SAMIT BASU

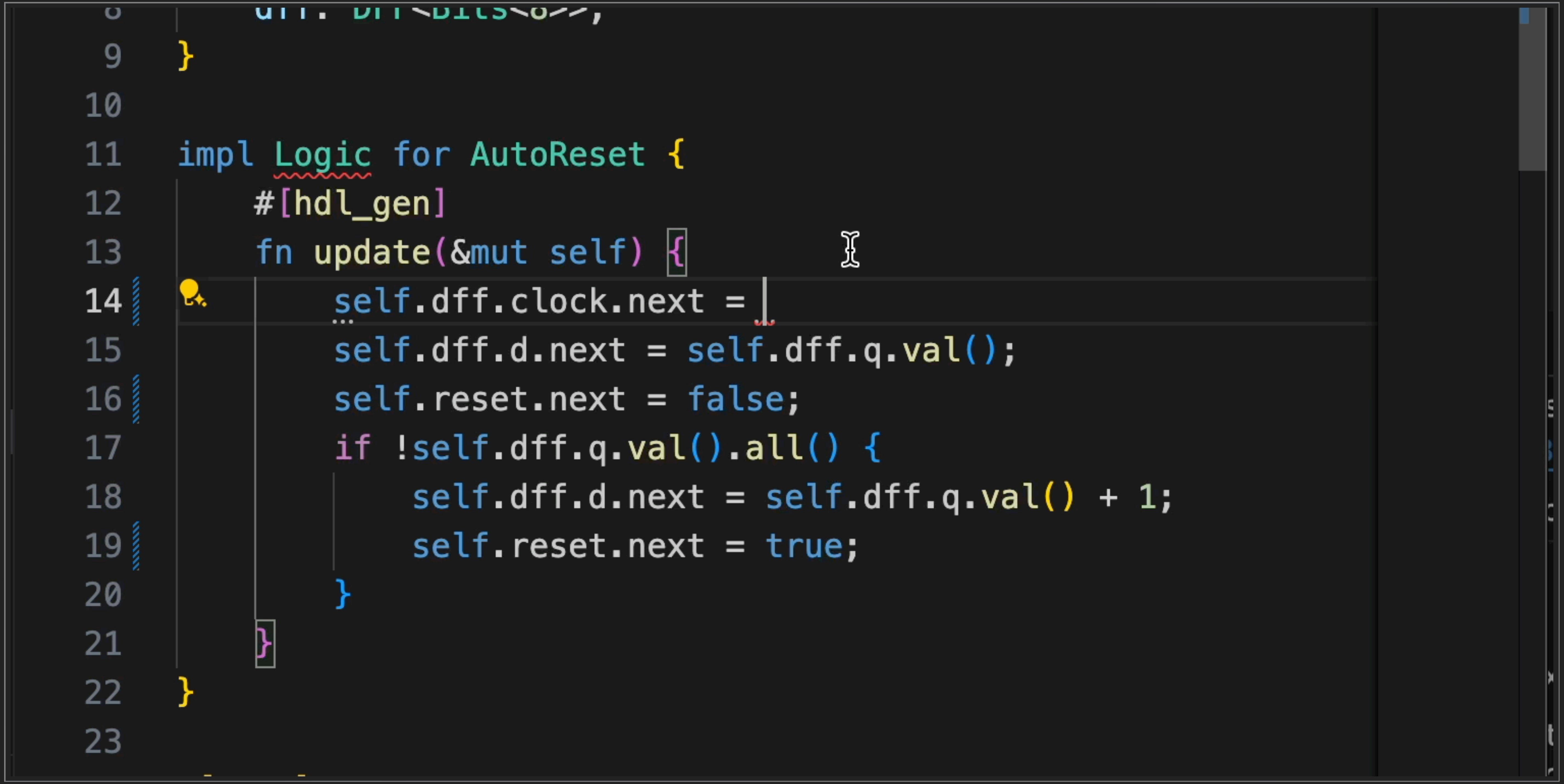
OSDA 24

VIRTUAL

RUST AS A HARDWARE DESCRIPTION LANGUAGE

WHY RUST?

- **Safety** - stop errors ~~at compile~~ at edit time!
- **Reuse** & Sharing - package manager, generics, and crates
- **Testing** - built-in, batteries included
- **Tooling** - IDE support



```
}

impl Logic for AutoReset {
    #[hdl_gen]
    fn update(&mut self) {
        self.dff.clock.next = 
        self.dff.d.next = self.dff.q.val();
        self.reset.next = false;
        if !self.dff.q.val().all() {
            self.dff.d.next = self.dff.q.val() + 1;
            self.reset.next = true;
        }
    }
}
```

RustHDL

- Open Source
- Transpile AST
- Event-based simulator
- Medium scale designs
  - developed
  - tested
  - commercially deployed

https://rust-hdl.org/

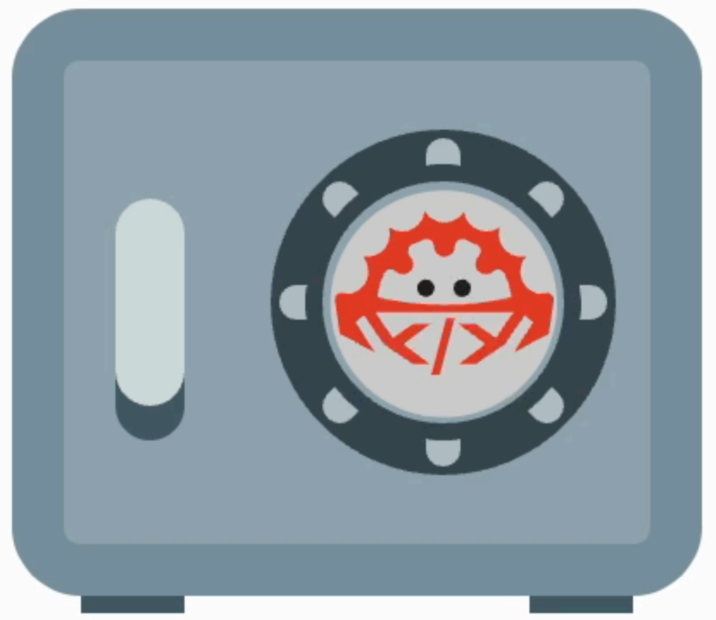
(Source, Docs, Discord)

rust-hdl.org
RustHDL
Tutorial
API Docs
Crates 📦
GitHub

RustHDL

Write FPGA Firmware using Rust! 🎉🦀

Alchitry Cu - Blinky Tutorial - 5min ⏱️



Safe

Use `rustc` to check the validity of your firmware with strongly typed interfaces that are checked at *compile* time.

Powerful

Easily package complex designs into easy-to-reuse modules that you can reuse easily. Connecting components is simple and missing or erroneous connections are caught at compile time or with the built in static analysis passes

Batteries Included

Need an asynchronous FIFO? Or a SDR memory controller? Or a one shot? Use the provided set of *widgets* to get started. Most are generic and can be used to handle arbitrary data types.

COMPARISON TO OTHER HDLS

	RustHDL	CHISEL	MyHDL
Embedding Language	Rust	SCALA	Python
Types	Strong and static	Strong and static	Loose and dynamic
Coding style	Behavioral imperative	Structural Generators	Generators
Ecosystem	crates.io	MavenCentral	pypi
Metaprogramming	Generics and Macros	Generics	Yes

STRUCTURE

- Circuits are *composed* into structs
- Input/output signals (ports) also composed
- pub controls visibility
- Encapsulation and reuse
- Designs are hierarchical

```rust
#[derive(LogicBlock)]
// v--- Modules use simple composition of structs
pub struct SPIMaster<const N: usize> {
    // Clocks are a type ---v
    pub clock: Signal<In, Clock>,
    // Signal has direction --v
    pub data_outbound: Signal<In, Bits<N>>,
    // Signal has type --------v
    pub start_send: Signal<In, Bit>,
    //v--- pub visibility control
    pub data_inbound: Signal<Out, Bits<N>>,
    // Bus ------v
    pub wires: SPIWiresMaster,
    // Local scratchpad ---v
    local_signal: Signal<Local, Bit>,
    // D Flip Flop --v
    state: DFF<SPIState>,
    //           ^-- With enumerated value
    cs_off: Constant<Bit>,
    //           ^-- (Rust) Run time initialized constant
}
```

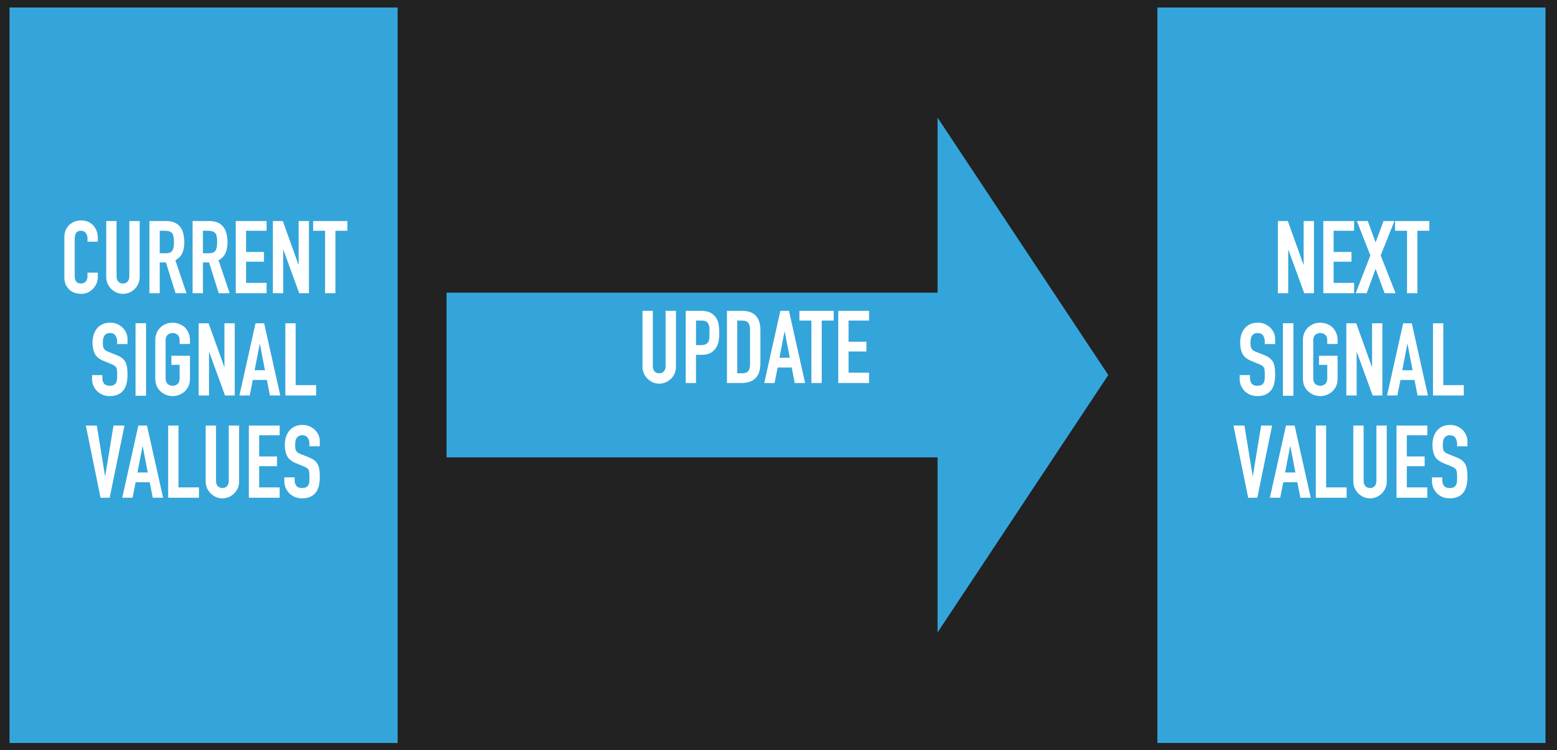
CONNECT

- State encapsulated in D Flip Flops, BRAMs, etc.
- Code is **valid** Rust
- Latch prevention via yosys
- Rust-like (sane) scoping

```rust
// Design is parametric over N - the size of the counter
impl<const N: usize> Logic for Strobe<N> {
  // v-- Attribute to generate HDL
  #[hdl_gen]
  // v-- Update function is attached to any logic circuit
  fn update(&mut self) {
    // v-- latch prevention
    self.counter.d.next = self.counter.q.val();
    // v-- mux control signal
    if self.enable.val() {
      //  v-- value assigned if mux control is true
      self.counter.d.next = self.counter.q.val() + 1;
    }
    // v-- combinatorial logic
    self.strobe.next = self.enable.val() &
      (self.counter.q.val() == self.threshold.val());
    // v-- higher priority mux for previous mux output
    if self.strobe.val() {
      self.counter.d.next = 1.into();
    }
  }
}
```

TOWARDS PURE FUNCTIONS

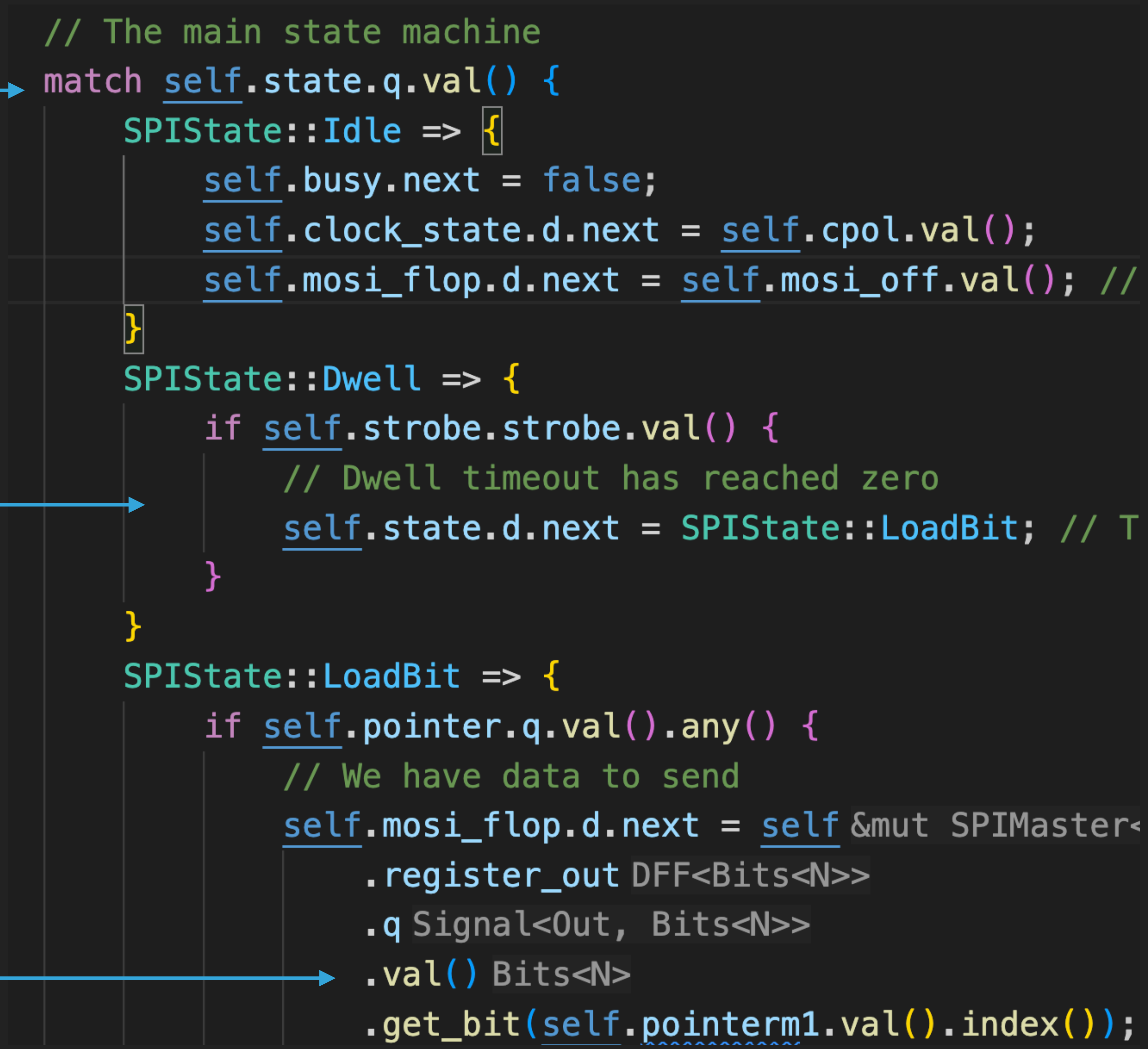
- Idea taken from Lucid HDL
- Connect function maps current values to "next ones"
- Last update wins
- No surprises from blocking vs unblocking assignments…



```rust
// Design is parametric over N - the size of the counter
impl<const N: usize> Logic for Strobe<N> {
  // v-- Attribute to generate HDL
  #[hdl_gen]
  // v-- Update function is attached to any logic circuit
  fn update(&mut self) {
    // v-- latch prevention
    self.counter.d.next = self.counter.q.val();
    // v-- mux control signal
    if self.enable.val() {
      //  v-- value assigned if mux control is true
      self.counter.d.next = self.counter.q.val() + 1;
    }
    // v-- combinatorial logic
    self.strobe.next = self.enable.val() &
      (self.counter.q.val() == self.threshold.val());
    // v-- higher priority mux for previous mux output
    if self.strobe.val() {
      self.counter.d.next = 1.into();
    }
  }
}
```

FINITE STATE MACHINES

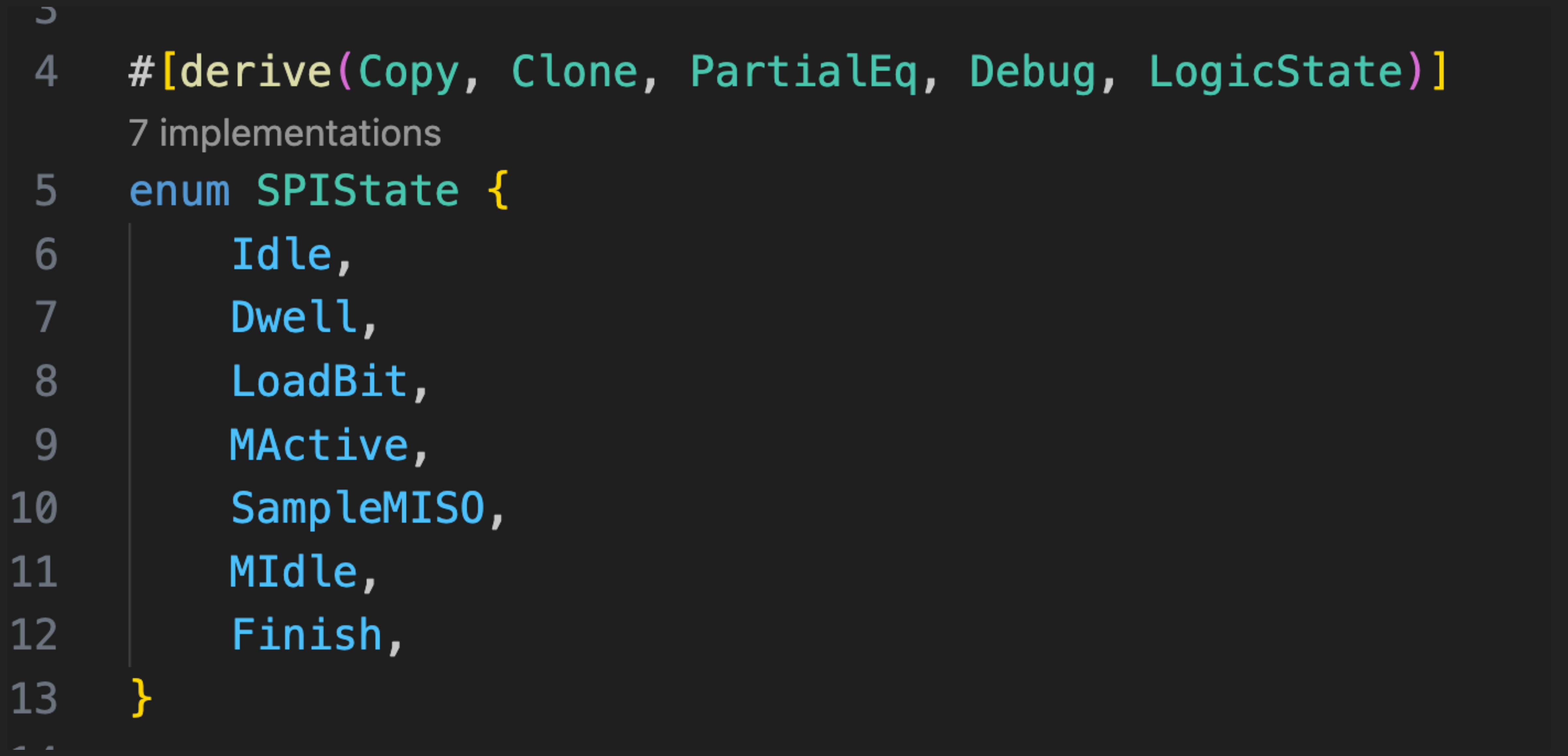
- Match on current state
- Rust match is exhaustive
- Next state is assigned here
- Improves readability
- Enums are simple and typed
- Note type hints from RA



```
// The main state machine
match self.state.q.val() {
    SPIState::Idle => {
        self.busy.next = false;
        self.clock_state.d.next = self.cpol.val();
        self.mosi_flop.d.next = self.mosi_off.val(); //
    }
    SPIState::Dwell => {
        if self.strobe.strobe.val() {
            // Dwell timeout has reached zero
            self.state.d.next = SPIState::LoadBit; // T
        }
    }
    SPIState::LoadBit => {
        if self.pointer.q.val().any() {
            // We have data to send
            self.mosi_flop.d.next = self &mut SPIMaster<
                .register_out DFF<Bits<N>>
                .q Signal<Out, Bits<N>>
                .val() Bits<N>
                .get_bit(self.pointerm1.val().index());
```

SIMPLE ENUMS

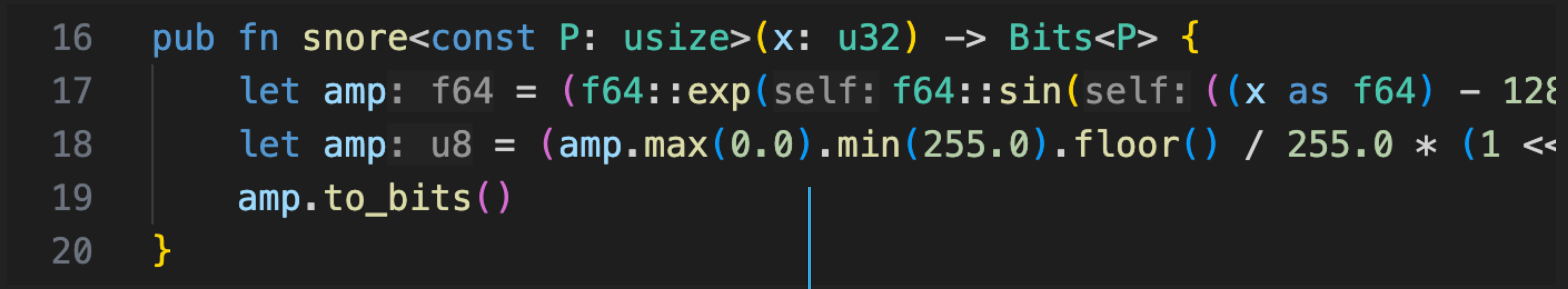
- Unlike C enums, these are strongly typed
- Rust guarantees that enum values are always valid
- Cannot carry data, just aliases for values
- Values are unspecified to allow for encoding optimization



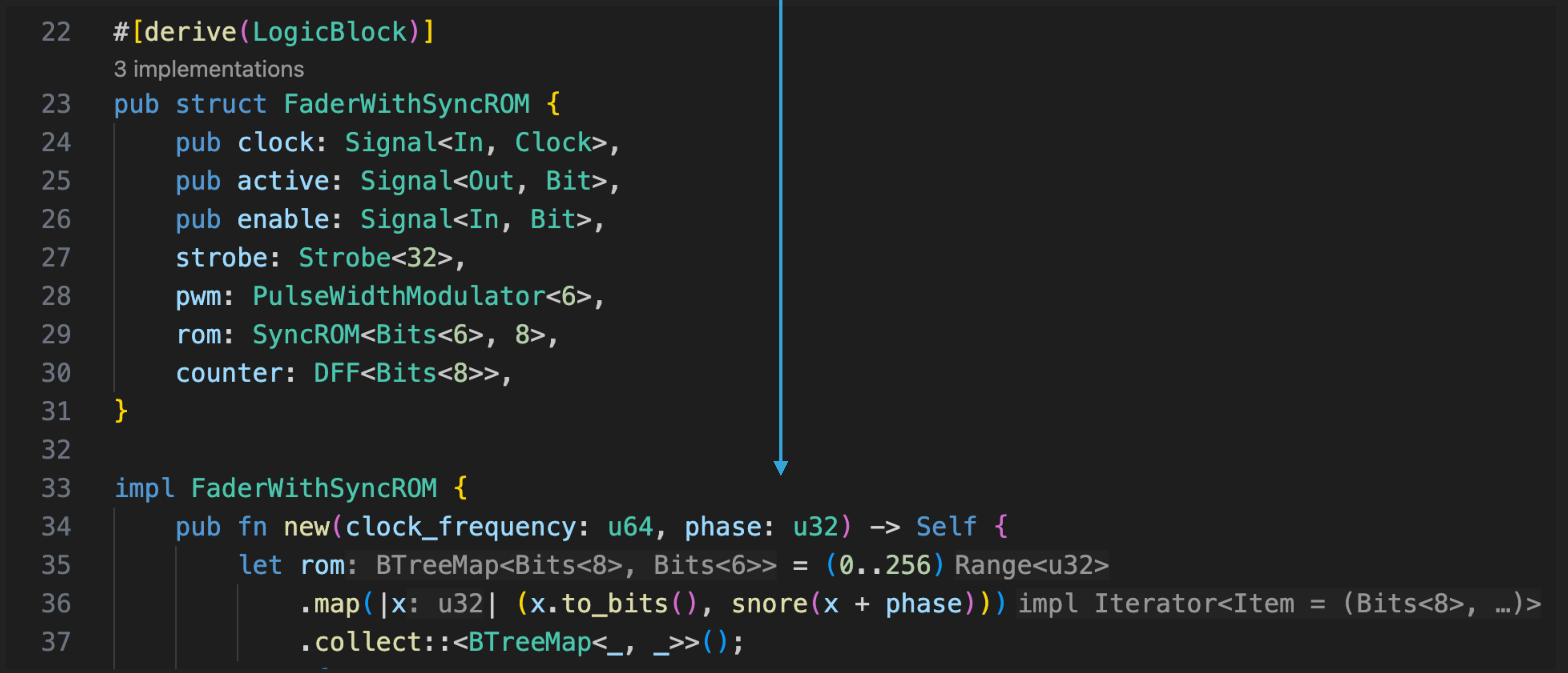
```
#[derive(Copy, Clone, PartialEq, Debug, LogicState)]
7 implementations
enum SPIState {
    Idle,
    Dwell,
    LoadBit,
    MActive,
    SampleMISO,
    MIdle,
    Finish,
}
```

CONFIGURATION

- Circuits are configured at RustHDL run time
- Allows for arbitrary complexity in configuration
- Avoids need for external programs to compute tables/functions/etc.
- Add unit tests and functional tests to the configuration



```rust
pub fn snore<const P: usize>(x: u32) -> Bits<P> {
    let amp: f64 = (f64::exp(self: f64::sin(self: ((x as f64) - 12
    let amp: u8 = (amp.max(0.0).min(255.0).floor() / 255.0 * (1 <
    amp.to_bits()
}
```



```rust
#[derive(LogicBlock)]
3 implementations
pub struct FaderWithSyncROM {
    pub clock: Signal<In, Clock>,
    pub active: Signal<Out, Bit>,
    pub enable: Signal<In, Bit>,
    strobe: Strobe<32>,
    pwm: PulseWidthModulator<6>,
    rom: SyncROM<Bits<6>, 8>,
    counter: DFF<Bits<8>>,
}

impl FaderWithSyncROM {
    pub fn new(clock_frequency: u64, phase: u32) -> Self {
        let rom: BTreeMap<Bits<8>, Bits<6>> = (0..256) Range<u32>
            .map(|x: u32| (x.to_bits(), snore(x + phase))) impl Iterator<Item = (Bits<8>, …)>
            .collect::<BTreeMap<_, _>>();
```

INTERFACES

- Logical grouping of signals
- Can be nested
- Allow signals in *both* directions

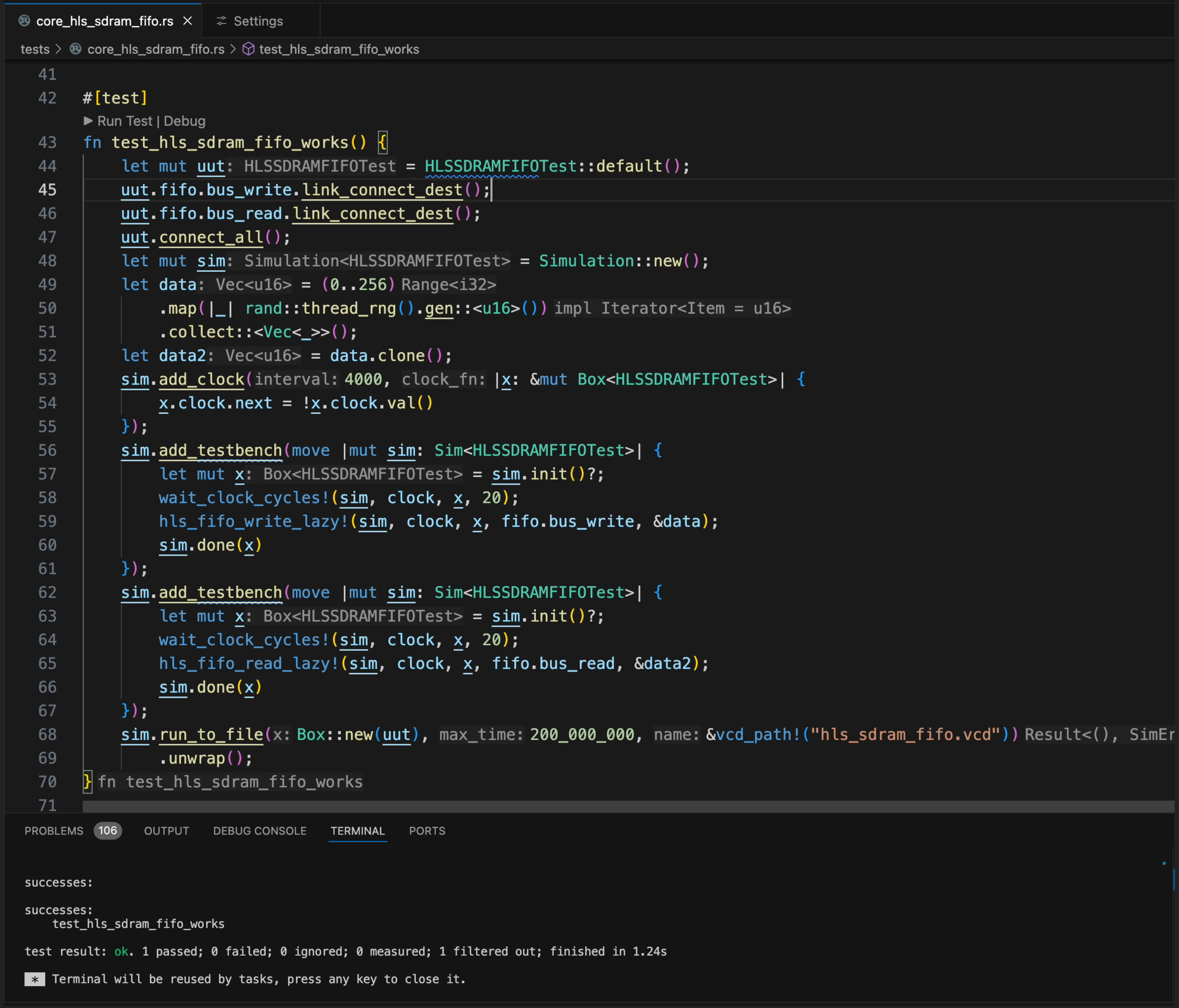
```rust
//        v-- indicates its an interface
#[derive(LogicInterface, Clone, Debug, Default)]
//        v-- "mating" interface
#[join = "SDRAMDriver"]
pub struct SDRAMDevice<const D: usize> {
  // Interfaces can be generic --^
  pub clk: Signal<In, Clock>,
  pub we_not: Signal<In, Bit>,
  pub read_data: Signal<Out, Bits<D>>,
  pub write_enable: Signal<In, Bit>,
}
```

- Can `join` interfaces with a single line of code

```rust
fn update(&mut self) {
    I2CBusDriver::join(&mut self.controller.i2c,
      &mut self.test_bus.endpoints[0]);
}
```

SIMULATE FROM YOUR IDE

- No special configuration or tooling
- Batteries included
- Full debugging available
- Set breakpoints based on state/ etc.



core_hls_sdram_fifo.rs × Settings

tests > core_hls_sdram_fifo.rs > test_hls_sdram_fifo_works

```
#[test]
▶ Run Test | Debug
fn test_hls_sdram_fifo_works() {
    let mut uut: HLSSDRAMFIFOTest = HLSSDRAMFIFOTest::default();
    uut.fifo.bus_write.link_connect_dest();
    uut.fifo.bus_read.link_connect_dest();
    uut.connect_all();
    let mut sim: Simulation<HLSSDRAMFIFOTest> = Simulation::new();
    let data: Vec<u16> = (0..256) Range<i32>
        .map(|_| rand::thread_rng().gen::<u16>()) impl Iterator<Item = u16>
        .collect::<Vec<_>>();
    let data2: Vec<u16> = data.clone();
    sim.add_clock(interval: 4000, clock_fn: |x: &mut Box<HLSSDRAMFIFOTest>| {
        x.clock.next = !x.clock.val()
    });
    sim.add_testbench(move |mut sim: Sim<HLSSDRAMFIFOTest>| {
        let mut x: Box<HLSSDRAMFIFOTest> = sim.init()?;
        wait_clock_cycles!(sim, clock, x, 20);
        hls_fifo_write_lazy!(sim, clock, x, fifo.bus_write, &data);
        sim.done(x)
    });
    sim.add_testbench(move |mut sim: Sim<HLSSDRAMFIFOTest>| {
        let mut x: Box<HLSSDRAMFIFOTest> = sim.init()?;
        wait_clock_cycles!(sim, clock, x, 20);
        hls_fifo_read_lazy!(sim, clock, x, fifo.bus_read, &data2);
        sim.done(x)
    });
    sim.run_to_file(x: Box::new(uut), max_time: 200_000_000, name: &vcd_path!("hls_sdram_fifo.vcd")) Result<(), SimEr
        .unwrap();
} fn test_hls_sdram_fifo_works
```

PROBLEMS 106 OUTPUT DEBUG CONSOLE TERMINAL PORTS

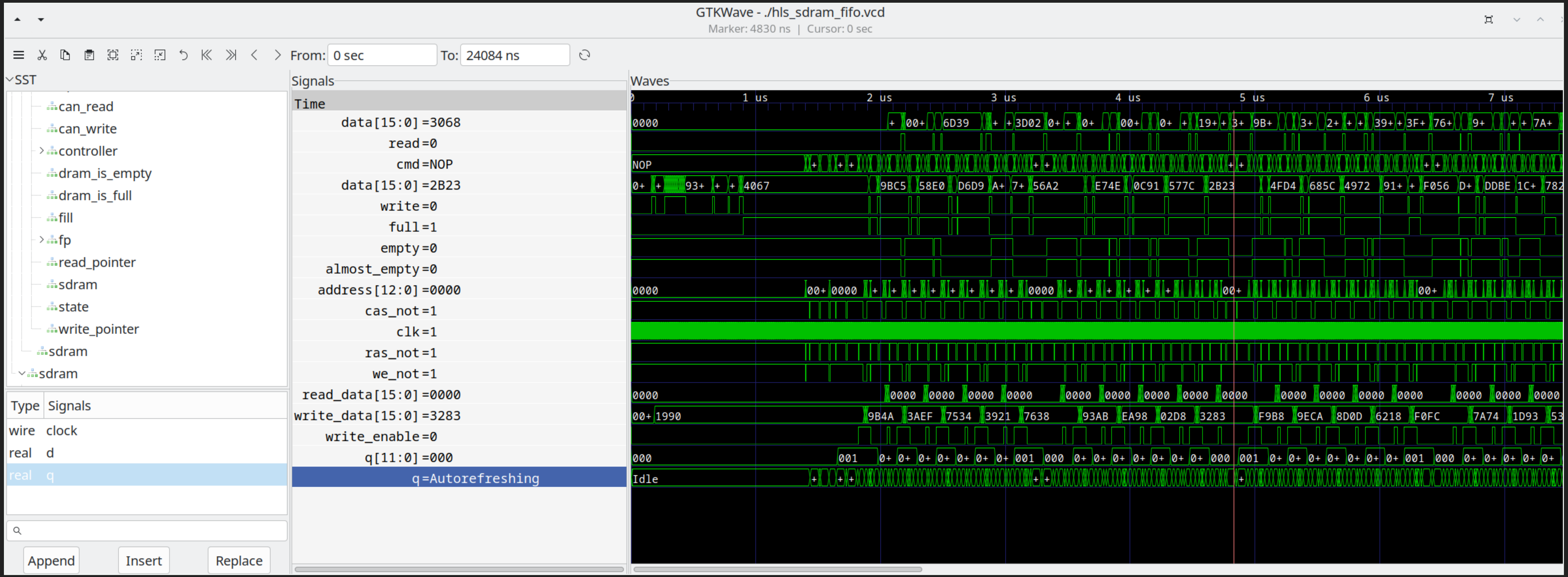
```
successes:

successes:
    test_hls_sdram_fifo_works

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 1 filtered out; finished in 1.24s

 * Terminal will be reused by tasks, press any key to close it.
```

SIMULATION



GTKWave - ./hls_sdram_fifo.vcd
Marker: 4830 ns | Cursor: 0 sec

From: 0 sec To: 24084 ns

SST
- can_read
- can_write
- controller
- dram_is_empty
- dram_is_full
- fill
- fp
- read_pointer
- sdram
- state
- write_pointer
- sdram
- sdram

Type	Signals
wire	clock
real	d
real	q

Append Insert Replace

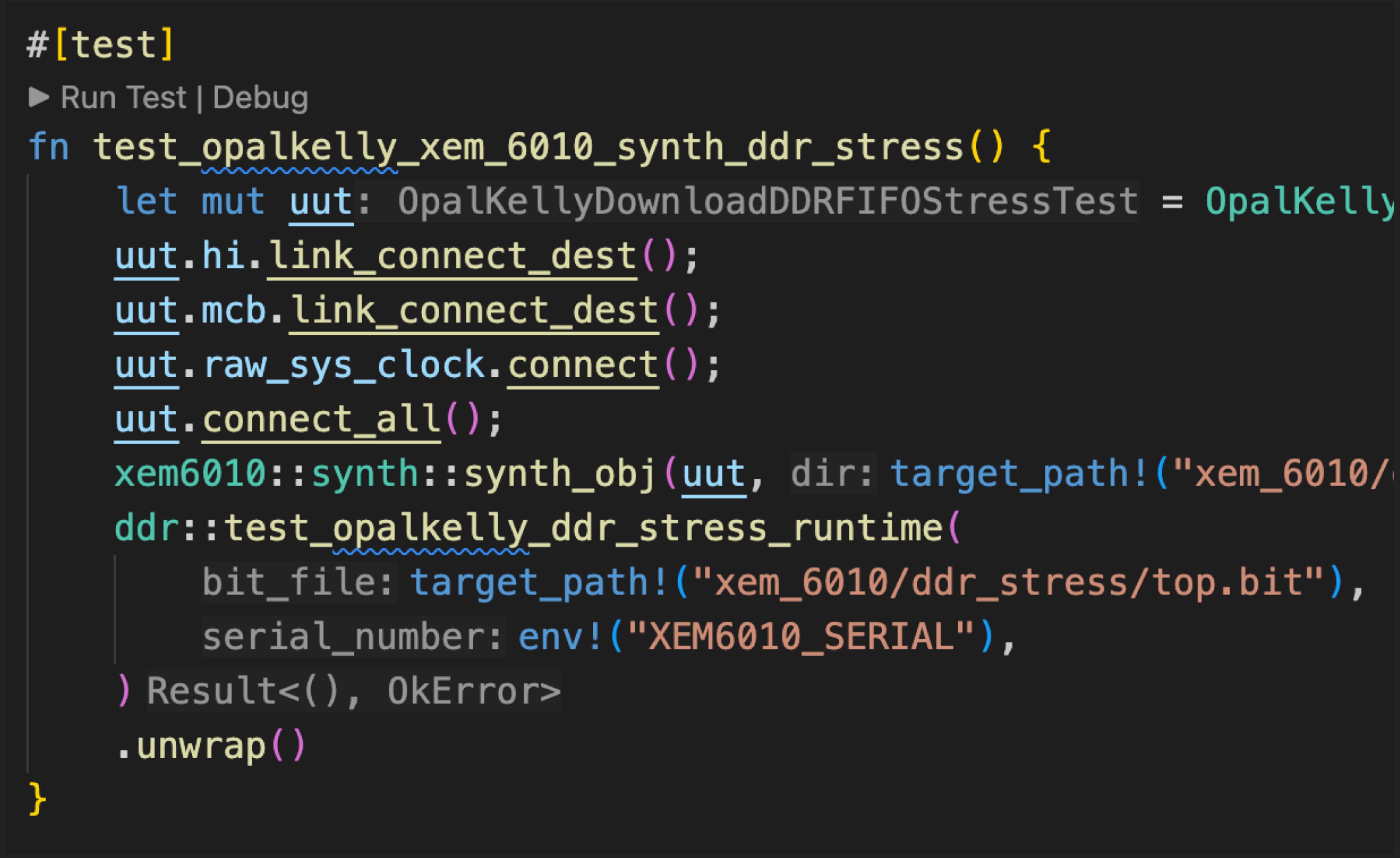
Signals:

```
Time
        data[15:0]=3068
            read=0
             cmd=NOP
        data[15:0]=2B23
           write=0
            full=1
           empty=0
    almost_empty=0
   address[12:0]=0000
         cas_not=1
             clk=1
         ras_not=1
          we_not=1
   read_data[15:0]=0000
  write_data[15:0]=3283
    write_enable=0
         q[11:0]=000
           q=Autorefreshing
```

- Multiple clock domains
- Supports combinatorial inter-module logic and non-synchronous designs
- Example of SDRAM chip, controller, and a FIFO

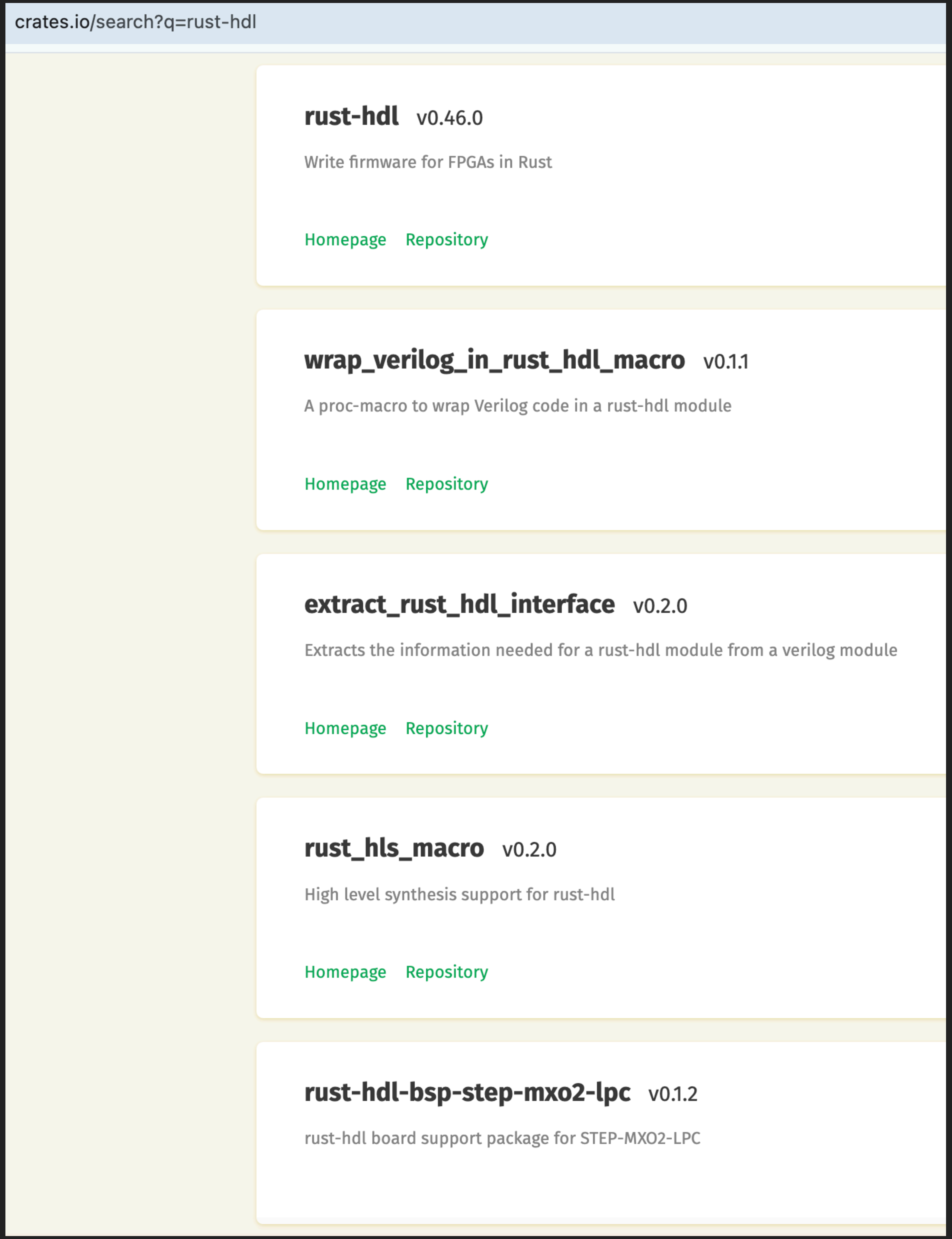
DEEPENING THE TESTING STACK

Test Type	Example
Bench	Hardware
HW Integration	Hardware in the loop
Timing Verification	Timing Analysis
Simulation	Test bench
Unit Test	Functional
HDL Compilation	Latch detection
HDL Generation	Write-before read, etc
Rust Compilation	Type correctness
Rust Linting	Unused expressions
Language Server	Editor squiggles


```
#[test]
▶ Run Test | Debug
fn test_opalkelly_xem_6010_synth_ddr_stress() {
    let mut uut: OpalKellyDownloadDDRFIFOStressTest = OpalKelly
    uut.hi.link_connect_dest();
    uut.mcb.link_connect_dest();
    uut.raw_sys_clock.connect();
    uut.connect_all();
    xem6010::synth::synth_obj(uut, dir: target_path!("xem_6010/
    ddr::test_opalkelly_ddr_stress_runtime(
        bit_file: target_path!("xem_6010/ddr_stress/top.bit"),
        serial_number: env!("XEM6010_SERIAL"),
    ) Result<(), OkError>
    .unwrap()
}
```

CRATES.IO AND THE SHARE ECOSYSTEM

- Board support packages (BSPs) provide hardware and FPGA specific support
- Easily shared on **crates.io**.
- Anyone can package up and contribute
  - Meta-programming features
  - BSPs
  - Reusable circuit components
- Rust & cargo provide excellent version management
- Semantic versioning for hardware



crates.io/search?q=rust-hdl

rust-hdl v0.46.0

Write firmware for FPGAs in Rust

Homepage Repository

wrap_verilog_in_rust_hdl_macro v0.1.1

A proc-macro to wrap Verilog code in a rust-hdl module

Homepage Repository

extract_rust_hdl_interface v0.2.0

Extracts the information needed for a rust-hdl module from a verilog module

Homepage Repository

rust_hls_macro v0.2.0

High level synthesis support for rust-hdl

Homepage Repository

rust-hdl-bsp-step-mxo2-lpc v0.1.2

rust-hdl board support package for STEP-MXO2-LPC

EARLY USER FEEDBACK

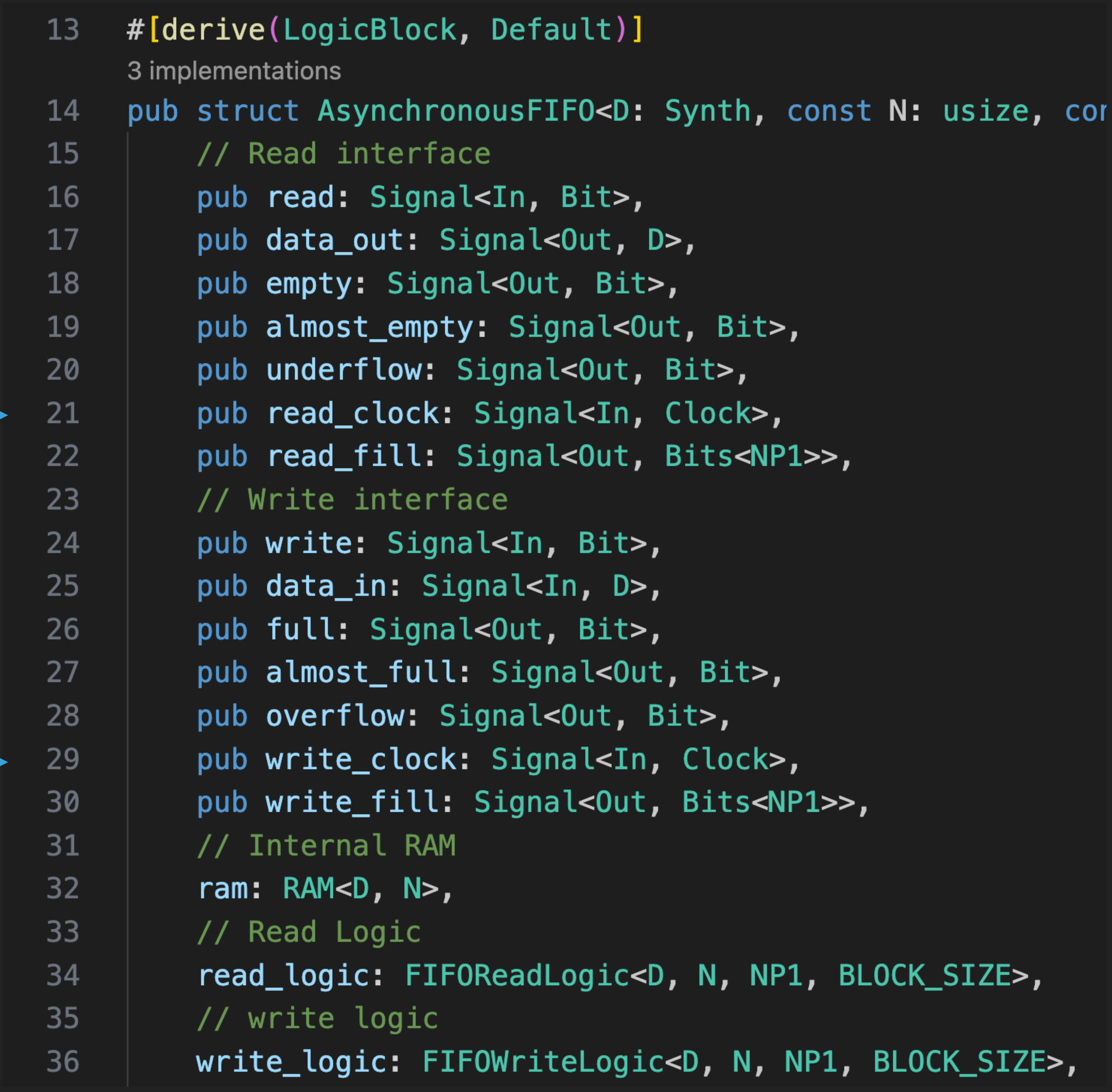
- Need more language features to make it feel more “Rusty”:
  - Local variables
  - Type inference
  - Match/if expressions
  - Early returns
  - Want rich enums, structs, arrays, etc
- Fewer foot guns, more backends, etc.

```
fn update() {
  //  v--- local variable with inferred type
  let a = match self.state {
    //     ^--- match expression
    State::Idle => return(3);
    // Early returns ---^
    State::Busy => 1,
  };
}
```

```
enum OpCode {
  Noop,
  Jump(b24),
  Load{dest: Register, src: Register},
  Save([b56; 8])
}
```

CLOCKING FOOT GUN

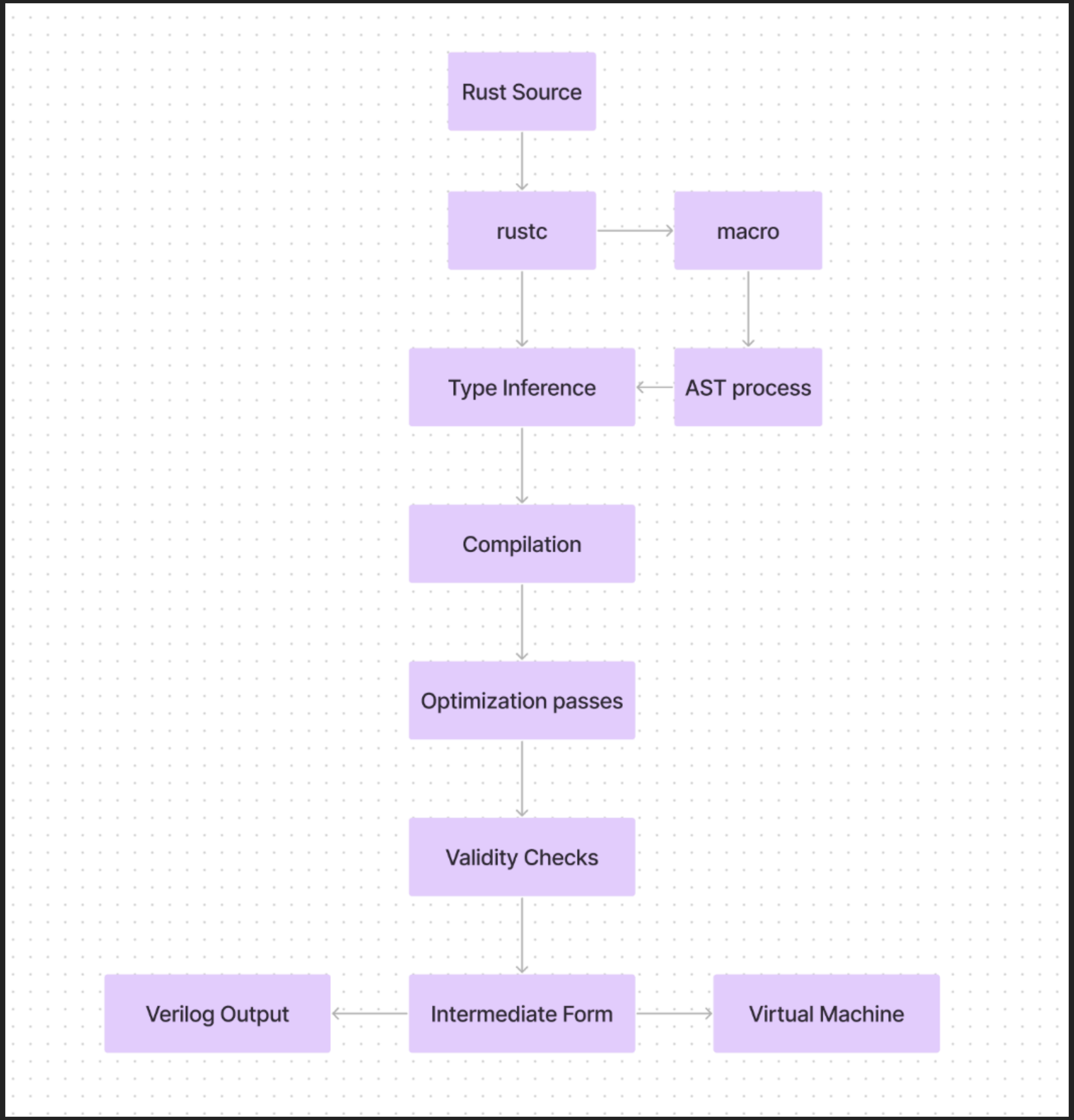
- With multiple clock domains, **Clock** type is not enough
- Which clock?
- Which signals go with what clock?
- RustHDL is no help here…
- Ideally, the type system should constrain:
    - Signal type
    - Signal direction
    - Signal time domain



```
#[derive(LogicBlock, Default)]
3 implementations
pub struct AsynchronousFIFO<D: Synth, const N: usize, co
    // Read interface
    pub read: Signal<In, Bit>,
    pub data_out: Signal<Out, D>,
    pub empty: Signal<Out, Bit>,
    pub almost_empty: Signal<Out, Bit>,
    pub underflow: Signal<Out, Bit>,
    pub read_clock: Signal<In, Clock>,
    pub read_fill: Signal<Out, Bits<NP1>>,
    // Write interface
    pub write: Signal<In, Bit>,
    pub data_in: Signal<In, D>,
    pub full: Signal<Out, Bit>,
    pub almost_full: Signal<Out, Bit>,
    pub overflow: Signal<Out, Bit>,
    pub write_clock: Signal<In, Clock>,
    pub write_fill: Signal<Out, Bits<NP1>>,
    // Internal RAM
    ram: RAM<D, N>,
    // Read Logic
    read_logic: FIFOReadLogic<D, N, NP1, BLOCK_SIZE>,
    // write logic
    write_logic: FIFOWriteLogic<D, N, NP1, BLOCK_SIZE>,
```

RHDL

- Build a complete co-compiler
- Provide type inference, etc.
- Far more Rust-like
- Fewer surprises and limitations
- Easier to use, harder to build
- Stay tuned…



THANK YOU

Samit Basu

basu.samit@gmail.com

Special thanks to the folks that have supported RustHDL and RHDL throughout the years.

Especially J. Vernet, T. Witzel, and the Digital Forge team.

Also special thanks to TheZoq2 for encouraging me to talk about it!

RustHDL.org