zhang, and Jian Wang. 2024. AVM-BTB: Adaptive and Virtualized Multi-level Branch Target Buffer. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 17–31. doi:10.1109/ISCA59077.2024.00012

[46] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, USA, 280–292. doi:10.1109/MICRO.2014.50

[47] Guillem López-Paradís, Adrià Armejach, and Miquel Moretó. 2021. gem5 + rtl: A Framework to Enable RTL Models Inside a Full-System Simulator. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 29, 11 pages. doi:10.1145/3472456.3472461

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*.

[49] Michele Marazzi, Tristan Sachsenweger, Flavien Solt, Peng Zeng, Kubo Takashi, Maksym Yarema, and Kaveh Razavi. 2024. HiFi-DRAM: Enabling High-fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 133–149. doi:10.1109/ISCA59077.2024.00020

[50] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (June 2023), 25 pages. doi:10.1145/3591234

[51] Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. 2015. Architectural Simulators Considered Harmful. *Micro, IEEE* (Nov 2015), 4–12.

[52] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. UDP: Utility-Driven Fetch Directed Instruction Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1188–1201. doi:10.1109/ISCA59077.2024.00089

[53] Gagandeep Panwar, Muhammad Laghari, Esha Choukse, and Xun Jian. 2024. DyLeCT: Achieving Huge-page-like Translation Performance for Hardware-compressed Memory. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1129–1143. doi:10.1109/ISCA59077.2024.00085

[54] Julian Pavon, Ivan Vargas Valdivieso, Carlos Rojas, Cesar Hernandez, Mehmet Aslan, Roger Figueras, Yichao Yuan, Joël Lindegger, Mohammed Alser, Francesc Moll, Santiago Marco-Sola, Oguz Ergin, Nishil Talati, Onur Mutlu, Osman Unsal, Mateo Valero, and Adrian Cristal. 2024. QUETZAL: Vector Acceleration Framework for Modern Genome Sequence Analysis Algorithms. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 597–612. doi:10.1109/ISCA59077.2024.00050

[55] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on.* IEEE, 1–4.

[56] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *44th ISCA*. 14 pages.

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. doi:10.1145/2499370.2462176

[58] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *IISWC*.

[59] Berkeley Architecture Research. 2024. Chipyard Framework. https://github.com/ucb-bar/chipyard.

[60] Yesin Ryu, Yoojin Kim, Giyong Jung, Jung Ho Ahn, and Jungrae Kim. 2024. Native DRAM Cache: Re-architecting DRAM as a Large-Scale Cache for Data Centers. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1144–1156. doi:10.1109/ISCA59077.2024.00086

[61] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 475–486. doi:10.1145/2485922.2485963

[62] Kaustubh Shivdikar, Nicolas Bohm Agostini, Malith Jayaweera, Gilbert Jonatan, José L. Abellán, Ajay Joshi, John Kim, and David Kaeli. 2024. NeuraChip: Accelerating GNN Computations with a Hash-based Decoupled Spatial Accelerator. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 946–960. doi:10.1109/ISCA59077.2024.00073

[63] Mojtaba Abaie Shoushtary, Jose Maria Arnau, Jordi Tubella Murgadas, and Antonio Gonzalez. 2024. Memento: An Adaptive, Compiler-Assisted Register File Cache for GPUs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 978–990. doi:10.1109/ISCA59077.2024.00075

[64] Sawan Singh, Arthur Perais, Alexandra Jimborean, and Alberto Ros. 2024. Alternate Path μ-op Cache Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1230–1245. doi:10.1109/ISCA59077.2024.00092

[65] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Trans. Des. Autom. Electron. Syst.* 27, 4, Article 32 (feb 2022), 27 pages.

[66] Boyu Tian, Yiwei Li, Li Jiang, Shuangyu Cai, and Mingyu Gao. 2024. NDPBridge: Enabling Cross-Bank Coordination in Near-DRAM-Bank Processing Architectures. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 628–643. doi:10.1109/ISCA59077.2024.00052

[67] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. 2021. Reticle: a virtual machine for programming modern FPGAs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 756–771. doi:10.1145/3453483.3454075

[68] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[69] Xin Wang, Jagadish Kotra, Alex Jones, Wenjie Xiong, and Xun Jian. 2024. Counterlight Memory Encryption. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 724–738. doi:10.1109/ISCA59077.2024.00058

[70] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai "Helen" Li, and Yiran Chen. 2024. NDSEARCH: Accelerating Graph-Traversal-Based Approximate Nearest Neighbor Search through Near Data Processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 368–381. doi:10.1109/ISCA59077.2024.00035

[71] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 77–89.

[72] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 268–281.

[73] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2019. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *HPCA*.

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

[75] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2024. Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 931–945. doi:10.1109/ISCA59077.2024.00072

[76] Zhiheng Yue, Huizheng Wang, Jiahao Fang, Jinyi Deng, Guangyang Lu, Fengbin Tu, Ruiqi Guo, Yuxuan Li, Yubin Qin, Yang Wang, Chao Li, Huiming Han, Shaojun Wei, Yang Hu, and Shouyi Yin. 2024. Exploiting Similarity Opportunities of Emerging Vision AI Models on Hybrid Bonding Architecture. In *2024 ACM/IEEE*

51st Annual International Symposium on Computer Architecture (ISCA). 396–409. doi:10.1109/ISCA59077.2024.00037

[77] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. 2022. PDL: a high-level hardware design language for pipelined processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 719–732. doi:10.1145/3519939.3523455

[78] Monir Zaman, Mustafa M. Shihab, Ayse K. Coskun, and Yiorgos Makris. 2018. Towards a Cross-Layer Framework for Accurate Power Modeling of Microprocessor Designs. In *2018 28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 229–236. doi:10.1109/PATMOS.2018.8464153

[79] Jianping Zeng, Tong Zhang, and Changhee Jung. 2024. Compiler-Directed Whole-System Persistence. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 961–977. doi:10.1109/ISCA59077.2024.00074

[80] Nathan Zhang, Rubens Lacouture, Gina Sohn, Paul Mure, Qizheng Zhang, Fredrik Kjolstad, and Kunle Olukotun. 2024. The Dataflow Abstract Machine Simulator Framework. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 532–547. doi:10.1109/ISCA59077.2024.00046

[81] Yilong Zhao, Mingyu Gao, Fangxin Liu, Yiwei Hu, Zongwu Wang, Han Lin, Ji Li, He Xian, Hanlin Dong, Tao Yang, Naifeng Jing, Xiaoyao Liang, and Li Jiang. 2024. UM-PIM: DRAM-based PIM with Uniform & Shared Memory Space. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 644–659. doi:10.1109/ISCA59077.2024.00053

[82] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng