TxHDL: A Hardware Description Language That Is a Rust Library

Filip Filmar and Dragiša Janković

***Abstract*—TxHDL is a hardware description language with no grammar of its own. A design is a Rust program: a unit is a struct, its state is a field, its behaviour is an `async fn` that waits for clock edges, and a value that lives across a wait is a register the source never named. The language is a library and a few macros; the parser, the type checker, the module system and the tooling are Rust's. The same source runs as a cycle-accurate simulation, writes a waveform, and lowers to a netlist in Verilog and VHDL, and the build simulates that netlist under nvc and Verilator against the trace the simulation wrote, so a design that passes is right in three forms at once. The largest design so far is Vreteno, a three-stage RV32IMC core with machine-mode traps, an interrupt line and a bus with its data memory, a timer and a serial port on it: written in the subset the lowering reads, checked in lockstep against a reference model, checked as a netlist against its own trace, synthesized, placed and routed on an Artix-7 at 100 MHz in 2151 LUTs, its data memory a device in block RAM, and run on the board, where it said on its serial port what the simulation had said. The language is proven on hardware. This paper presents the system and its results.**

***Index Terms*—Hardware description languages, embedded domain-specific languages, Rust, RISC-V, high-level synthesis.**

I. Introduction

A hardware description language usually arrives with a parser, a type checker, a module system and an editor of its own, each written once more. TxHDL does not. It starts from a specification of a language shaped like Rust and deletes every construct Rust already provides: structs, generics, traits, modules, functions, `async` and `await`. What remains is the part that means hardware, and it fits in a library of five modules and a handful of macros. A design is an ordinary Rust crate, compiled by `rustc`, and the type checker finds two drivers on a wire, a clock-domain crossing without a synchroniser, and a cycle count in a combinational body, as type errors.

That library is also a runtime. A design runs as a simulation with a stated model of time, one process per unit, and writes a waveform in FST or VCD. It is also the input of a lowering: an attribute on a unit reads its `run` loop and emits Verilog and VHDL, and the build simulates the netlist against the trace the Rust simulation wrote, under two simulators, so the three forms of a design are held to one behaviour. Fig. 1 shows the whole.

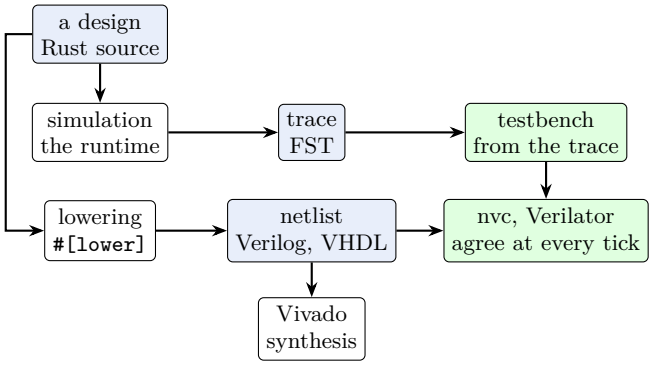

Fig. 1. One source, three forms. The runtime simulates it and writes a trace; the lowering emits its netlist; a testbench generated from the trace replays it into the netlist under nvc and Verilator, and the build fails unless every register and output agrees at every tick.

The paper presents the system in that order. Section II is the language: what a unit is, how time is modelled, how units are wired, and how control flow on a signal is spelled. Section III is the simulation and its traces. Section IV is the lowering and the checks that close the loop. Section V is Vreteno, an RV32IMC core written in the language, from a reference model to synthesis. Section VI gathers the numbers.

II. The Language

A. A unit

Listing 1 is a complete unit: a counter that ticks an output once every eight cycles while enabled. It is the unit on the project's cheat sheet, and it shows everything a unit has. The struct holds the state, one register of three bits. The `Unit<In, Out>` trait names the ports as type parameters, which `#[lower]` fills in from `run`, whose arguments they are: an input wire and an output wire here, channels, or tuples or structs of either, elsewhere. The body is one `loop`. Its first statement waits for the rising edge of the default clock; then it reads, plainly, the register and the input; then it drives, with `with!` the register under its condition and with `set` an output wire. The attribute `#[lower]` says the unit lowers, Section IV, and `derive(Trace)` names its fields in a waveform.

```
#[derive(Trace, Default)]
pub struct Counter {
    pub n: Reg<U<3>>,
}

#[lower]
impl Unit for Counter {
    async fn run(&mut self, enable: In<Bit>, tick: Out<Bit>) {
        loop {
            DefaultClock::rising().await; // wait, then read
            with!(self <= { enable.get() ? n: self.n + 1 });
            tick.set(self.n == 7);
        }
    }
}
```

Both authors are independent. Filip Filmar is the corresponding author, at f@hdlfactory.com; Dragiša Janković is reached at linkedin.com/in/dragisa-jankovic.

This paper presents TxHDL as it stands: the language, its runtime, the traces and netlists it produces, the checks that hold them together, and the Vreteno RV32IMC core built with it. Every listing is included from the project repository `HDL/txhdl` at build time, every waveform is drawn from a trace the build produced, and every number is from a run the build made. The companion documents [1] hold the rest.

The authors used a large language model, Claude, as an assistant in exploring the concepts and in writing and constructing the documents and the programs; every commit of the repository says so and carries the prompts that led to it, verbatim.

Built by Bazel from commit `34467ec22df6` on 2026-10-02.

Listing 1. A unit: state, ports, one loop that waits, reads and drives. From `ex_cheat.rs`.

Values are `Bit`, IEEE 1164's nine-valued `Logic`, and the sized vectors `U<N>` and `I<N>`, with Rust's operators for arithmetic, shifts, bitwise logic and the compares, and methods for what has no operator: `slice`, `concat`, `sext`, `zext`, the arithmetic shift and the signed compare. A struct of values with `derive(Value)` is a value with a width, and a transaction that moves over a channel is such a struct with `derive(Transaction)`. There is no width arithmetic in the type system, since stable Rust has none; a width that is the sum of two others is stated, as `concat::<_, M>` states it, the operand's width left to Rust.

B. Time

Time runs in ticks. A clock is a type with a period and a phase in ticks, and its rising edge is an event a process waits for. The executor polls every process once per tick and then commits the drives; a process that has crossed an edge at this tick cannot cross it again, so a second wait written after a first waits for the next edge. Fig. 2 draws the rule. A process waits, then reads: a register, in an expression or through `get`, is what the last edge latched, a wire's `get` what it carries, a memory's `read` its asynchronous port. Drives are deferred: `set` takes effect at the end of the tick, so a read after a drive in the same iteration still sees what the edge latched, as it does in hardware. The default clock has period two, so its falling edge is a tick of its own and can be waited for as well.

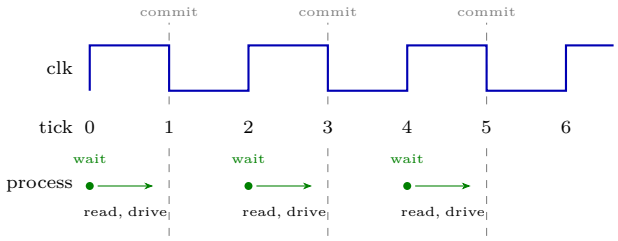

Fig. 2. The model of time. A process waits for an edge, reads state as that edge left it, and drives; the drives commit at the end of the tick. One wait per edge: the next wait is the next edge.

C. Wires and channels

A wire is created once and split into two ends, `Out` and `In`; `Out` is not `Clone`, so a second driver is a value that cannot be made, and the oldest error in hardware design arrives as a move error. A channel is a transaction plus the handshake the language supplies, split into `Tx` and `Rx`. Its runtime form is an elastic buffer of two, Fig. 3: a send is an offer, `valid` and `data` this tick and the transaction in the buffer at the end of it, allowed when there is room; a receive is a take, `ready` this tick and the head gone at the end of it. Both commit as register drives do, so two units that run every cycle pass one transaction per cycle with `valid` and `ready` registered on both sides, and no combinational path joins them. A receiver's `wait` is an event like an edge: the next edge at which the channel holds a transaction.

A clock-domain crossing is a type, `Crossing<T, A, B>`, and the only way a value in clock `A` becomes one in clock `B`; a wire whose ends are in different clocks does not type-check. An interface is declared once with its members and its roles,

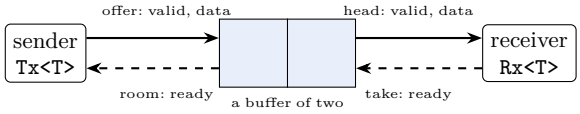

Fig. 3. A channel. The sender offers into a two-entry buffer and reads whether there is room; the receiver takes from its head. Every wire is registered at the buffer, so neither side sees the other's logic.

```
/// The ALU: ten operations by `f3`, with `sub` telling subtract from
/// add and the arithmetic shift from the logical.
#[lower]
fn alu(f3: U<3>, sub: Bit, a: U<32>, b: U<32>) -> U<32> {
    let sh = b.slice::<0, 5>();
    select!(f3.raw() => {
        0 => mux(sub, a - b, a + b),
        1 => a << (sh.raw() as usize),
        2 => lt_signed(a, b).zext(),
        3 => Bit::from(a < b).zext(),
        4 => a ^ b,
        5 => mux(sub, sra(a, sh.raw() as usize), a >> (sh.raw() as usize)),
        6 => a | b,
        _ => a & b,
    })
}
```

Listing 2. `select!`: the ALU of the Vreteno core, one value chosen by the function code. From `core.rs`.

and `interface!` emits a struct per role with the directions the role has, so a bus master, a target and a monitor are three views of one declaration.

D. Control flow on a signal

Hardware does not branch: both arms of a choice exist and a condition selects. Three macros spell that, each named for what it is rather than for the Rust it resembles, and Rust's `if` is read the same way. `with!(self <= { c ? r: v })` is the drives of the unit, each under its condition, a group of them under one with an `else`, and `when!(c => self { r: v })` the group with the condition first: both arms exist, a multiplexer chooses, nothing branches. `case!` is the group with many arms, Rust patterns with guards and alternatives, the first match winning, so a state machine is one `case!` over its state register. An `if` over a `bool`, with the drives under it written as `set`, and its `else if` and `else`, is the same priority chain, and its arms nest. `select!` chooses among values rather than drives, `match` on a value by another name, and lowers to a chain of multiplexers; `mux` is its two-arm case. Listing 2 is the Vreteno core's ALU, a `select!` over the function code, in a function under `#[lower]`, which the lowering writes in where the step calls it.

E. Pipelines and configurations

An `async fn` of awaited operators is a pipeline: each operator takes the cycles its implementation takes, a value alive across an await is a register, and the depth follows from the operators rather than from the source. `#[pipeline(mul = 3, add = 1)]` emits the Verilog of such a function with the stated latencies, and the same function with `mul = 1` is two stages shorter without an edit. A configuration is a trait with associated types and constants, a build is a type alias, and a design is generic over its configuration; `main` names the build, and elaboration is the program running.

III. Simulation and Traces

`Running::new(unit.run(inputs, outputs))` is a design under test, stepped a tick or a cycle at a time while the testbench

reads its wires between steps; `Reg` and `Mem` give out second handles for that. `Wave::from_env` names what to watch, by the field names `derive(Trace)` registered, and writes an FST or a VCD where the environment points; a compound value is one vector and one signal per field beside it, and an enum is written with its variant names. Fig. 4 is the counter of Listing 1, drawn by the build from the trace the run wrote, through the project's converters and a TikZ generator; no figure in this paper was drawn by hand.

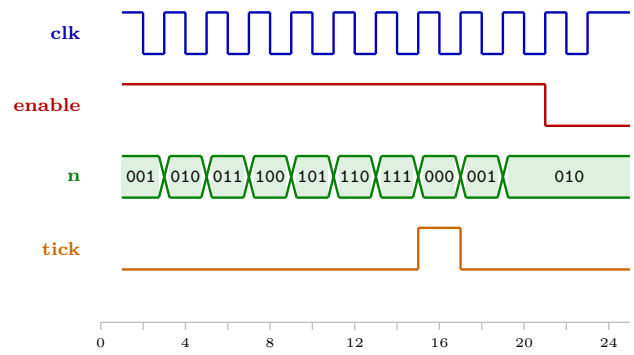

Fig. 4. The counter, from its trace: `enable` drops at cycle 10, `n` counts while it is high, `tick` marks seven.

IV. From Source to Netlist

A. The lowering

`#[lower]` reads a unit's `run` as it is written and builds a netlist from it, without a second language: one loop, or several under `join2`, each a process on the rising or the falling edge of its clock and a clocked block of its own; register and port reads into names, `with!`, `when!`, `case!` and `select!`, the datapath operators, memories written through `at` and read by index, and channel ports as a valid/ready handshake whose wait guards the drives. A `let` that computes is a wire named for it, so the netlist reads as the source does. What the macro does not read it refuses with a message that names the construct, and the subset it reads is also plain Rust: a unit that lowers is the unit that simulates, one file. A unit whose fields are units lowers too: its `run` makes the channels between the children and joins their `run`s, and the netlist is one module holding an instance of each child's, every channel between them the runtime's buffer of two as a module of its own, so a design composed of parts is checked whole at its ports. Listing 3 is what the counter became.

```
‘timescale 1ns/1ps
module counter(
  input clk,
  input rst,
  input enable,
  output tick
);
  reg [2:0] n = 3’h0;
  always @(posedge clk) begin
    if (rst) begin
      n <= 0;
    end else begin
      if (enable) begin
        n <= (n + 3’h1);
      end
    end
  end
  assign tick = (n == 3’h7);
endmodule
```

Listing 3. The counter's Verilog, as the build printed it. The VHDL says the same.

B. The checks

A netlist that is only printed is a claim. The build simulates it, twice. The example is run with two environment variables set, so beside its trace it writes the unit's VHDL and Verilog; a generator reads the trace and the unit's ports and writes a testbench, in VHDL and again in Verilog, that replays the trace's inputs into the entity and asserts, tick by tick, that its registers and outputs are what the trace holds; nvc simulates the VHDL and Verilator the Verilog, nvc built from source by the build and Verilator fetched by it as a prebuilt binary; and each test passes only if every comparison held. The testbench's timing is the model of time made literal: an input the trace holds at tick $2k$ is applied at $2k-1$, a register is checked at $2k+1$, a wire at $2k-1$, and a channel's registered inputs a tick later than a wire's. Table I lists the units that pass; Fig. 5 is one of them, a stage between two channels, whose handshake on both sides is what its testbench replays. A unit is also checked live: a Verilog or VHDL module becomes a unit of the runtime, Verilator in process or nvc as a child process behind it, and the examples document runs the stage beside its own Verilog and its own VHDL in three pipelines on the same offers, which take the same words at the same ticks.

TABLE I
Lowered units simulated against their traces, each under nvc and under Verilator.

Unit	What it shows
counter	a register, `with!`, an output
sampler	two processes, one per edge
blinky	a configuration's constants baked in
sequencer	`case!` on a state enum, guards
ALU	ten operators by `case!`
scratchpad	a memory written and read
decoder	`slice`, `concat`, `sext`, `select!`
stage	two channels, the handshake
MAC	`mul::<M>`, a product at a stated width
tap	a send under `if`, a receive with no wait
buffer	the runtime's channel, as a part
divider	wires kept as fields, seen and checked
Gray counter	functions under `#[lower]`, inlined
Vreteno	the whole core, 647 ticks
router	the bus fanned out by address
data memory	on the core's bus, the same run
timer	on the core's bus, the same run
serial port	on the core's bus, the same run
FIFO	`D` words between two channels
station	two tagged inputs, beside hand Verilog
stream ALU	four latencies, results out of order
top	a unit of units, a station and the ALU

The second simulator earned its place on its first run. Every unit's VHDL had agreed with its trace; the same units' Verilog, from the same statements, had two defects VHDL cannot have, an unsized number inside a concatenation and a shift made logical by Verilog's whole-expression signedness, and neither showed until the Verilog was simulated. Both were found by the core and fixed in the emitter.

V. Vreteno

Vreteno is a 32-bit RISC-V core, the RV32I base integer set with the M and C extensions, and the largest design in the language. Fig. 6 is its structure. It is one unit and one process. The fetch stage reads the instruction memory at the program counter into an instruction register, with its program counter and a valid bit. The execute stage decodes the fields of

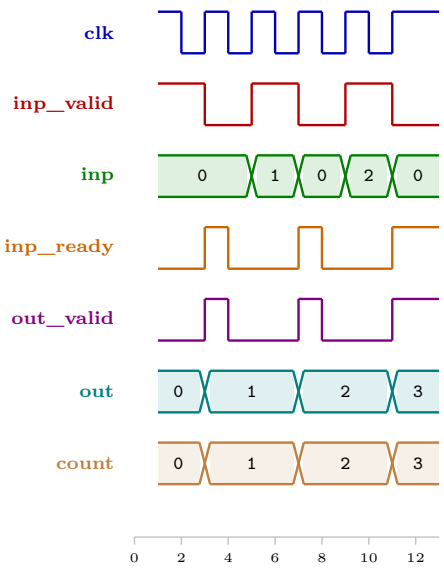

Fig. 5. A stage between two channels, from its trace: a word offered every other cycle is taken, passed on incremented in the same cycle, and counted. The netlist agrees at every tick.

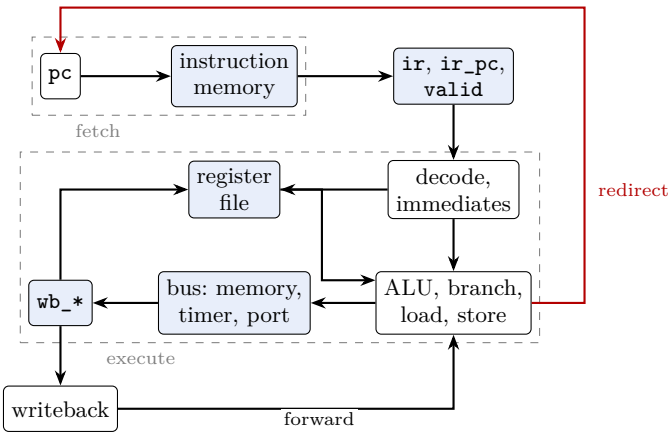

Fig. 6. Vreteno: three stages in one process. The instruction register and the writeback registers are the boundaries; a taken branch, jump or trap redirects the fetch and costs one bubble; the value in writeback is forwarded into execute, a loaded one by a stall of one cycle instead.

the word in that register with `slice`, `concat` and `sext`, reads two operands from the register file, computes with the ALU of Listing 2, sends a store or a load out on the bus, where the data memory is a device like the timer and the serial port and a load's answer comes back into the writeback registers, and, on a taken branch or a jump, redirects the fetch and squashes the word fetched that cycle, which is the one-cycle penalty. The writeback stage extends a loaded word, writes the register file and retires; an operand the writeback stage is about to write is forwarded into execute, except a loaded one, which lands at the edge and would be too late: the instruction that wants it waits one cycle, the only stall there is. The core has machine-mode traps: an `ecall` or a word it does not know saves its address and cause in `mepc` and `mcause`, saves and clears the interrupt enable in `mstatus`, and redirects to `mtvec` as a taken branch redirects; `mret` returns; an interrupt line, pending in `mip` and enabled in `mie`, is taken in place of the instruction in execute, which retires writing nothing; a bus of two channels, a request out and the word read back, through a router that decodes a device per address range, on which the core-local interruptor raises the second interrupt when its count reaches its compare and the third when a program writes its own software bit, a serial port says what the program writes and takes what a terminal types, and the data memory itself is a device, the channels between them all being the runtime's buffer of two as hardware; the six CSR instructions read and write the eight control registers. Multiply and divide are one sequencer over the magnitudes, a multiply one step in the part's DSP blocks and a division thirty-two restoring steps, the signs put back at the end, and the instruction stalls for them, off the critical path. The whole is written in the subset the lowering reads, Listing 4 being the fetch's drives, so the one file simulates and is a netlist.

The core is checked three ways. A reference model of the instruction set, written as a plain program over the architectural state, runs in lockstep with it: the model steps when the core retires an instruction, and after every cycle the program counter, the thirty-one registers, the control registers and the halt must agree, and at the halt the data memory must.

```
// The fetch's next counter: zero on reset, parked one word
// past the halting instruction, held while halted or
// stalled, the target on a redirect, else the next
// instruction, two bytes on or four. Reset, park and
// hold are known early and the redirect late, so the two
// candidates fold the early conditions in and the redirect
// chooses last, one multiplexer from the instruction memory.
self.redirect.set(
    (run & jump) | int_take | resume_take | st_take | wb_fault,
);
let redirect = self.redirect.get();
let park = run & stop;
let hold = stall | (stop & !run) | in_debug;
let zero = U::<32>::from(0u32);
let width = mux(short, U::<32>::from(2u32), U::<32>::from(4u32));
let advance = mux(hold, fetch_pc, fetch_pc + width);
let go = mux(rst, zero, mux(park, link, advance));
let jmp = mux(
    rst,
    zero,
    mux(
        park,
        link,
        mux(
            resume_take,
            dpc,
            mux(int_take | st_take | wb_fault, mtvec, target),
        ),
    ),
);
self.pc.set(mux(redirect, jmp, go));
case!(rst => {
    Bit::One => { self.valid <= Bit::Zero },
    _ if wb_fault.to_bool() => { self.valid <= Bit::Zero },
    _ if stall.to_bool() => {},
    _ if stop.to_bool() => { self.valid <= Bit::Zero },
    _ if (in_debug | dbg_take).to_bool() => {
        self.valid <= Bit::Zero
    },
    _ => {
        self.ir <= fetched;
        self.ir_bad <= f_fault;
        self.ir_c <= Bit::from(short);
        self.ir_pc <= fetch_pc;
        self.valid <= !redirect
    },
});
```

Listing 4. The fetch's drives: the next counter as two candidates, the early conditions folded into each and the late redirect choosing between them; the instruction register under a `case!`. From `core.rs`.

The demonstration program, which traps twice, and sixty-four random programs of two hundred instructions, with traps among them, pass; a broken shift, a branch that does not squash, or a return from a trap with the wrong status fails at its first use. The netlist is checked as every lowered unit is, under nvc and Verilator, with the program in its instruction memory. And the Verilog is synthesized, Table II, on an Artix-7 with a clock constraint of ten nanoseconds, by Vivado, which the build installs itself from the vendor's archive into a cache the machine shares. The instruction memory became a read-only memory in two block RAM tiles, one for each of the two words the fetch reads, since an instruction may cross from one into the next, the data memory's four byte lanes four block RAMs in two, since the third stage registers their read, and the register file distributed RAM. Placed and routed with the same constraint, the core meets ten nanoseconds with 0.296 ns to spare; the critical path runs from a register field of the instruction register through an operand's read and its forwarding into the ALU's compare and the writeback value, and is three quarters routing. Fig. 8 is the opening of the demonstration, from the trace, and Fig. 7 the serial port's line as the demonstration's O goes out, from the same trace: the write into the port, then the start bit, the byte low bit first and the stop bit, four cycles each.

The same netlist goes the other way as well. Mapped onto Nangate45, an open 45 nm standard cell library, by Yosys, and then floorplanned, placed, routed and timed by OpenROAD, both of them fetched by checksum as Debian packages and unpacked by the build, it comes to 16 995 cells on a die 293 μm square, meets 3.000 ns with 0.1575 ns to spare, and routes with 0 violations. What changes between the two targets is the memories. An FPGA has a block RAM to put the instruction memory in and a small distributed memory to put the register file in, and a standard cell library has neither: the instruction memory, which nothing in the core writes, becomes a mask ROM of gates whose size depends on the program in it, and the register file becomes a thousand and twenty four flip-flops. That is why the same core is 489 flip-flops on the board and 1 499 on silicon. The flow, the layout and everything a tapeout would still want are in [2].

TABLE II
VRETENO ON AN ARTIX-7, AFTER SYNTHESIS.

Resource	Used
LUTs, as logic	2151
LUTs, as distributed RAM	48
Flip-flops	489
Block RAM tiles	2
DSP blocks	4
Errors, critical warnings	0

A. Programs written for it

A core that runs only what its own assembler emits is a core that has been tested, not one that can be programmed. Vreteno is RV32IMC, so compilers target it, and two do: Rust through the ruleset's own toolchain, and C++ through the GNU bare-metal RISC-V toolchain, whose `rv32imc/ilp32` multilib is this machine exactly and carries `newlib` and a `libstdc++` built for it. Both are fetched by the build against a checksum. Nothing is installed to compile for the core, as nothing is installed to simulate or to synthesise it.

A program for the core is built for a different machine than the build runs on, so a rule with a platform transition carries

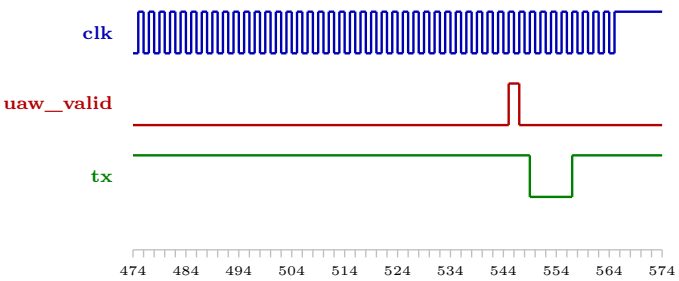

Fig. 7. A byte out on Vreteno's serial port, from the trace: the write reaches the port and the line carries the frame, a start bit, the byte low bit first and a stop bit, four cycles each; the later requests are the program's polls for the next byte.

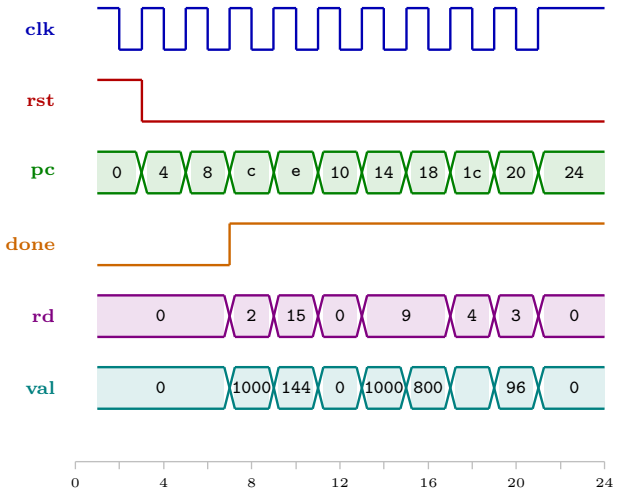

Fig. 8. Vreteno's first cycles, from its trace: reset, then an instruction per edge with the fetch counter running ahead, and the writeback decoded into its register and value.

it across, and `//tools/elf2vreteno` turns the linked ELF into the two images the core is loaded from. Both languages end at the same place: the same machine, the same terminal on the serial line, the same assertion that the line came out and that the core stopped itself.

What is interesting is not that it works but what the machine will not allow, because the core meets each restriction by going quietly wrong rather than by refusing, so each had to become a check that speaks. The instruction memory is 4 KiB, which the loader truncates to without a word and the fetch wraps within, and a program linked anywhere but zero is never reached; each is refused by the image tool, by name, where it can be said out loud. A load from the boot memory was one such restriction too, a constant beside the code that the program could not read, until every load went to the bus with the boot memory on it, read-only at zero; `.rodata` now sits beside the code. The compressed instructions were another, until the core ran them; now the compilers' own targets are the core's, and the programs are a quarter smaller for it.

That size is the real constraint on what a standard library can mean here. The half of it that is templates and constants is usable and both programs use it; the half that needs an operating system underneath, a heap, file descriptors, an unwinder, is not, and no amount of toolchain fetches it into existence. The C++ program says hello and prints a number its compiler

worked out, in 63 words.

VI. Results

What the system does, in numbers counted from the tree on September 28, 2026. The runtime is nine Rust modules, one macro crate and one crate of parts, with one external crate, an FST writer. A hundred and four examples compile, each one concept, and every one that runs writes a waveform the examples document draws. A hundred and two netlists, the units of Table I among them, are simulated as VHDL under nvc and as Verilog under Verilator against the traces their Rust simulations wrote, and agree at every tick; three hundred and fifty-four tests run under `bazel test //...`, and every one of them is hermetic, the simulators built from source or fetched as a binary and the synthesis tool installed by the build. Vreteno is 1664 lines of core, 137 of data memory, 260 of timer and 249 of serial port, with the interrupt controller, 510 lines, and the router, one unit for any number of ports, in the parts library, checked in lockstep against a 752-line model on a demonstration program and sixty-four random ones, with interrupts raised at random, checked as a netlist on 647 ticks with every register compared at each, synthesized in two minutes to the figures of Table II, placed and routed at 100 MHz, and programmed into the board, where it said OK on its serial port and echoed the three bytes typed back, the six bytes its simulation had produced, and lit its LEDs as the board top says, which an author saw and photographed: the language is proven on hardware, end to end, from a Rust source to a board that answers. Twenty-four documents describe the system, this one included, and each is built by the same command that builds the designs, with every listing included from the tree and every figure drawn from a trace, so none can drift from the code.

VII. Related Systems

Chisel [3] and SpinalHDL embed a hardware language in Scala with the same structural architecture: a module is a class, a bundle has directions, `when` and `Mux` are control flow on a signal, and elaboration is the program running. TxHDL's structural half is that architecture in Rust, where a second driver is a move error and a configuration is a type. Its behavioural half, an `async fn` that is a sequence, a pipeline or a process according to how it is driven, has no counterpart in Chisel; its nearest relatives are XLS [4], whose Rust-shaped DSLX the compiler schedules into pipelines, and Calyx [5], whose `seq` and `par` are an `await` and a `parallel!` seen from the other side. RHDL [6] embeds a hardware language in Rust with the clock domain as a type parameter, which TxHDL takes directly. What sets TxHDL apart from all of them is not any one construct but the loop the build closes: one source, three forms, and a test that fails when they disagree.

VIII. Where the Work Is

The work is done on the authors' own Forgejo instance, `git.hdlfactory.com`, in the repository `HDL/txhdl`: the mainline, the pull requests, the continuous build that renders every document and runs every check, and the nightly release of every document as a PDF. That instance is by invitation. A mirror of the mainline is public on GitHub, at `github.com/filmil/hdl-txhdl`, pushed on every change and never forced; it holds the same tree, the same history and the same releases, and is read-only. So the tree this paper is built from is public, and every listing in it can be read there; the way in for a contributor is the corresponding author, who opens an account on the instance. The whole of it is under the Apache License, version 2.0.

IX. Conclusion

A hardware description language can be a Rust library. The language is small enough to fit on one page, the runtime simulates it with a stated model of time, the lowering turns it into Verilog and VHDL, and the build holds the three to one behaviour under two simulators. A RISC-V core written in it runs, lowers, is checked against its own trace, runs at 100 MHz on an Artix-7 with its data memory in block RAM, and has run on the board, saying on its serial port what its simulation said: the language is proven on hardware. The path that limits the clock is a jump through a register.

References

[1] F. Filmar and D. Janković, "TxHDL documents," project repository `HDL/txhdl`, `//docs:cover`, 2026.
[2] F. Filmar and D. Janković, "A tapeout prep in TxHDL: Vreteno on an open 45 nm node," project repository `HDL/txhdl`, `//docs:tapeout`, 2026.
[3] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," in *Proc. 49th Design Automation Conference*, 2012.
[4] Google, "XLS: Accelerated HW synthesis," `github.com/google/xls`.
[5] R. Nigam et al., "A compiler infrastructure for accelerator generators," in *Proc. ASPLOS*, 2021.
[6] S. Basu, "RHDL: A hardware description language embedded in Rust," `github.com/samitbasu/rhdl`.