h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences

– Faculty of Computer Science –

Is high-level synthesis from Rust possible using existing tools?

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science (B.Sc.)

by
Lennart Eichhorn
Matriculation number: 759253

First Examiner	:	Prof. Dr. Stefan Rapp
Second Examiner	:	Prof. Dr. Ronald Charles Moore

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 5. Juli 2023

Lennart Eichhorn

ABSTRACT

This thesis explores generating digital circuits from Rust functions. Synthesis of these low-level hardware descriptions from specifications written in high-level languages is known as high-level synthesis (HLS). Existing HLS tools primarily focus on older systems programming languages or custom languages for their input specifications. Regarding productivity and useability, Rust provides various benefits over these older systems programming languages. This research uses an existing HLS tool, PandA Bambu, to synthesize specifications written in Rust. Bambu is an open-source research framework for HLS. The present study takes advantage of the fact that the Rust compiler and Bambu share the same intermediate representations, as they are based on the LLVM compiler infrastructure. No prior study has investigated using existing HLS tools with Rust.

A library for using HLS in RustHDL, a Rust-based hardware definition language, was developed. It was used to generate and test designs from different Rust specifications. In most cases, these designs achieved similar results in terms of size and performance than designs generated from equivalent C++ specifications. Most Rust language features are supported for high-level synthesis with Bambu, with the most notable restriction being the disallowance of undefined behavior resulting in program termination (panic). Additionally, this research found that the rust ecosystem can be leveraged and that it is possible to synthesize specifications with dependencies on other crates.

ZUSAMMENFASSUNG

Diese Arbeit untersucht die Generierung digitaler Schaltkreise aus Rust-Funktionen. Der Prozess der Synthese dieser Hardwarebeschreibungen aus Spezifikationen, die in höheren Programmiersprachen geschrieben sind, wird als High-Level-Synthese (HLS) bezeichnet. Bestehende HLS-Werkzeuge verwenden für ihre Eingabespezifikationen hauptsächlich ältere Systemprogrammiersprachen oder speziell für diesen Zweck erstellte Sprachen. In Bezug auf Produktivität und Benutzerfreundlichkeit ist Rust diesen älteren Systemprogrammiersprachen überlegen. Diese Arbeit untersucht die Verwendung eines vorhandenen HLS-Werkzeugs, PandA Bambu, zur Synthese von Spezifikationen, die in Rust geschrieben sind. Bambu ist ein quelloffenes Forschungsframework für HLS. Diese Arbeit nutzt die Tatsache, dass sowohl der Rust-Compiler als auch Bambu dieselben Zwischenrepräsentationen nutzen, da beide auf der LLVM-Compilerinfrastruktur aufbauen. Es sind keine früheren Arbeiten bekannt, die die Nutzung von Rust als Eingabesprache für existierende HLS-Werkzeuge untersuchen.

Um die Verwendung von HLS in Rust zu ermöglichen, wurde eine Softwarebibliothek entwickelt die HLS in RustHDL, eine auf Rust basierenden Hardwarebeschreibungssprache, integriert. Sie wurde verwendet, um Schaltungen aus verschiedenen Rust-Spezifikationen zu generieren und zu testen. In den meisten Fällen erzielten diese Schaltungen ähnliche Ergebnisse in Bezug auf Größe und Leistung wie Schaltungen, die aus äquivalenten C++-Spezifikationen generiert wurden. Die meisten Sprachfunktionen von Rust werden für die High-Level-Synthese mit Bambu unterstützt, wobei die größte Einschränkung darin besteht, dass undefiniertes Verhalten, das zum Programmabbruch führt (Panik), nicht erlaubt ist. Darüber hinaus wurde festgestellt, dass das Rust-Ökosystem genutzt werden kann und dass auch die Synthese von Spezifikationen mit Abhängigkeiten von Rust-Bibliotheken möglich ist.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

Part I

THESIS

1 INTRODUCTION

The popularity of the Rust programming language is rising, and it is one of the most admired programming languages [Sta16] [Sta20] [Sta23] [Bug22]. It integrates modern tooling like standardized dependency management, testing, documentation generation, formatting, and building. In the future, adoption will probably increase further, and it could replace C++ as the most common systems programming language [Bug22]. Rust also provides benefits in domains other than systems programming. It has been shown that Rust can be used for other fields such as GPU programming, web development, or logic programming [Sah22] [Byc22] [Kyr22]. These fields profit from some of the benefits of Rust, like guaranteed memory safety and improved productivity [Bug22] [Cos19].

This paper explores how Rust can be used in FPGA firmware development. Usually, FPGA firmware is developed in a hardware description language (HDL) such as Verilog or VHDL. In these languages, the programmer has to describe the hardware in detail. This low-level approach can lead to efficient designs but is quite time-consuming [Mil20]. The RustHDL project facilitates expressing hardware descriptions in Rust similar to traditional HDLs [Smi21]. In addition to manually designing hardware in an HDL, it is also possible to use high-level synthesis (HLS) to generate hardware descriptions in HDLs from an algorithmic description written in high-level programming languages. Typically systems programming languages are used for writing these specifications. This increases productivity at the cost of slightly less optimized designs [Mil20]. There are multiple HLS tools available that can synthesize HDL descriptions from C++ code. Some of these tools are based on the LLVM compiler infrastructure [Nan16]. The only previous report on using Rust as an HLS language focuses on a limited subset of Rust and its formal verification [Har22].

An investigation to identify HLS tools compatible with Rust was conducted. A modular approach was developed to seamlessly integrate HLS tools with RustHDL. By employing this approach, a proof-of-concept integration with the PandA Bambu HLS framework was achieved, demonstrating the feasibility of using Rust as a source language for HLS. The performance of designs generated from various algorithms was compared with that of designs generated from algorithmically equivalent C++ code. The evaluation showed that, in most cases, the Rust-based workflow produced designs with similar characteristics to those derived from C++-based workflows.

2 STATE OF THE ART

It is assumed that the reader has some knowledge of systems programming and the Rust programming language. This section provides an overview of the topics that are relevant to this paper.

2.1 DESIGN USING TRADITIONAL HARDWARE DESCRIPTION LANGUAGES

Traditionally, logic design is done in hardware description languages (HDL). The two historically relevant ones are Verilog and VHDL. VHDL has a slightly higher level of abstraction and some features that make it easier to manage bigger projects. Both can be used to model the structure of hardware equally efficiently, so the choice is mostly a matter of personal preference [Smi96] [Soz22].

SystemVerilog is a superset of Verilog which merges Verilog with many of the features found in VHDL. In some ways, the relation between Verilog and SystemVerilog is comparable to the relation between C and C++ in terms of features and abstraction. It is the de facto standard HDL nowadays [Fos15] [Soz22].

HDLs are used to describe circuits in a register-transfer level (RTL) abstraction. On RTL, these languages describe registers that can hold state and the combinational (time-independent) logic that connects them. In HDLs, the registers and combinational logic can be bundled into a module to make it reusable. These modules can then be connected to form a larger circuit. This is the basic structure of an HDL design [Soz22].

A typical HDL workflow comprises four phases, design, verification, synthesis, and implementation. In the design phase, the circuit is designed in an HDL. The design is then verified in various ways. For verifying the behavior of a design, test benches are defined. These test benches (usually also written in HDL) instantiate the module of the design under test (DUT), exercise the inputs, and verify that the outputs behave as expected. A logic simulator is used to execute the test benches. This is comparable to unit testing in programming languages. After the design is verified, a logic synthesis tool is used to synthesize the design into an optimized gate-level logic description (netlist). A formal equivalence tool can verify that the netlist is equivalent to the original design. In the implementation phase, the netlist is mapped to the target hardware [Fla20]. Additional verification can take place in any of these phases.

2.2 WHAT IS A FIELD-PROGRAMMABLE GATE ARRAYS

There are two main target platforms for logic design: field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC). FPGAs are off-the-shelf components capable of implementing any digital circuit [Bot21].

As the name field programmable gate array implies, they consist of programmable logic gates with programmable connections between them. ASICs are custom integrated circuits designed for a specific purpose. Design costs and time when targeting FPGAs are much lower than for custom circuits. Another benefit of using FPGAs is that the circuit can be upgraded after deployment. The downsides of using FPGAs are lower performance and higher power consumption compared to custom circuits. They are considered compelling options for small to medium-sized projects, as they do not have the high upfront cost of application-specific integrated circuits (ASIC) [Bot21] [Nan16].

So instead of producing the circuit into a chip, it is possible to program an FPGA that emulates the circuit [Bot21]. For example, a design describing the logic gates and connections that make up a CPU can be programmed to the FPGA, so it will behave exactly like the CPU and can process the same instructions. This can be used for prototyping because the CPU design can be verified in hardware with external peripherals, like memory and I/O devices. If the design is not specially designed to run well on an FPGA, a 'soft' CPU will perform worse on an FPGA than as an ASIC because of the additional overhead caused by programmable logic instead of fixed logic [Bal07].

In the past, FPGAs were mostly used for prototyping hardware development. This is still the most common use case for FPGAs, but it is not the only one. In recent years the usage of FPGAs as pseudo-general-purpose computational accelerators has become more relevant [Nan16].

FPGAs consist of many programmable logic elements (LE) that can represent a simple boolean logic function. Logic elements usually have between 4-6 input bits; the exact number depends on the manufacturer and the model of FPGA. The inputs are used to address a programmable lookup table (LUT). The LUT contains the truth table of the boolean logic function. The output of the LUT is coupled with an output register of the logic element. This way, the LE can also act as a flip-flop (FF) [Bot21].

Multiple logic elements are grouped into a logic block (LB). A logic block provides a local interconnect between the logic elements. This programmable local interconnect can be used to define connections between the inputs of the logic block and the inputs and outputs of the logic elements. The local interconnect is usually realized as a programmable crossbar interconnect [Bot21].

An FPGA is made up of multiple LBs that are connected by global routing. There are multiple ways in which global routing can be realized, but island-style architecture is the most popular. This architecture is based on a two-dimensional grid of horizontal and vertical wires with logic blocks in between. Each logic block has programmable connections to the wires beside it. The horizontal and vertical wires can get connected by programmable switches at their crossings [Bot21].

All the programmable components are controlled by configuration stored in static RAM (SRAM) cells. FPGAs are programmed with a bitstream that contains the individual bits for every SRAM cell in the correct order [Bot21]. They usually have a serial interface where the FPGA can accept the bitstream and program itself.

Every path that the data takes through the circuit needs a certain amount of time. The longer the path and the more things are in it, the longer it takes to take it. A critical path is a path that must meet certain timing requirements for the design to function properly [Mic95]. If the circuit, for example, is clocked, then some actions should probably be finished before the next clock cycle. If they do not finish in time, the clock needs to be slowed down, or the circuit will behave in unexpected ways. For this reason, the slowest critical path limits the maximum frequency [Bot21] [Mic95].

Modern FPGAs improve critical path delay by hardening certain features. Hardening means that a feature is implemented in a fixed way instead of being programmable. Which features are hardened depends on the FPGA model. All modern FPGAs include hardened circuitry for arithmetic operations in their logic blocks. The exact details of the hardening depend on the FPGA model, but it usually involves a fast path for the carry bit between LUTs. This is at least three times faster than an implementation based on pure LUTs [Bot21].

Digital signal processing (DSP) blocks are another common feature of modern FPGAs. They minimize the number of soft logic operations needed to implement common DSP algorithms. Like LBs, they are connected to programmable routing. Depending on the FPGA model, they can be configured to perform operations like multiplication, addition, subtraction, accumulation, and/or multiplication-accumulation in various sizes [Bot21] [Lah19].

Bigger hardware designs almost always require a memory buffer. Building a memory buffer out of logic blocks is possible but not very efficient. Modern FPGAs include hardened memory blocks that can be used as memory buffers. They are called block random access memory (BRAM). BRAMs are over 100 times denser than soft memory made from LUTs. BRAMs are about 25% of the area of modern FPGAs [Bot21] [Lah19].

Converting a register-transfer level (RTL) design to a bitstream is a multi-step process. The first step is synthesis. The RTL design is converted to a structural gate-level description. This is called the netlist. The next step consists of mapping the netlist to the block types available on the specific FPGA model. A series of optimizations can also be performed at this stage. After that, the blocks are placed on specific blocks on the FPGA, and the connections between them are routed. Minimizing the routing distance between logic blocks is crucial because routing accounts for over 50% of the critical path delay. The last step is producing a bitstream. At this point, it is known how every part of the FPGA should behave, so the bitstream for the specific FPGA can be generated [Bot21].

All these steps are performed by a computer-aided design (CAD) tool usually provided by the FPGA vendor. Nearly all FPGA manufacturers integrated these tools into their respective integrated development environments (IDE). Recent developments in the open-source community have led to the development of open-source tools that can perform these steps [Bar23].

NOTE

Different FPGA vendors use different names for the same things [Lah19]. This paper uses LB, made of LEs, made of LUTs, and FFs.

2.3 SYSTEMVERILOG

In SystemVerilog, units of logic can be encapsulated into modules. Modules can have inputs, outputs, and internal states. Listing 1 shows a simple module that implements a blinker. The blinker has a clock input and a blinker output. At every rising edge of the clock, an internal counter is incremented by one. Once the counter reaches 10, the blinker output is toggled, and the counter is reset to zero.

Listing 1. Simple blinker module in SystemVerilog

VERILOG

```verilog
module Blinker (
    input  wire clock,
    output wire blinker
);

  parameter DELAY = 10;
  reg [6:0] counter = 0;
  reg state = 0;

  always @(posedge clock) begin
    counter <= counter + 1;
    if (counter == DELAY - 1) begin
      state   <= ~state;
      counter <= 0;
    end
  end

  assign blinker = state;

endmodule
```

Verification in SystemVerilog can be achieved by using test benches that instantiate the design under test (DUT) and exercise its inputs. Listing 2 shows a testbench for the blinker module. The testbench instantiates the blinker module and connects it to a clock signal. It also asserts that the blinker output toggles every ten clock cycles.

Listing 2. Testbench for the blinker module in SystemVerilog

```verilog
module blinker_tb ();
  reg clock = 1'b0;
  always #1 clock <= ~clock;

  wire blinker;

  Blinker DUT (
      .clock  (clock),
      .blinker(blinker)
  );

  initial begin
    $display("Testing blinker");
    for (int i = 0; i < 20; i++) begin
      assert (blinker == 1'b0);
      #20;
      assert (blinker == 1'b1);
      #20;
    end
    $display("Blinker blinks correctly!");

    $finish();
  end
endmodule
```

2.4 ALTERNATIVE HARDWARE DESCRIPTION LANGUAGES

There are multiple modern HDLs that try to improve on the shortcomings of Verilog and VHDL. Most of them were created to improve productivity and increase the speed of development. Besides structural improvements, they try to bring some features from modern programming languages to hardware design. This includes linting, formatting, dependency management, namespaces/scoping, and better support for large projects. Usually, alternative HDLs are transpiled to Verilog and synthesized with the normal Verilog toolchain for the target FPGA [Soz22].

2.4.1 *RustHDL*

RustHDL is a Rust-based HDL that allows describing RTL logic in Rust. The logic can then be transpiled to Verilog. RustHDL also includes tools for simulation and verification. Because a RustHDL-based design is just a Rust program, it can use most of the Rust ecosystem features. This includes the Rust testing framework for verification and simulation. It also makes it possible to easily reuse and share designs using the Rust package manager cargo. The Rust compiler can verify the validity of a design at compile time [Smi21].

In RustHDL, modules are defined as structs that implement the `LogicBlock` trait. The fields of these structs correspond to the external ports of the module. These ports are defined as `Signal` with a direction and a data type. If the module uses another module, it is also defined as a field. RustHDL provides basic modules for common functions like D flip-flops (DFF) or constants. It also contains a library of more complex modules like an i2c controller or memory blocks [Smi21].

Listing 3 shows the struct for a module that has the same functionality as the SystemVerilog blinker module in Listing 1.

Listing 3. Simple blinker module in RustHDL

```rust
#[derive(LogicBlock)]
pub struct Blinker {
    pub clock: Signal<In, Clock>,
    pub blinker: Signal<Out, bool>,
    blinker_state: DFF<bool>,
    counter: DFF<Bits<32>>,
    delay: Constant<Bits<32>>,
}
```

The struct only defines input and output signals and submodules of a module. The combinational logic that connects these signals is described in the update function. Every RustHDL signal has a `.next` field that determines the value of the signal after the update function. The update function must assign a value to all the `.next` fields. If there are multiple assignments to the .next fields, the last one is valid, like in normal Rust [Smi21].

Rust can express much more than just combinational logic, so RustHDL only allows a limited synthesizable subset of Rust inside the update function. Local variables are not allowed, and assignments can only be made to the `.next` fields of signals. Function and method calls are limited to a small subset of library functions, and only range-based for loops are allowed [Smi21].

RustHDL also enforces an additional set of semantic rules about what the update function has to do. For example, the update function must always assign a value to the `.next` field of signals declared as outputs. It also must always assign a value to the `.next` fields of the inputs of its submodules. This ensures that every signal always has a well-defined value. RustHDL also forbids to use of the `.next` value on the right side of an expression, so the result of the update function can only depend on the state from the previous cycle. RustHDL enforces these rules at run-time and generates somewhat helpful error messages. SystemVerilog does not apply these restrictions, which can be a source of bugs [Smi21]. The RustHDL update function for the blinker module is shown in Listing 4.

Listing 4. Testbench for the blinker module in RustHDL

```rust
impl Logic for Blinker {
    #[hdl_gen]
    fn update(&mut self) {
        // Connect the D flip-flops to the clock
        self.counter.clock.next = self.clock.val();
        self.blinker_state.clock.next = self.clock.val();

        // Connect the inputs of the D flip-flops to their own outputs
        // This makes sure they are always driven
        self.counter.d.next = self.counter.q.val();
        self.blinker_state.d.next = self.blinker_state.q.val();

        // Connect the blinker output to the state D flip-flop
        self.blinker.next = self.blinker_state.q.val();

        // Increment the counter
        self.counter.d.next = self.counter.q.val() + 1u64.to_bits();

        // Reset the counter when it reaches the delay
        if self.counter.q.val() == self.delay.val() - 1 {
            self.blinker_state.d.next = !self.blinker_state.q.val();
            self.counter.d.next = 0u64.to_bits();
        }
    }
}
```

The Rust unit testing framework can be used to simulate and verify RustHDL modules. Listing 5 shows a testbench for the blinker module. The testbench instantiates the blinker module and connects it to a clock signal. It also asserts that the blinker output toggles every ten clock cycles. The test case can be run with `cargo test` like any other Rust unit test.

Listing 5. Simulating and verifying the blinker

```rust
    #[test]
    fn blinker_works() {

        let blinker = Blinker::new(10);


        let mut simulation = Simulation::new();
        simulation.add_clock(1, |blinker: &mut Box<Blinker>| {
            blinker.clock.next = !blinker.clock.val()
        });


        simulation.add_testbench(move |mut sim: Sim<Blinker>| {
            let mut blinker = sim.init()?;

            for _cycle in 0..20 {
                sim_assert_eq!(sim, blinker.blinker.val(), false, blinker);
                wait_clock_cycles!(sim, clock, blinker, 10);
                sim_assert_eq!(sim, blinker.blinker.val(), true, blinker);
                wait_clock_cycles!(sim, clock, blinker, 10);
            }

            sim.done(blinker)
        });


        simulation
            .run_to_file(Box::new(blinker), 500, &vcd_path!("blinker_works.vcd"))
            .unwrap();
    }
```

The `generate_verilog` function can generate a Verilog description of the module, as shown in Listing 6. The generated Verilog code can then be used with any Verilog toolchain to deploy the design to an FPGA [Smi21].

Listing 6. Generating verilog from the RustHDL blinker

```rust
use blinker::Blinker;

fn main() {
    let mut device = Blinker::new(10);
    device.connect_all();
    let data = generate_verilog(&device);
    fs::write("./blinker.v", data).expect("Unable to write file");
}
```

RustHDL code is more verbose than a Verilog description, primarily because of the additional type annotations and the need to explicitly assign every signal's `.next` value. Rust macros can be used significantly reduce the amount of code. RustHDL includes macros for common tasks like setting up test benches or connecting clocks of submodules [Smi21]. To showcase RustHDL more explicitly, the above examples did not use these macros.

2.5 DESIGN USING HIGH-LEVEL SYNTHESIS

High-level synthesis (HLS) is a process that can generate an RTL specification of a circuit from a description in a high-level programming language. This is a more productive approach than writing RTL code by hand, but the resulting designs are usually less efficient. The generated RTL code can then be used in the same way as manually written RTL code [Mil20] [Nan16] [Lah19] [Cou09].

The main advantages of HLS are reduced design time and lower development cost. On average, the development time of HLS designs is only a third of that of equivalent RTL designs. The lower development time comes with the tradeoff that HLS designs, on average, take up around 40% more basic FPGA resources than RTL designs. The performance of HLS designs is around two-thirds of that of an RTL design. These metrics generalize over many different

HLS tools and input languages. In addition, there is a lot of variance between different studies. In about 40% of designs, the HLS design was actually more performant than the RTL design. In about 30%, it was more resource efficient. However, for development time, the results are conclusive; in 90% of the experiments, the HLS design required less development time than the RTL design [Lah19].

2.5.1 *How high-level synthesis works*

High-level synthesis tools generate a timed RTL implementation of a circuit from a functional specification. A functional specification is an untimed description of the desired behavior of the circuit [Cou09].

Usually, HLS tools separate HLS into seven tasks [Cou09]. They can be separated into three stages.

First, the functional specification is compiled into a formal model. The formal model is usually a control and data flow graph (CDFG). In the CDFG, every node (called a basic block) represents a static sequence of statements. The edges between the nodes represent conditional data flow. Opposed to a normal data flow graph (DFG), the edges can be conditional [Cou09]. In this step, transformations can be applied to the functional specification. These transformations can include loop unrolling, inlining, dead code elimination, and other common software compilation optimizations [Cou09].

The resulting formal model then has to be mapped onto hardware resources. The types of resources are functional units, storage elements, and connection units. HLS tools include RTL descriptions for all of their supported resource types. Functional units carry out the actual data processing. They can be multipliers, arithmetic logic units, or other custom functions. Storage elements store values over multiple cycles. They include registers, memories, or other custom storage elements. Connection units represent the connections between functional units and storage elements. This category includes resources like multiplexers and buses [Cou09].

Allocation determines the kinds and number of resources that are needed to satisfy the design constraints. Scheduling then schedules the operations into cycles based on the available resource types. Operations that have no data dependencies on each other can be scheduled in parallel. The next step after scheduling is called binding. During binding operations, get assigned to specific functional units. Variables that carry a value over multiple cycles must be bound to storage elements. Finally, connection units are assigned to specific connections between functional units and storage elements. Scheduling information can be used to bind multiple variables with non-overlapping lifetimes to the same storage elements [Cou09].

The final step of HLS is to generate the RTL architecture for the design. Classically the architecture consists of a controller and a datapath. The datapath consists of the allocated hardware resources. The controller is responsible for driving the datapath [Cou09].

The datapath contains all storage elements, functional units, and connection units. These components can be connected arbitrarily by buses. The datapath inputs are data inputs from external sources and control inputs from the controller. The outputs of the datapath can be data outputs or control outputs. In addition to the data inputs, the datapath also receives control signals from the controller. The control signals determine how the components are connected. They orchestrate the data flow through the datapath [Cou09].

The controller is usually a finite state machine (FSM) consisting of three parts. The state register contains the id of the current state. The output logic generates the control signals that drive the datapath based on the current state. The output logic also generates control outputs that can act as inputs for the datapath. The next state logic computes the next state of the FSM based on inputs and the current state [Cou09].

2.5.2 *High-level synthesis tools*

HLS tools can be distinguished into two major categories. Those that accept a general-purpose language and those that accept a domain-specific language (DSL) as input. Using a DSL as input can lead to better-performing designs, but it also raises challenges for adoption. A general-purpose language makes it easier for the algorithm designer, who is usually a software developer, to write code [Nan16]. The most common input languages for HLS are C based, including C, C++, and SystemC [Lah19].

HLS tools are usually available as a part of the IDE provided by the FPGA vendors. For example, AMD Xilinx provides the Vivado HLS tool as part of its IDE, and Intel Altera includes the Intel HLS Compiler in its Quartus Prime IDE. The most popular HLS tool in academia is Vivado HLS [Lah19]. There are also a few open-source HLS tools available. The most relevant open-source HLS tool that is currently actively maintained is PandA Bambu [Nan16].

2.6 LLVM

LLVM is a modular compiler framework that can be used to build compilers for many different programming languages. It defines a low-level code representation called LLVM intermediate representation (LLVM IR) [Lat04].

LLVM IR is a strongly typed, static single assignment (SSA) based intermediate representation. It is designed to be easy to compile to machine code and easy to optimize. The IR has a low-level, language-independent type system. This type system captures enough type information to safely perform a number of aggressive transformations that would traditionally be attempted only in source-level compilers of type-safe languages. LLVM IR is not intended to be a universal compiler IR, so it does not capture all of the language-specific type information. Some concepts not represented in LLVM IR are classes, inheritance, or exception-handling semantics. Language-specific transformations have to be performed by compilers before converting the code to LLVM IR [Lat04].

Because of this, compilers based on LLVM need to implement their own front end that processes input in their source language. The compiler front end emits LLVM IR, which is passed to the LLVM backend. The backend performs a variety of transformations and optimizations. The processed LLVM IR is then passed to a code generator backend which translates it into native code [Lat04]. The LLVM project includes code generator backends for many targets, including all common CPU architectures [Lat04] [Rus18].

2.7 SYSTEMS PROGRAMMING LANGUAGES

Systems programming languages are programming languages that can deal with low-level details of memory management, data representation, and concurrency. They are often designed for use in resource-constrained, performance-critical, or close-to-hardware programs. They are used to implement operating systems, embedded systems, device drivers, and other software that interacts with hardware. Common systems programming languages include C, C++, and Rust [Kla23] [Str12].

2.8 THE RUST PROGRAMMING LANGUAGE

Rust is a modern systems programming language aiming to replace C and C++ as the industry standard systems programming language. It offers zero-cost memory safety, a robust type system, and a standardized build system. Rust surpasses all other common memory-safe languages in terms of performance while offering many built-in features that more established systems programming languages tend to lack. For these reasons, it has been voted the most-loved programming language every year since 2016 [Bug22] [Cos19] [Kla23].

Rust overcomes the longstanding trade-off between the control over resource management of systems programming languages and the safety guarantees of higher-level languages. It is the first industry-supported programming language to do so [Bug22] [Jun17].

Rust achieves this by enforcing that every variable is always owned by a single owning scope. When that scope ends, the variable is destroyed. The time between creation and destruction of a variable is called its lifetime. Rust provides semantics for moving ownership between scopes [Kla23]. This model of scope-based resource management is called RAII. It is used by many systems programming languages, including C++. While C++ encourages the programmer to break out of that system by using pointers to variables that are not owned by the current scope or its parents, [Str12] Rust does not allow this.

Instead of pointers, Rust uses a type of reference with attached lifetime information called a borrow. The lifetime of a borrow ends when the lifetime of its owner ends. A borrow to a local variable of a function ends at the end of the function's scope. The borrow checker statically guarantees at compile-time that every borrow always points to a valid thing in memory. It does so by ensuring that the lifetime of every borrow accessible in a given scope ends at or after the lifetime of the current scope. Doing this also ensures that no borrows can be used in contexts that would outlive their scope. This makes it impossible for a reference to outlive the thing it points to [Kla23].

The compiler also makes sure that there exists only either one mutable borrow or multiple immutable borrows to a value at the same time. This ensures that the values of borrows do not change unexpectedly. It also enables some optimizations that would be difficult to perform otherwise. The programmer can opt out of the borrow checker by annotating code as `unsafe`. This allows the developer to use raw pointers and perform other inherently unsafe operations. Unsafe code is sometimes necessary to implement low-level data structures, such as smart pointers (`Box`, `Vec`, `String`) or types with internal mutability (`RefCell`). The standard library safely encapsulates these constructs, so most Rust programs do not need to use unsafe code [Kla23] [Bug22]. There is ongoing work on formally verifying that the unsafe code in the standard library is safely encapsulated by its types [Jun17].

Rust is a compiled language. The Rust compiler (rustc) can compile Rust code to native code for many different platforms. It accomplishes this by compiling Rust to LLVM IR and then using LLVM for code generation. This allows Rust to support many different platforms without having to implement a backend for every platform. It also enables Rust to utilize the large suite of advanced optimizations collected by the LLVM project. The compiler does not generate LLVM IR directly from the input Rust code. Instead, the input code gets passed through multiple intermediate representations (IRs); HIR, THIR, and MIR. HIR and THIR still resemble Rust code, but some constructs are desugared. The rust compiler uses these stages to perform type checking and verification. Verified THIR is converted into MIR, which is a control-flow graph (CFG) based representation of the code. rustc performs flow-sensitive safety checks like borrow-checking on this level. The MIR is also used to apply various Rust-specific optimizations to the code. The Rust compiler can be configured to output the various intermediate representations instead of generating machine code [Rus18].

An important part of what makes the people using the Rust ecosystem so productive is that Rust offers standardized tooling. Every Rust project (also called crate) contains a `Cargo.toml` file specifying the project metadata and dependencies. The `cargo` tool can perform all standard tasks like building, executing, unit testing, integration testing, formatting the code, generating documentation, downloading dependencies, building dependencies, managing dependencies, publishing the crate, benchmarking, setting up projects, and more. The official community crate registry crates.io can be used to find dependencies easily. It also links to the automatically generated documentation for every crate. The tooling alone makes Rust a much better development experience than most systems languages [Bug22] [Kla23].

2.9 PANDA BAMBU

Bambu is an open-source academic HLS tool [Fer21] [Nan16]. Its architecture is designed to make it easy to experiment with new ideas across high-level synthesis and related topics. It supports input specifications in standard C/C++ or the intermediate representations of GCC or Clang/LLVM [Fer21].

Bambu is based on a three-stage design. The front end is used to convert input specifications in various formats into a static single assignment (SSA) IR. The middle end performs various transformations and analyses on the SSA IR. The backend performs the actual architectural synthesis [Fer21].

The front end utilizes a custom GCC or Clang plugin to process an input specification in any of the formats supported by these two compilers. Notably, this includes support for LLVM IR through Clang. Bambu then generates its own SSA IR from the call graph and control flow information provided by GCC or Clang [Fer21].

The middle end applies a set of analyses and transformations to the specification. This includes common software compilation optimizations such as loop optimizations and dead code elimination. It also includes more HLS-specific optimizations. For example, multiplications and division with constants are replaced with shift and add operations because real multiplication is expensive in hardware. Bitwidth and range analysis optimizations are also performed to find the ideal width for the data path [Fer21].

The backend performs the actual architectural synthesis. This is mostly done as described in Section 2.5.1. The most significant difference in Bambu is that the synthesis process acts on every function individually. Each function gets its own controller and datapath. If it calls into another function, the generated logic for the other function is instantiated as a submodule. Bambu is able to optimize submodules that are shared between multiple modules. Bambu provides hardcoded optimized modules for functions from common libraries such as `libc` or `libm`. It also offers more advanced optimizations like pipelining individual functions. The resulting architecture can be translated to VHDL or Verilog [Fer21].

3 CONCEPT, IMPLEMENTATION, AND ARCHITECTURE

The goal of this paper is to use Rust as a source language for HLS using an existing HLS tool. First, a suitable toolchain is devised. Then a concept for integrating the toolchain with RustHDL is developed. Finally, a proof-of-concept implementation shows that the process works and enables the evaluation of the resulting solution.

3.1 BASIC TOOLCHAIN FOR SYNTHESIZING RUST

As no tool can currently synthesize Rust directly, the toolchain needs to convert the Rust code into a language that can be used by an existing HLS tool. The Rust compiler can be used to generate LLVM IR instead of machine code, as its backend is based on LLVM. It is frequently updated, and the generated LLVM IR usually uses the latest LLVM version. This means that a suited HLS tool also needs to be actively maintained to support the generated LLVM IR. The HLS tool should also be open-source, as it could become necessary to adjust the tool for Rust support.

Three HLS tools are based on LLVM IR, namely PandA Bambu, SmartHLS (formerly known as LegUp), and Vivado HLS. All of them seem to use LLVM primarily to process C and C++ input. PandA Bambu is the only one of these three tools that is open-source and actively maintained [Nan16].

In the first step, the Rust compiler is used to convert a Rust function to an LLVM IR function. The second step passes the generated LLVM IR function to Bambu, which converts it to Verilog. Figure 1 depicts that toolchain.

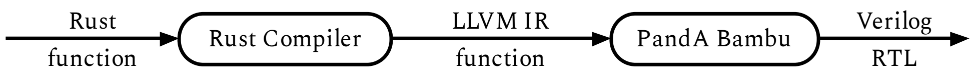


Figure 1. A toolchain for high-level synthesis from Rust

3.1.1 *Toolchain proof of concept*

This section will demonstrate how the toolchain can be used to synthesize Rust functions and which additional restrictions apply.

The `minmax` function shown in Listing 7 finds the minimum and maximum in an array of integers. The implementation is kept in a C-like style so that it can be compared to an algorithmically identical C++ implementation later on. The function takes a pointer to an array of integers (`numbers`), the number of elements in the array (`numbers_length`), and two pointers to integers (`out_max` and `out_min`). It finds the smallest and largest value of the array and writes them to the memory locations pointed to by `out_max` and `out_min`.

Listing 7. `minmax` *implementation in Rust*

```rust
/// Minmax function that is as similar as possible to the equivalent cpp function
#[no_mangle]
pub unsafe extern "C" fn minmax(
    numbers: *mut i32,
    numbers_length: i32,
    out_max: &mut i32,
    out_min: &mut i32,
) {
    let mut local_max = i32::MIN;
    let mut local_min = i32::MAX;
    for i in 0..numbers_length {
        if *numbers.offset(i as isize) > local_max {
            local_max = *numbers.offset(i as isize);
        }
        if *numbers.offset(i as isize) < local_min {
            local_min = *numbers.offset(i as isize);
        }
    }
    *out_max = local_max;
    *out_min = local_min;
}
```

The function needs to be annotated with `#[no_mangle]` to instruct the Rust compiler to preserve the function name in LLVM IR. This is required because Bambu uses the function name to generate the name of the Verilog module. The function is also marked as `extern "C"` to instruct the Rust compiler to generate LLVM IR that is compatible with the standard C ABI. `extern "C"` functions only allow data types in their signatures that have C representations. This does not include Rust slices and tuples. While these things can be represented in LLVM IR, Bambu only supports C-compatible function signatures. This is the reason for the arguments of the function containing a pointer and its length instead of a slice, which is basically the same thing. Finally, the interface of the function is marked as `unsafe` because it uses raw pointers, which are inherently unsafe.

The algorithm performed by the function is simple. The local variables `local_min` and `local_max` store the current minimum and maximum values. They are initialized to the smallest and largest 32-bit integer, respectively. The function then iterates over the memory area containing the input numbers. For each input number, the local minimum or maximum values is updated if the current number is smaller or larger. Finally, the local minimum and maximum values are written to the output memory locations `out_max` and `out_min`.

Listing 8. Compiling Rust to LLVM IR with the flags for HLS

```shell
rustc \
  src/minmax.rs -o minmax.ll \
  --crate-type=lib \ # The output is a library
  --emit=llvm-ir \ # Emit LLVM IR instead of machine code
  -C llvm-args=--opaque-pointers=false \ # Disable opaque pointers
  -C no-vectorize-loops \
  -C panic=abort \
  -C overflow-checks=off \
  -C opt-level=3 \
  -C linker-plugin-lto=on \
  -C embed-bitcode=on \
  -C lto=fat
```

The first step is to compile the function to LLVM IR using the Rust compiler. The command shown in Listing 8 contains the options that are required or recommended to generate LLVM IR that can be used by Bambu.

`src/keccak.rs -o keccak.ll` specifies the filenames. The following argument, `--crate-type=lib`, tells the Rust compiler that it should generate a library. The most important option is `--emit=llvm-ir`, which tells the compiler to emit LLVM IR instead of machine code.

Recent versions of the Rust compiler will generate LLVM IR with opaque pointers(`ptr` instead of `i32*`) by default. Bambu does not support this. Opaque pointers can be turned off by passing `-C llvm-args="--opaque-pointers=false"` to rustc.

In Rust, panicking is a core part of the language. Panics are caused when unrecoverable errors occur. The Rust compiler generates code that unwinds the stack on panic by default. Bambu can currently not synthesize this behavior. The `-C panic=abort` instructs rustc to terminate the program on panic. This is also not synthesizable, but Bambu seems to compile the code if the panics are unreachable. It may be possible to use the code anyway by linking a dummy implementation for the panic function, but this would result in undefined behavior.

The `-C no-vectorize-loops` option turns off loop vectorization. This prevents rustc from generating vector instructions, which are currently not supported by Bambu.

By default, integer overflows are forbidden in Rust and will result in a panic. Disabling overflow checks for arithmetic operations reduces the number of places where a function can cause a panic significantly. This is safe because integer overflow is well-defined in Rust. The `-C overflow-checks=off` option disables overflow checks. The Rust compiler can also generate additional debug assertions, which can make finding and fixing bugs easier. Failing these assertions manifests in a panic at runtime. To avoid this, the `-C debug-`

`assertions=off` option can be used to disable debug assertions. On every optimization level other than `0`, they are automatically disabled. This option is not required in the example.

The final set of parameters enables various Rust compiler optimizations. They serve three purposes. First, the Rust compiler can perform more optimizations than Bambu because it has more information about the code. The second purpose is to reduce the amount of code that Bambu has to synthesize. The third and probably most crucial purpose is eliminating dead code that could cause panic. For this reason, it is basically always required to set the optimization level to a level higher than `1`.

Listing 9. High-level synthesize LLVM IR into Verilog

SHELL

```
bambu minmax.ll \
  --compiler=I386_CLANG16 \
  --top-fname=minmax \
  -O5
```

The LLVM IR can then be used with Bambu to generate Verilog. The command shown in Listing 9 instructs Bambu to generate a Verilog module named `minmax` from the LLVM IR file `minmax.ll`.

The `--compiler=I386_CLANG16` flag instructs Bambu to its Clang 16 front end for processing input. The `--top-fname=minmax` option specifies the name of the top-level function. The top-level function will be used as the entry point in the hardware design, comparable to the `main` function in a C program. `--top-fname` also sets the name of the generated Verilog module. The `-O5` option instructs Bambu to perform optimizations that increase the performance at the cost of a bigger design.

3.1.2 *Interface of the generated Verilog module*

The interface of the generated `minmax` module is shown in Figure 2. It can be split into three categories. The first section contains clocking and control signals, including `clock`, `reset`, `start_port`, and `done_port`. It will also contain a `return_port` if the function has a return type. Table 1 describes what each of these signals does.

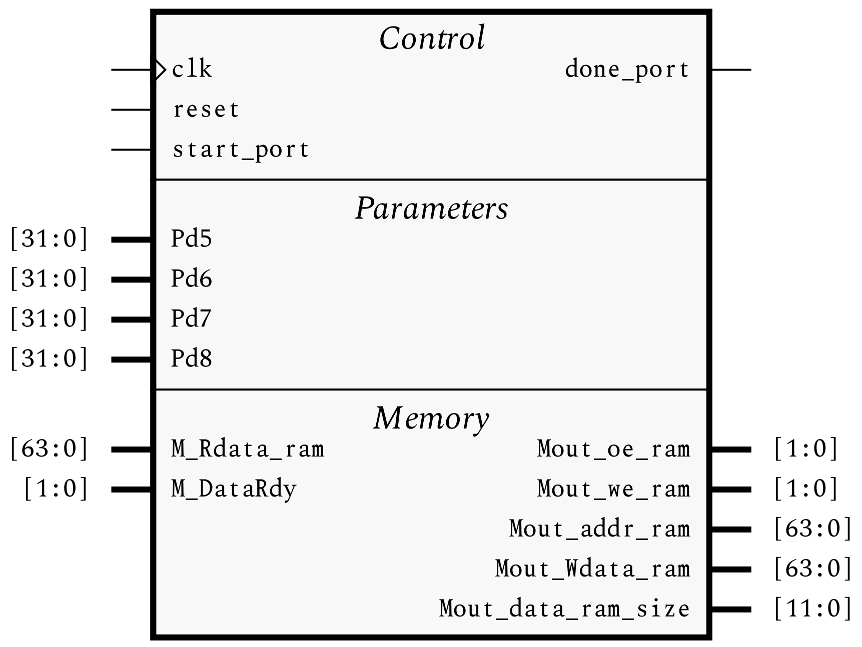


Figure 2. Interface of the synthesized module

Table 1. The control signals of modules generated by Bambu

Port	**Description**
clock	The clock signal is used to clock the module.
`reset`f	Resets the module if set to low.
start_port	The module will start executing if this is high and the function is not already running. Pinning this to high will cause the function to repeat.
done_port	Pulses high for one cycle when the module is done.
return_port	Does only exist if the function has a return value. Contains the return value while done_port is high. It can contain random values during function execution.

The second section contains the function parameters. When LLVM IR is used as input for Bambu, the real names of the function parameters get lost, they are replaced with numbered names like `Pd5`. The order of the signals stays the same as the order of the inputs to the original function. Bambu always numbers the parameters in order of appearance in the source. The original names can be mapped to the numbered names based on their order. In this case, `Pd5` is `numbers_length`, `Pd6` is `numbers`, `Pd7` is `out_max`, and `Pd8` is `out_min`. Memory pointers got converted to 32-bit values, but Bambu can also be configured to generate other address sizes.

The original function takes a pointer to memory as input, so the generated module needs to be able to access that memory. Bambu generates a memory interface for the module, which needs to be connected to memory. During function execution, the module will use this memory interface to retrieve and modify the input values. Because Bambu generates a two-channel memory interface by default, the memory signals are twice the size expected. For all generated modules in the remainder of this paper, Bambu will be configured to generate only a single channel. The address size is 32-bit without additional configuration, and the data width gets automatically selected based on the design. In this case, the data width is 32-bit. Because the module has two channels, all memory signals are double the size. Their first half corresponds to the first channel, and their second half to the second channel. Table 2 describes the signals of the memory interface in more detail.

Table 2. The memory interface of modules generated by Bambu

Port	**Size per channel in bits**	**Description**
`Mout_oe_ram`	1	Set to 1 to read from the channel.
`Mout_we_ram`	1	Set to 1 to write to the channel.
`Mout_data_ram_size`	$\log_2(\text{dataWidth}) + 1$	Set the width of bits that should be written to the memory. It can be a value between 0 and the width of your data.
`Mout_addr_ram`	addressWidth	Select the address this channel should operate on.
`M_Wdata_ram`	dataWidth	Contains the data that will be written to memory if `Mout_we_ram` is set.
`M_Rdata_ram`	dataWidth	Contains the data that was read from memory if `Mout_oe_ram` was set in the last cycle.
`M_DataRdy`	1	Nonzero if the memory is not ready.

3.2 INTEGRATING THE TOOLCHAIN WITH RUSTHDL

As shown in the previous section, using the toolchain requires a few manual steps. Especially the numbered parameter names are challenging to work with because they change after every modification to the input. RustHLS, a library for integrating the toolchain with RustHDL, was created to solve this issue. It tries to make the toolchain easier to use. The library makes it possible to create a single Rust crate containing both RTL and HLS descriptions.

Cargo provides a way to hook into the build process using build scripts. These are small Rust programs that can modify the Source code before it is passed to the Rust compiler. The Rust compiler also allows custom code transformations at compile-time using procedural macros. Procedural macros are Rust functions that run at compile-time and can modify the code they are applied to.

RustHLS provides a `#[hls]` macro that can be used to mark Rust modules for HLS. The macro is evaluated two times. Before compilation, a build script finds all modules marked with `#[hls]`. It extracts every marked module into a separate temporary crate. These crates are synthesized to Verilog using an automated version of the toolchain described in Section 3.1. A Rust module containing a RustHDL struct that wraps the synthesized Verilog is generated for each Verilog file. That Rust module is then placed alongside the original source code. During compilation, the `#[hls]` macro is evaluated by the Rust compiler. The macro then emits import directives for the module that were synthesized during the build script. It also evaluates if the macro is used in a valid context and emits helpful error messages if it is not. Figure 3 shows all steps performed by RustHLS.

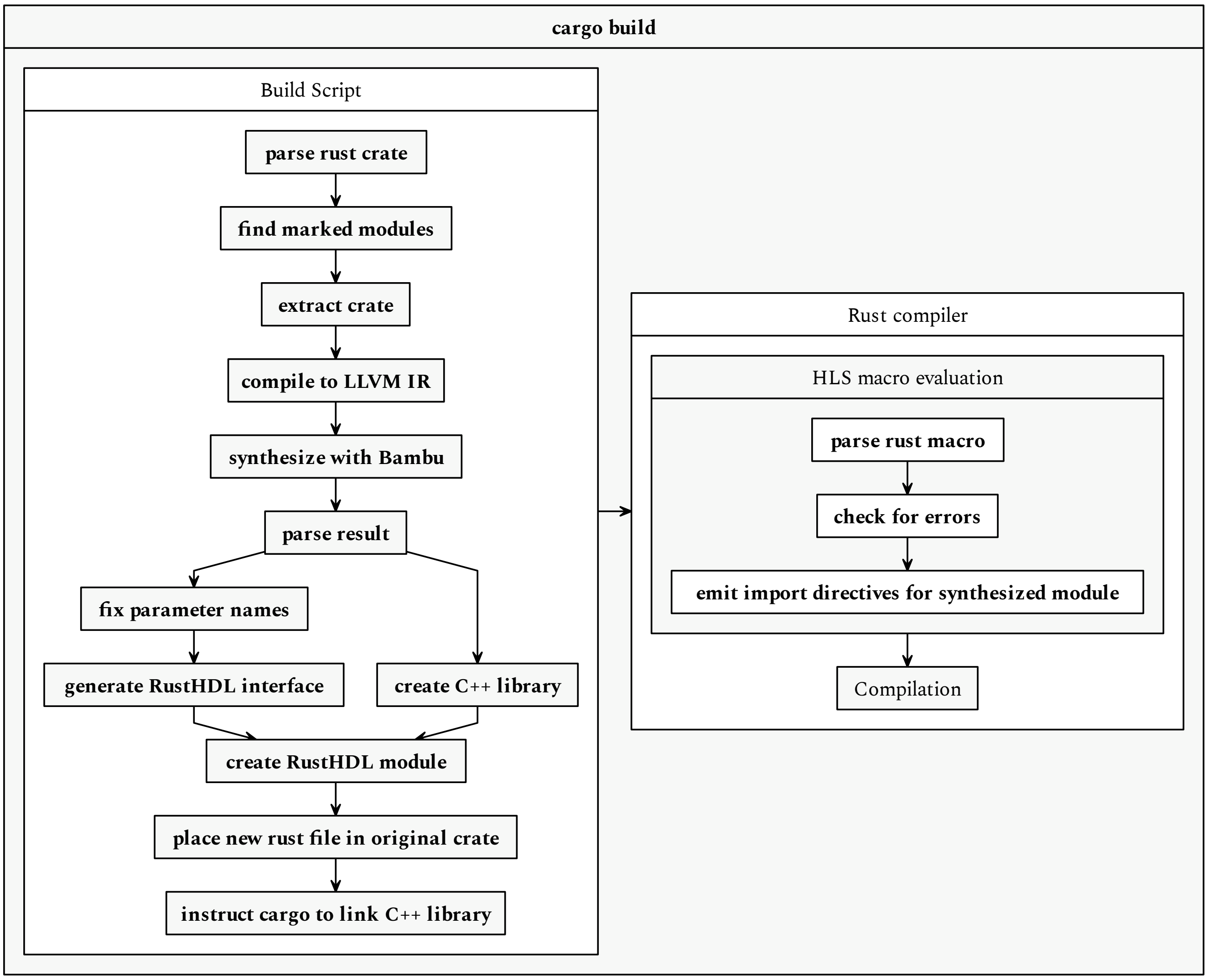


Figure 3. The steps performed by the RustHLS

RustHDL is not able to simulate embedded Verilog. Verilator, a SystemVerilog simulator, can create a C++ library from Verilog that simulates a single component. For every synthesized module, such a library is created. The RustHDL structs generated by RustHLS contain the glue code necessary to call into the simulation library. The build script emits the directives necessary to compile and link the generated libraries. Cargo will follow these directives to build and link the libraries into the final binary.

Using RustHLS makes it possible to use the toolchain in a RustHDL project. The modules can be used, simulated, and tested like regular RustHDL modules. This enables behavioral verification directly in Rust unit tests.

As part of this thesis, an integration of the toolchain with RustHDL was developed. It enables the use of the toolchain in a RustHDL project. The integration is implemented as a rust library, RustHLS. It provides a `#[hls]` macro that can be used to mark Rust modules for HLS. Listing 10 shows how it can be used to annotate a module for HLS.

Listing 10. Using RustHLS for synthesis of a Rust function

```rust
use rust_hdl::prelude::*;
use rust_hls_macro::hls;

#[rust_hls_macro::hls]
pub mod minmax_hls {

    #[hls(
        bambu_flag = "--channels-type=MEM_ACC_11 --channels-number=1 -Os",
        rust_flag = "-C opt-level=z"
    )]
    pub unsafe extern "C" fn minmax(
        elements: *const i32,
        num_elements: i32,
    ) -> rust_minmax::MinmaxResult {
      let slice = std::slice::from_raw_parts(numbers, numbers_length as usize);

      slice.iter().fold(
          MinmaxResult {
              max: i32::MIN,
              min: i32::MAX,
          },
          |mut acc, &x| {
              if x > acc.max {
                  acc.max = x;
              }
              if x < acc.min {
                  acc.min = x;
              }
              acc
          },
      )
    }

    #[repr(C)]
    pub struct MinmaxResult {
        pub max: i32,
        pub min: i32,
    }

}
```

Lines 8 and 9 show how the `#[hls]` how the macro can be used to configure rustc and Bambu for this specific function. The example above will result in a `minmax_hls_synthesized.rs` file placed beside the source file. It will contain the synthesized RustHDL module. A shortened version of this file is shown in Listing 22. It defines a RustHDL struct that wraps the Verilog generated by Bambu. The struct also contains an update function that calls the simulation library generated by Verilator. When the HLS macro gets expanded during compilation, it will emit the import statements necessary to use the synthesized module.

Rust's integrated testing framework can be used to test both the specification and the synthesized design. Listing 11 shows how to do that. The tests can be run with `cargo test` like any other Rust tests.

Listing 11. Verifying the synthesized module with RustHDL

```rust
#[cfg(test)]
mod tests {
    /// The original specification
    use super::minmax_hls::{minmax, MinmaxResult};
    /// The generated component
    use super::Minmax;

    const TEST_INPUT: &[i32; 25] = [
            0x21258F79, 0x34D5CCF9, 0x3598261E, 0x7D154730, 0x5B284E05,
            0x5F97A42D, 0x10FEE5A0, 0x2C5BDA0C, 0x4D30A6F7, 0x30935AB7,
            0x4B5AA93F, 0x49A6E626, 0x61A57C16, 0x43B831CD, 0x01F22F1A,
            0x05E5635A, 0x64BEFEF2, 0x61367095, 0x787C5A55, 0x3C3EE88A,
            0x040C7922, 0x1841F924, 0x16F53526, 0x75F644E9, 0x3AF1FF7B,
        ];

    #[test]
    fn verify_behavioural_specification() {
        let result = unsafe { minmax(&numbers[0], 25) };

        assert_eq!(result.max, 0x7D154730);
        assert_eq!(result.min, 0x01F22F1A);
    }

    #[test]
    fn verify_hls_design() {
        /// The hls_test macro is provided by the RustHLS library
        /// It will generate a simple testbench for the module that runs until done_port goes high
        /// It will also connect the module to a memory containing the test input
        hls_test!(
            Minmax,
            minmax,
            memory = TEST_INPUT.clone(),
            u32,
            {
                /// Setup
                minmax.elements.next = 0u32.to_bits();
                minmax.num_elements.next = input_length.to_bits();
            },
            {
                /// Verification
                // println!("Memory: {:X?}", memory);
                let result = minmax.return_port.val().to_u64()
                let result = unsafe {
                    std::mem::transmute::<_, MinmaxResult>(result);
                };
                assert_eq!(result.min, 0x7D154730);
                assert_eq!(result.max, 0x01F22F1A);
            },
            /// , "trace_test.vcd"
        );
    }
}
```

The first test verifies that the specification is correct. The second test verifies that the generated design is correct.

The `hls_test!` macro is provided by the RustHLS library. It will expand to a test bench that runs the module until it indicates it is finished. It will also connect the module to a memory containing the test input. A regular software debugger can debug the design like in any other Rust program. Alternatively, the macro also provides a way to store a trace of a test run by specifying a filename. This trace can be used to debug the design. Of course, the module can also be tested without the macro, but mocking the memory takes about 20 lines of code, so the macro is a lot more convenient. If the module does not need memory, then testing the module is the same as testing any other RustHDL module.

In this example, the design is verified against predefined constants. It is also possible to call the specification and assert that the result is the same as the result of the design. This can be used to verify that the design is equivalent to the specification.

RustHLS was used to synthesize and run all tests in Chapter 4.

4 EXPERIMENTS AND RESULTS

A set of algorithms will be compared to evaluate how HLS from Rust compares to HLS from C++. Every algorithm will have a *C++* and equivalent Rust implementation. The equivalent *Rust* implementation is algorithmically as close to the C++ implementation as possible. A more Rust-like implementation, using Rust features like slices or iterators, is evaluated for some algorithms. It will be referred to as the *idiomatic Rust* implementation. The goal of the equivalent Rust implementation is to compare the performance of the toolchain and compiler. Comparing the idiomatic Rust implementation can show whether the toolchain is able to synthesize idiomatic Rust code. An implementation using a dependency from the Rust community crate registry crates.io is also compared for one algorithm. This shows that it is possible to use already existing crates for HLS. It will be referred to as the *crates.io Rust* implementation.

Each C++ implementation was synthesized and tested under four configurations: compiled with either Bambu's GCC (Version 8) or Clang (Version 16) frontend and optimized for either speed (`-O5`) or size (`-Os`). Rust implementations were synthesized and tested using RustHLS as described in Section 3.3. When optimizing a Rust implementation for speed, the Rust compiler was set to `-C opt-level=3` and Bambu to `-O5` . When optimizing for size, the Rust compiler was set to `-C opt-level=z` and Bambu to `-Os` .

Different designs were evaluated for the area of an FPGA they take up, their clock maximum frequency, and the number of clock cycles they take for a single function execution. Their average was used if a design had varying clock cycles between its test cases. The area and maximum frequency of a design were taken from the placement report of nextpnr. nextpnr was configured to target a Lattice ECP5 FPGA with 44k LUT and a speed grade of 6 (LFE5U-45F-6BG381C) for all tests. Bambu was configured to generate only a single memory channel memory interface instead of the default two-channel memory interface. This makes it easier to compare the traces of different designs. The sum of the number of LUTs, FFs, BRAMs, and DSP blocks was used to measure the area of a design. The number of clock cycles was measured using the simulations generated with RustHLS for Rust designs and Bambu's built-in testbench generator for C++ designs. The results of these two ways of simulating the designs are comparable as both internally use Verilator to generate the actual simulation for the designs.

The evaluation focuses mostly on the behavior of the synthesized hardware designs and not on their exact implementation details. Because of this, the synthesized Verilog files will be treated as black boxes.

4.1 TESTED ALGORITHMS

The three functions that were compared are minmax, KECCAK-$f[1600]$, and MD5.

The minmax function finds the minimum and maximum values in an array of signed 32-bit integers. This algorithm is chosen because it is simple and can be implemented in a single short function. This should make it easier to understand the differences between the different designs.

To assess the toolchain's optimization capabilities, the KECCAK-f[1600] function was chosen. This cryptographic function serves as the foundation for SHA-3 and is widely employed in various applications. The implementations take a pointer to 25 64-bit numbers and perform in-place permutations on these numbers. If a design performs all permutations directly on the input array, it will take many cycles to complete because every memory access takes one cycle.

The MD5 hash function was selected as an additional algorithm for evaluation. It is similar to KECCAK in that it contains a long sequence of non-branching logic operations while accessing an array of constants. MD5 does not permute its inputs but instead generates a 128-bit state based on the previous state and a 512-bit data block. It is well known that MD5 suffers from extensive vulnerabilities. The tested implementation takes two pointers as input, one to the previous state and one to the data block.

4.2 RESULTS FOR THE MINMAX DESIGNS

Three implementations of a minmax function were tested. The Rust implementation was already shown above in Listing 7. It is based on the C++ implementation shown in Listing 15. The idiomatic Rust implementation is shown in Listing 12.

Listing 12. Idiomatic `minmax` *implementation in Rust*

```rust
#[repr(C)]
pub struct MinMaxResult {
    pub max: i32,
    pub min: i32,
}

pub unsafe extern "C" fn minmax_idiomatic(
    numbers: *const i32,
    numbers_length: i32,
) -> MinMaxResult {
    let slice = std::slice::from_raw_parts(numbers, numbers_length as usize);

    slice.iter().fold(
        MinMaxResult {
            max: i32::MIN,
            min: i32::MAX,
        },
        |mut acc, &x| {
            if x > acc.max {
                acc.max = x;
            }
            if x < acc.min {
                acc.min = x;
            }
            acc
        },
    )
}
```

The Rust compiler should be able to compile the idiomatic Rust implementation to nearly the same LLVM IR as the equivalent Rust implementation. The most significant difference is that the idiomatic function returns a struct instead of writing its result to memory. The expectation is that the idiomatic Rust design always performs better than the C-like Rust design, as it does not have to access memory for writing the result. It has a `return_port` that contains the result.

Figure 4 shows that The Rust, idiomatic Rust, and Clang designs optimized for speed occupy approximately three times as much space as the size-optimized designs. When using the same optimization settings, all designs relying on LLVM toolchains require a similar amount of space. However, the GCC designs deviate from this trend. The speed-optimized design from GCC is only 1.2 times larger than its size-optimized design. The size-optimized design is also the smallest design overall. Interestingly, even the GCC design optimized for speed ranks as the third smallest design. Moreover, both designs based on the idiomatic Rust implementation occupy slightly less space than their counterparts using the C-like Rust implementation.

LLVM may incorporate optimizations that result in a notable increase in the required area when prioritizing speed. In contrast, GCC does not seem to employ such optimizations.

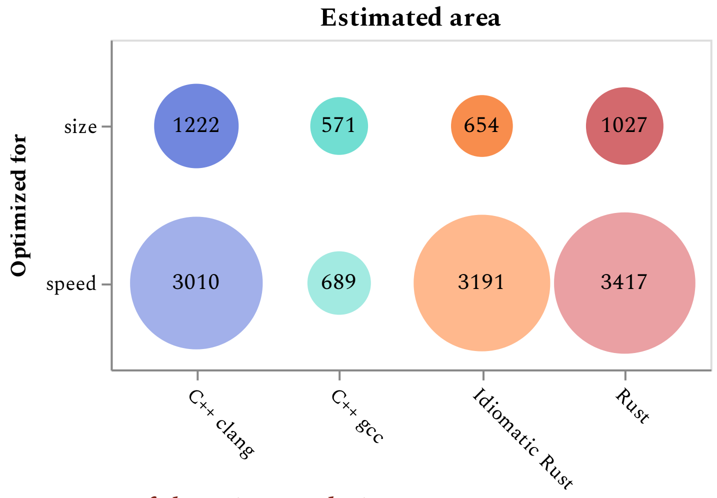


Figure 4. Area of the minmax designs

Each design underwent testing using test cases ranging from 0 to 50 elements. Figure 5 illustrates each design's average number of cycles.

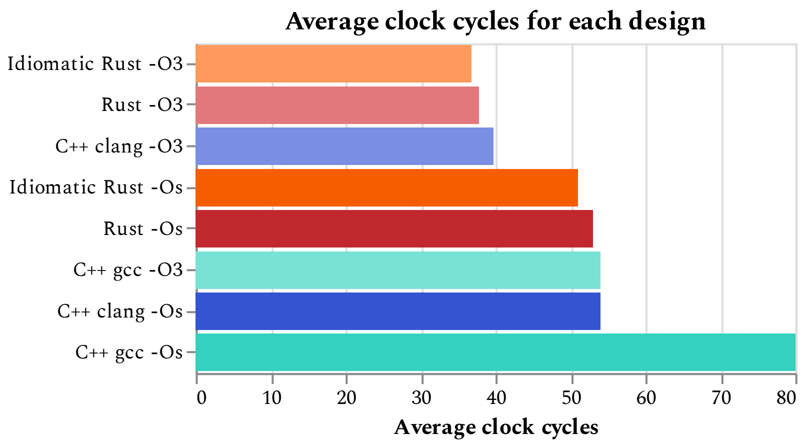


Figure 5. Average clock cycles of the minmax designs

Three distinct performance classes can be observed. The first class comprises the LLVM designs optimized for speed, the second class includes the LLVM designs optimized for size and the GCC design optimized for speed, and the slowest class encompasses the GCC design optimized for size.

Within the two fastest classes, their respective idiomatic Rust design stands out as the fastest, followed by the equivalent Rust design. The C++ designs come after them.

Notably, the fastest class exclusively consists of the big designs, whereas the smaller designs occupy the slower two classes. This observation suggests that the LLVM speed optimizations have indeed increased performance at the expense of area.

The GCC designs are both significantly slower. Each of them needs approximately 30% more cycles than the LLVM designs with the same optimization goal.

Figure 6 provides precise cycle counts for each test case, illustrating the three distinct performance classes. The line for `C gcc -03` overlaps with the line for `C clang -0s`

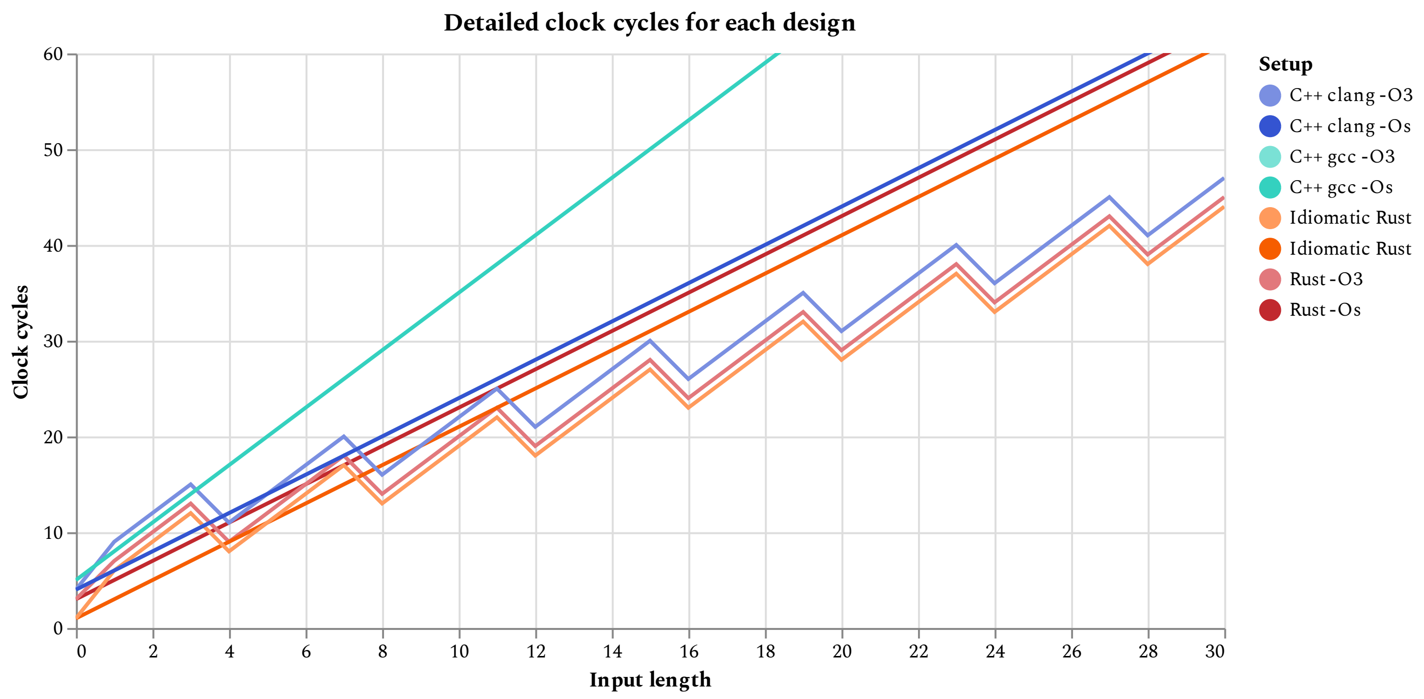


Figure 6. Clock cycles vs input length of the minmax designs

The slowest class takes 3 cycles per additional input. The second fastest class takes 2 cycles per additional input.

Interestingly the fastest designs require three cycles per input, except for every fourth input, where it requires three cycles less. This averages out to 1.5 cycles per additional input. It should be noted again that these designs are also the largest in size.

The LLVM designs optimized for speed have evidently unrolled the loop by a factor of four. This can be seen in the LLVM IR control-flow graph (CFG) depicted in Figure 26. The functions process a batch of 4 elements until less than four elements remain. Then they process the remaining elements individually. Bambu seems to preserve this structure in the generated designs. It explains why the big designs are also the fastest ones, as the additional size most likely contains the 4x operation. The small designs save area by only having the 1x operation.

The idiomatic Rust design is consistently the fastest design taking one cycle less than the next fastest design. It probably has this constant advantage because it does not have to write its results to memory after processing all elements. The C-like Rust designs are two cycles faster than the equivalent C++ designs compiled with Clang.

The GCC designs are all slower than the LLVM designs, potentially representing a trade-off for their reduced design size.

Another factor for design performance is the maximum frequency at which it can operate. Figure 7 provides an overview of the maximum frequency for each design.

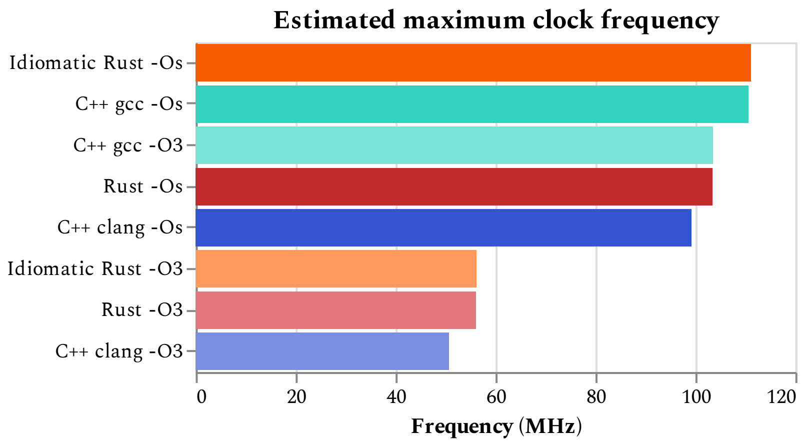


Figure 7. Maximum frequency of the minmax designs

Most notably, the LLVM designs optimized for speed are the ones that clock the slowest. The two Rust designs optimized for speed can clock slightly higher than the C++/Clang design optimized for speed. This is to be expected as they are also the designs that take up the most space, which most likely results in longer critical paths.

The five smaller designs all clock around 2x faster than the big designs. Their maximum frequency seems to be roughly ordered by the size of the design.

The small designs can run at much higher frequencies but require more clock cycles. The relevant metric for comparing the performance is the execution time. It is calculated by multiplying the number of cycles with the inverse of the maximum frequency. Figure 8 shows the average execution time for each design.

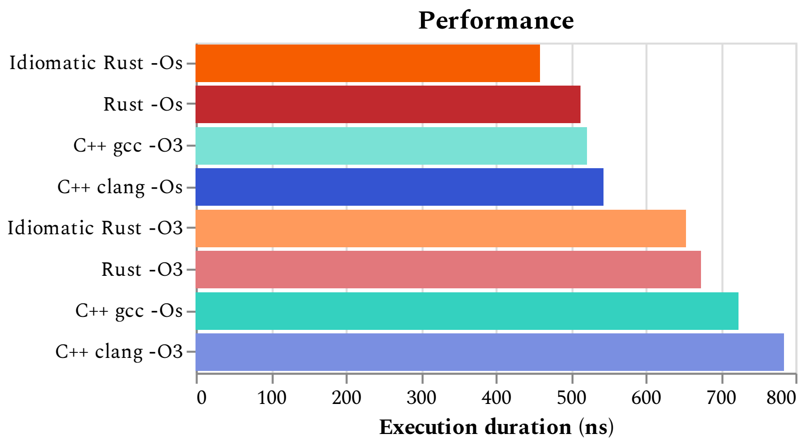


Figure 8. Average execution of the minmax designs

The LLVM designs optimized for speed perform between 25-30% better than those optimized for size. They still seem to split into two groups, although the divide is not as big as with the other measurements.

As expected, the significantly slower frequency of the big designs is more than enough to cancel out the advantage they gained by requiring fewer cycles.

The GCC designs clocked at nearly the same frequency. The lower cycle count of the design optimized for speed makes it perform better than the one optimized for size. It performs comparably to the LLVM-based designs optimized for size, while the one optimized for size performs comparably to the LLVM-based designs optimized for speed.

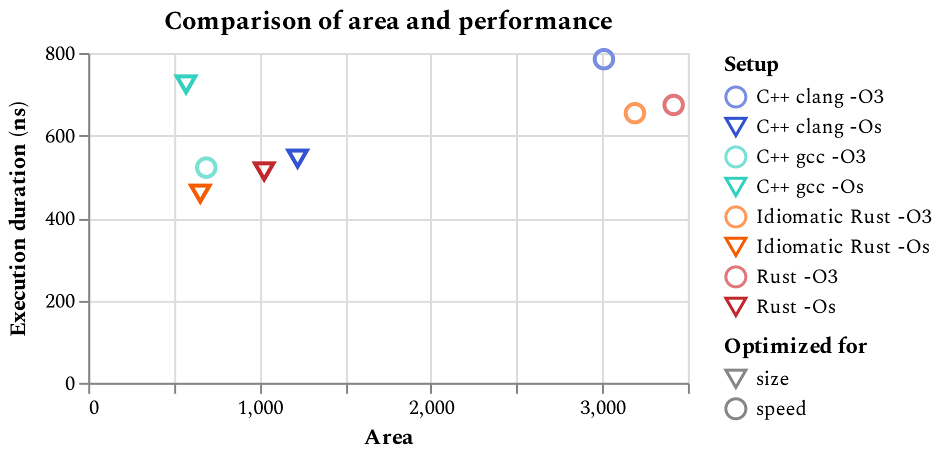


Figure 9. Area to performance of the minmax designs

The big designs perform worse and take up more area than the small designs. The idiomatic Rust design (size) is better in size and performance than nearly every other design. The only exception is the GCC design optimized for size, which is slightly smaller but performs significantly worse. These results can be seen in Figure 9 and Figure 10.

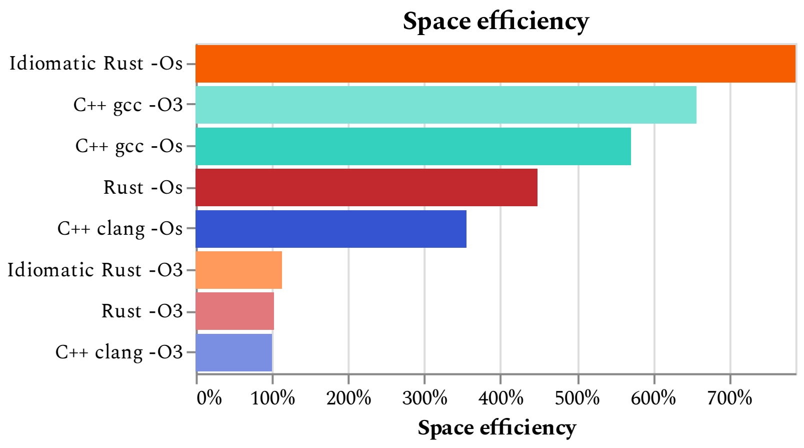


Figure 10. Space efficiency of the minmax designs

The idiomatic Rust design (size) is the most space-efficient design. The two GCC designs are second and third. The GCC design optimized for speed is a bit faster but also a bit bigger than the one optimized for size. This trade-off equalizes again in this metric, as their performance per area is nearly identical.

The big designs perform poorly in this metric because they are slower and take up more area than the small designs.

These measurements reveal several observations and trends:

- All designs generated by LLVM-based toolchains with the same optimization goal exhibit similar performance.
- The small designs perform better across all metrics except cycle count.
- Rust designs demonstrate slightly better performance.
- The idiomatic Rust design holds a slight advantage over the equivalent Rust design. Its slightly different interface probably causes this.
- For GCC, the optimization goal aligns with the resulting design benefits. The design optimized for code size is smaller, while the design optimized for speed performs faster.
- For LLVM, this alignment is only partially true. The designs optimized for size are smaller and faster than those optimized for speed.
- LLVM seems to employ more aggressive loop unrolling compared to GCC.
- Bambu appears to preserve unrolled loops in LLVM IR.

4.3 RESULTS FOR THE MD5 FUNCTIONS

Two implementations will be tested for the MD5 function. The C++ reference implementation is shown in Listing 16. The equivalent Rust implementation can be seen in Listing 17.

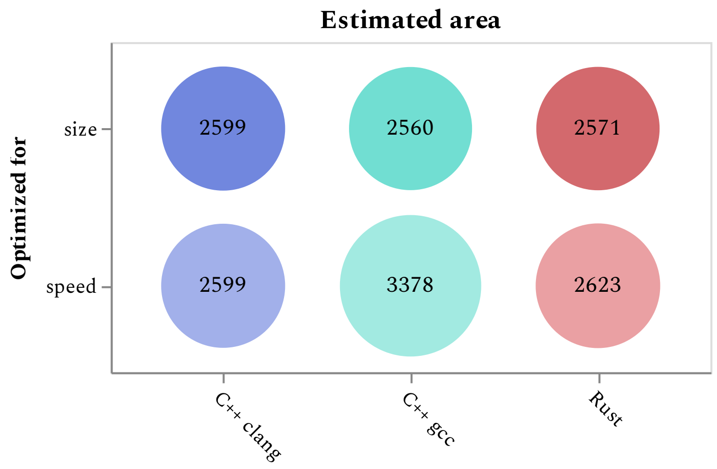


Figure 11. Area of the MD5 designs

As depicted in Figure 11, the area for all designs except the GCC design optimized for speed is roughly the same. The GCC design optimized for speed is 20% larger than the others. This seems to indicate that the designs generated by Bambu are all quite similar.

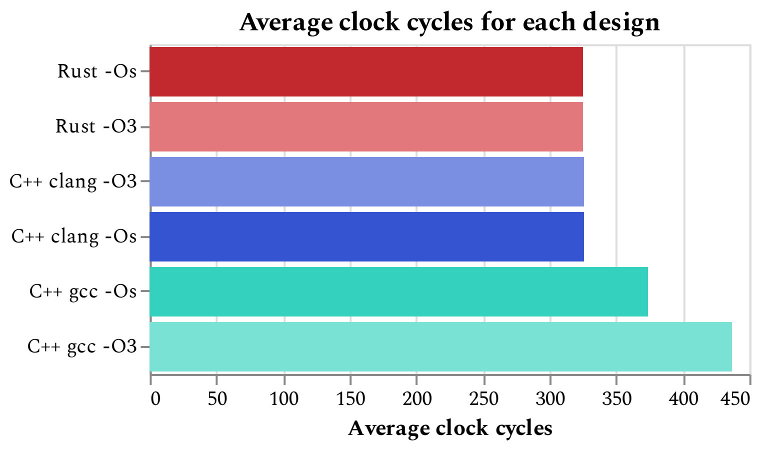


Figure 12. Clock cycles of the MD5 designs

Figure 12 shows that the LLVM-based designs all require the same number of cycles. The GCC design optimized for speed needs around 10% more cycles, and the GCC design optimized for size 20% more.

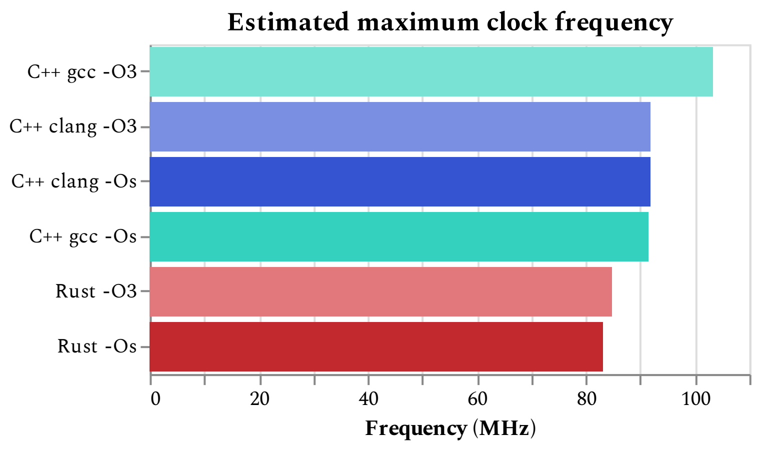


Figure 13. Maximum frequency of the MD5 designs

The maximum frequency of the Clang-based designs is about 10% higher than the Rust-based designs, as seen in Figure 13. The GCC design optimized for speed has the highest frequency, while the GCC design optimized for size is comparable to the Clang designs.

Figure 14. Execution time of the MD5 designs

Figure 14 shows the execution time for each design. The execution time of the Clang-based designs is about 10% lower than the Rust-based designs. As their number of cycles is equal, this represents their slightly higher frequency. The GCC designs are the slowest, although they are only around 15% slower than the Clang designs. The GCC design optimized for speed is the slowest design overall.

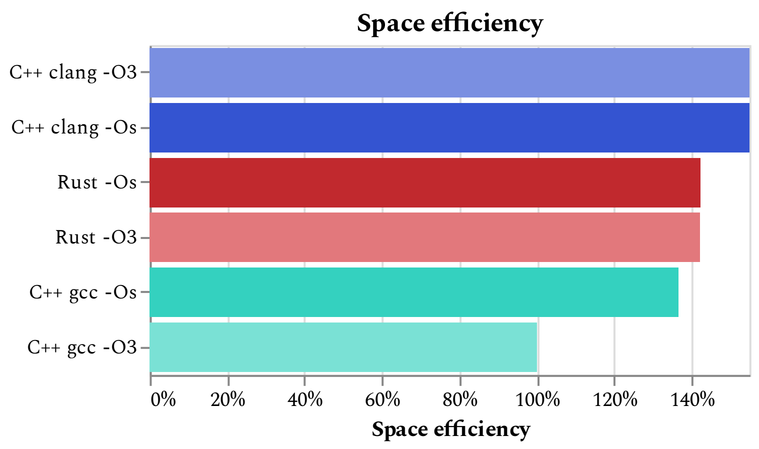


Figure 15. Space efficiency of the MD5 designs

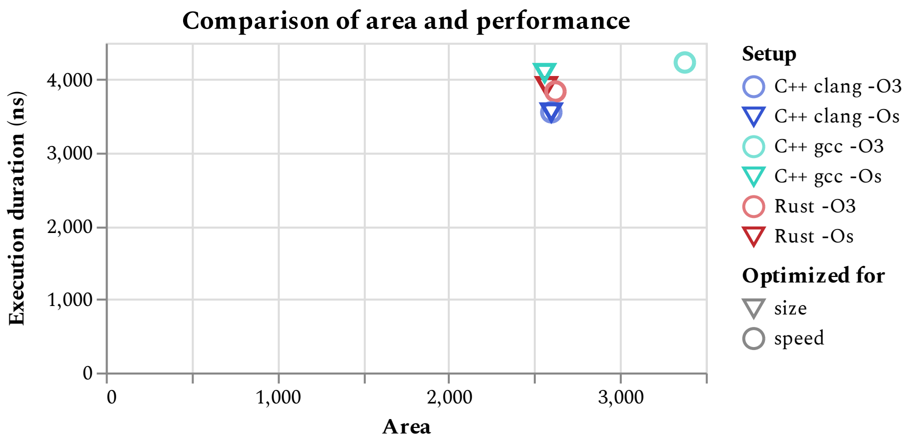


Figure 16. Area to performance of the MD5 designs

The Figure 16 and Figure 15 clearly show what the other measurements already indicated; all designs perform the same. The only exception is the GCC design optimized for speed, which is slightly slower and a bit larger than the other designs.

These measurements reveal several observations and trends:

- In this case, Bambu seems to perform similarly across all configurations.
- Rust designs have slightly worse performance than Clang.

- The optimization goal does not make a difference.

4.4 RESULTS FOR THE KECCAK-f[1600] DESIGNS

Listing 18 presents the reference C++ implementation. It is based on a reference implementation provided by the Keccak team. It operates on a pointer to an array of 25 64-bit elements and performs 24 rounds of mutation on these elements. Each round consists of 5 functions that transform the elements in various ways. The resulting values after 24 rounds represent the output of the function. Listing 13 depicts the KECCAK-f[1600] function.

Listing 13. Shortened `keccak` *function*

```rust
pub unsafe extern "C" fn keccak(a: *mut u64) -> () {
    for i in 0..24 {
        theta(a);
        rho(a);
        pi(a);
        chi(a);
        iota(a, i);
    }
}
```

The Rust implementation shown in Listing 19 closely follows the reference implementation. All functions of this direct port must be marked as unsafe because they use raw pointers. It is impossible to safely access raw pointers as they can potentially point to invalid memory.

The idiomatic Rust implementation, seen in Listing 20, is nearly identical to the Rust implementation. The main difference is that it casts the pointer to an array of size 25. This allows the compiler to verify that the array is never accessed out of bounds. It also could allow the compiler to optimize the code better, as it can assume that the array is never accessed out of bounds.

The fourth implementation uses the keccak crate from crates.io [Rus23]. This crate provides optimized implementations of the KECCAK sponge functions in different sizes. The tested function, depicted in Listing 14, calls the appropriate implementation from the crate. It is added to the comparison to determine whether using crates from the Rust ecosystem for HLS is practical.

Listing 14. Wrapper for the `keccak::f1600` *function from crates.io*

```rust
pub unsafe extern "C" fn keccak_crate(input: *mut u64) -> () {
    let input: &mut [u64; 25] = std::mem::transmute(input);
    keccak::f1600(input);
}
```

4.4.1 *Measurements*

Figure 17 illustrates the results of the area measurements. Both the Rust and idiomatic Rust designs exhibit similar sizes in both optimizations.

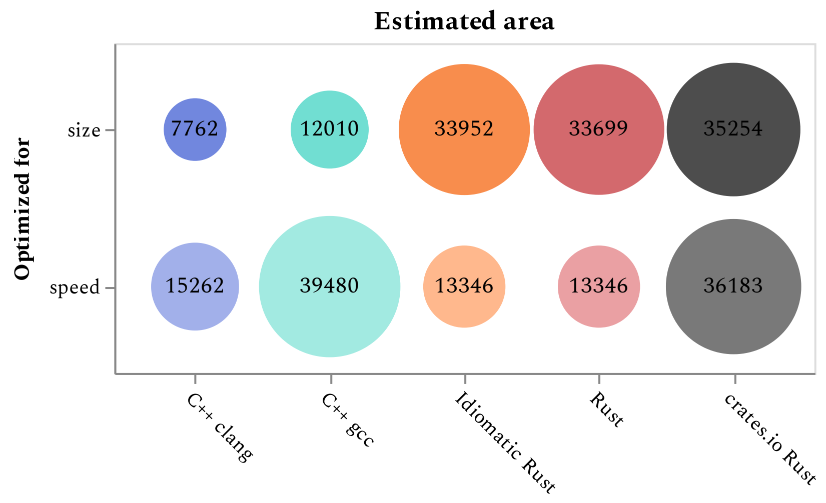


Figure 17. Area of the KECCAK-*f*[1600] *designs*

The three LLVM designs, optimized for speed and based on the reference implementation, also display similar sizes. This suggests that LLVM was able to optimize the code in a similar manner. When optimized for size, the Rust-based designs are noticeably larger than the Clang-based designs.

The Rust implementation from crates.io [Rus23] generates two nearly identically sized designs. They have roughly the same size as the other Rust designs optimized for size.

The GCC design optimized for speed is the biggest. The one optimized for size is the second smallest. This is what we would expect from the optimization goals.

The performance of each design was evaluated using three test cases. Every test case is a 25-element array of 64-bit integers. The input in the first test case was all zeros, and for the second, all 25s. The third test case received the result of the first test case as its input. The cycle count remained consistent across all designs and test cases. Figure 18 presents the results of the cycle count measurements.

Each of the 25 memory locations in the array needs to be both read and written at least once because they are used for input and output. Accessing memory takes one cycle and has a latency of two cycles, so the theoretical minimum for the number of cycles is 51. This only applies if the module loads all values into some kind of internal storage in the beginning, then takes one cycle to perform all rounds of keccak on that internal storage, and then writes the resulting values back to external memory. If the calculations are performed directly on external memory, the number of memory accesses will be significantly larger.

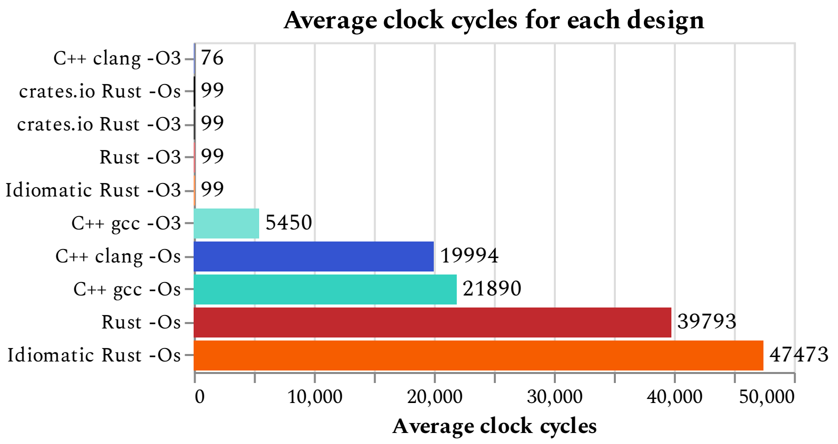


Figure 18. Clock cycles of the KECCAK-f[1600] *designs*

The LLVM designs optimized for speed all perform quite well, with the Clang design being the fastest. Interestingly the crate.io design optimized for size also performs quite well. The traces of these designs reveal that they all only required the minimum required amount of 50 memory accesses per calculation. The trace for the Clang design can be observed in Figure 19. The traces for the fast Rust designs are similar, but they take 48 cycles to do the calculation between the memory accesses. They require two cycles per round ($24 \cdot 2 = 48$). The Clang design only requires one cycle per round but has an additional cycle of overhead ($24 + 1 = 25$).

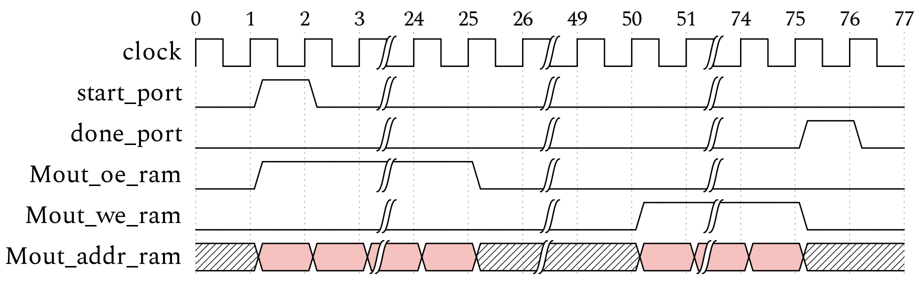


Figure 19. Trace of the Clang design optimized for speed

Surprisingly, the crates.io design, optimized for size, also took the same amount of cycles as the Rust designs optimized for speed. Notably, both crates.io designs perform memory reads during the calculation. These reads appear unrelated to the algorithm and random. Figure 20 depicts this behaviour. The addresses they try to access are out of bounds, and random values are returned during the tests. These memory accesses do not seem to affect the correctness of the result.

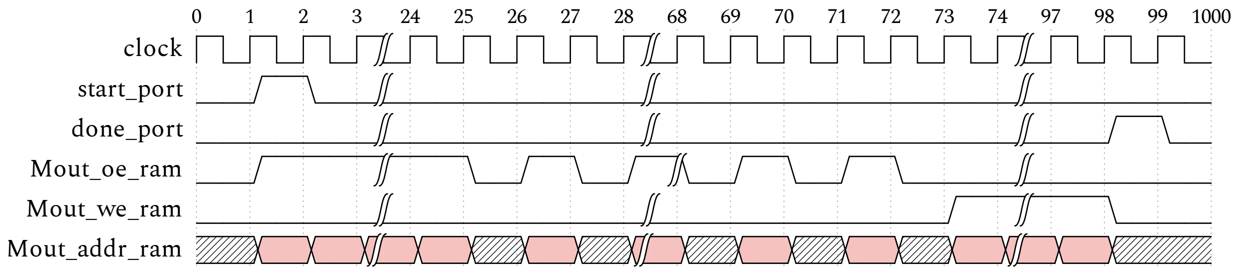


Figure 20. Trace of the crates.io design optimized for size

The remaining designs require significantly more cycles, earning them the classification of *slow* designs. The disparity between the fastest and slowest slow designs is approximately a factor of 10. The fastest slow design is 50 times slower than the slowest fast design. While the fast designs load all values in the beginning and store them when they are done, the slow designs constantly read and write the values in memory. The trace for the GCC design optimized for speed (Figure 21) shows that it constantly performs memory accesses. This behavior is representative of the other slow designs. How much of the calculations are performed on external memory seems to dictate the exact cycle count.

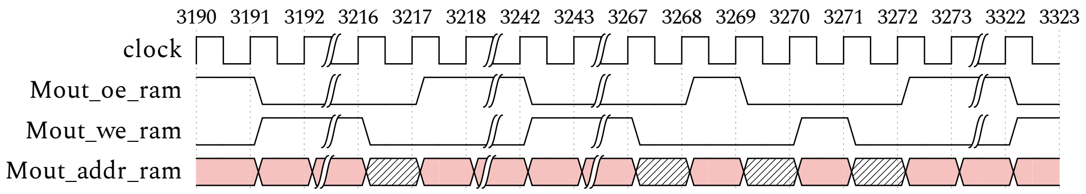


Figure 21. Clipped trace of the GCC design optimized for speed

It is likely that the designs optimized for size keep the split into the five functions (`theta` , `rho` , `pi` , `chi` , and `iota`) as it is usually smaller for a CPU program to be separated into multiple functions. As Bambu performs HLS on a per-function level, it might be unable to share internal memory between them.

Figure 22 displays the maximum frequency for each KECCAK-f[1600] design. The Rust designs optimized for speed, excluding the crates.io design, stand out as the fastest. The tradeoff to perform each round of KECCAK-f[1600] in two cycles rather than one proves advantageous, allowing them to run at higher frequencies compared to the LLVM design. Notably, both crate designs operate at the slowest frequency among all designs.

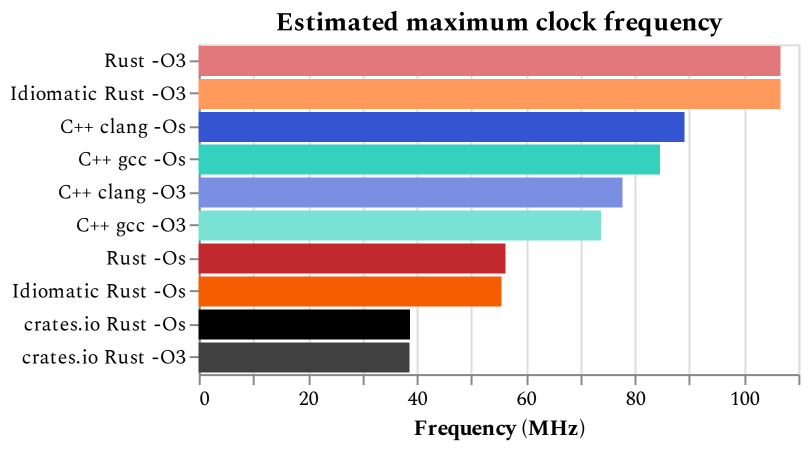


Figure 22. Maximum frequency of the KECCAK-f[1600] *designs*

The other Rust designs optimized for size run at a lower frequency than the other designs optimized for size. The size-optimized C++ designs run at the third and fourth highest frequency.

In Figure 23 the designs optimized for speed showcase similar performance to each other.

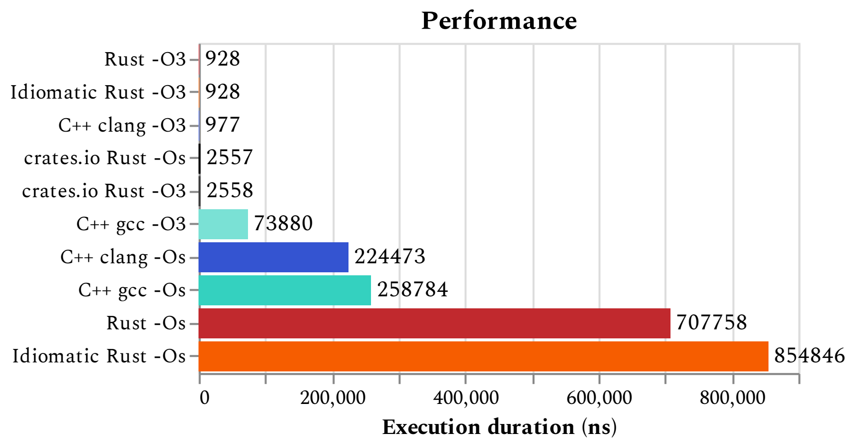


Figure 23. Execution time of the KECCAK-f[1600] *designs*

The execution time of the designs optimized for speed is roughly the same. Among the speed-optimized designs, the Rust designs are a bit faster than the LLVM designs. Even though these designs take more cycles, their higher clock speed is able to compensate for that. Among the fast designs, the crates.io designs are the slowest.

The designs optimized for size are all significantly slower. Among memory-accessing designs, the GCC design optimized for speed stands out as the fastest. It is still more than 20 times slower than the next fastest design.

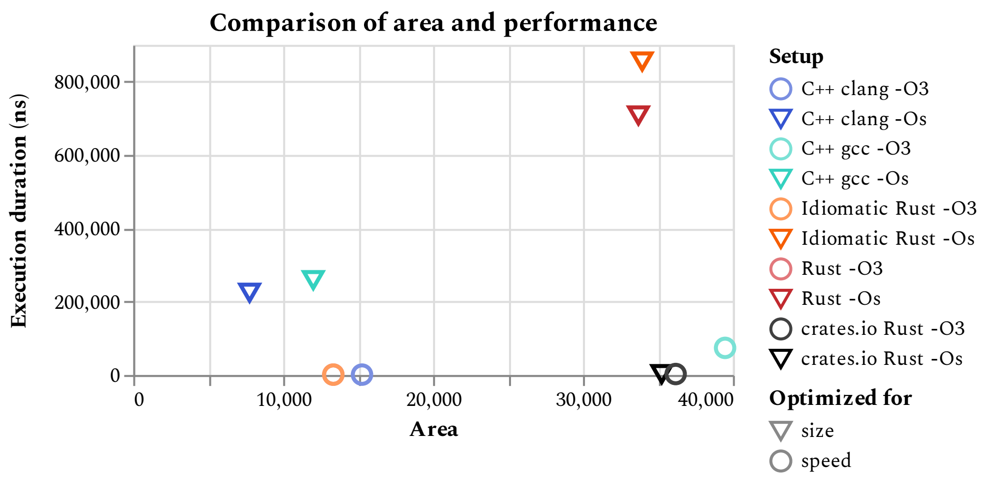


Figure 24. Area to performance of the KECCAK-f[1600] *designs*

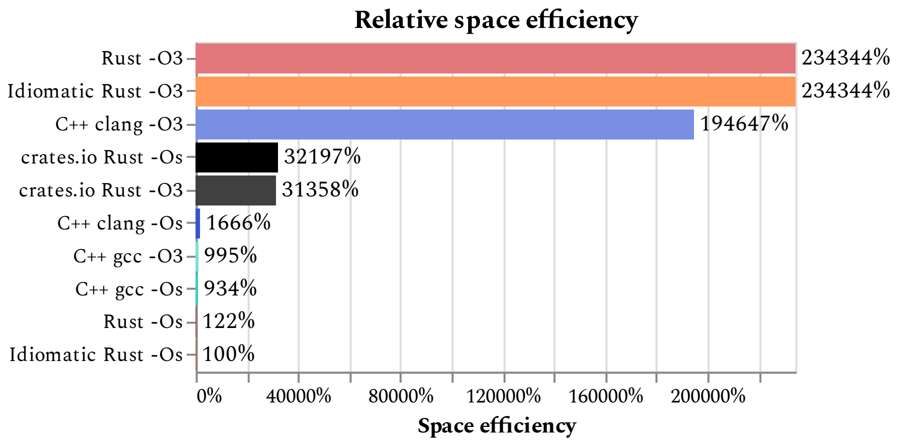


Figure 25. Space efficiency of the KECCAK-f[1600] *designs*

Figure 24 and Figure 25 demonstrate that the fast designs provide greater performance per area than the slow designs. While the size-optimized C++ designs are the smallest overall, their abysmal performance renders them less space efficient. The Rust designs optimized for size rank the worst, being both slow and large. On the other hand, the Rust designs optimized for speed are the most space-efficient. The crates.io designs are between them, with slightly slower performance than the other fast designs and similar size to the larger designs.

4.4.2 *Unexpected behavior of the crates.io design*

In the case of the crates.io design, it did not make any difference whether it was optimized for speed or for size. Listing 21 shows that their implementation explicitly inlines and unrolls the steps of a round. Apart from this explicit unrolling, the significant distinction lies in the variable number of rounds supported by the crates.io implementation. The Rust compiler seems to respect these optimizations. It generates basically the same LLVM IR for both optimizations. The biggest difference is that the version optimized for speed inlines the control of the main loop into the loop body.

When optimized for speed, the LLVM IR generated from other Rust implementations closely resembles the LLVM IR generated from the crates.io design. Notably, the crates.io designs are larger and slower, despite being based on similar LLVM IR. The most significant differentiation between them is the way they access the `KECCAK_ROUND_CONSTANTS` array. In each of the 24 rounds, a different constant from that array is needed. The array accesses use the same instructions in both cases, albeit in slightly different places. The crates.io implementation fetches the round constant in the middle of the round and the other rust implementations at the end. This difference should have no impact, as they have the same data dependencies and can be reordered. The crates.io design employs a pointer to an integer `i64*` to reference the array, while the other implementation utilizes a pointer to an array of known size `[24 x i64]*`.

It can be assumed that this causes Bambu to perform poorly for some reason. It is possible that the generated design attempts to access the array at an external memory location. This would also explain the 24 out-of-bounds memory accesses during calculations. These values from external memory are apparently not used during calculation. The generated design tries to access external memory when the actual values are in internal memory. It can be assumed that Bambu cannot prove that the instruction only accesses the internal memory. The design should, however, not try to access external memory in addition to internal memory. This is most likely a bug in Bambu.

The crates.io design also performs a check to verify that the number of rounds is not greater than 24 and panics if it is. It is safe to assume that Bambu detects that this will never happen in our case and optimizes it away. If Bambu did not do that, it would not be able to synthesize the design.

5 CONCLUSION

As mentioned previously, current HLS tools focus on the common systems-programming languages C or C++ as their input language. Rust could provide a more modern alternative.

It was shown that Bambu can be used to perform HLS from Rust. The Rust compiler can output LLVM IR, which can be used as an input for Bambu. When the Rust compiler is optimizing for speed, the generated designs are similar in terms of performance and size to the designs generated from C++/Clang. When optimizing for size, the results were mixed; some designs were smaller and faster, some were larger and bigger, and some were similar. They showed that the Rust-based designs usually perform similarly to the C++/Clang-based designs. It was also demonstrated that whether the Rust specifications are C-like or more Rust-like does not make a difference. A specification using an implementation from crates.io was able to be synthesized with promising results. In that specific case, the resulting design was larger than necessary, probably caused by a bug in Bambu. As expected, the support for Rust in Bambu is not as mature as for C++.

There are multiple challenges when using the Rust compiler to generate LLVM IR that is compatible with Bambu. Bambu does not support all instructions, so the Rust compiler needs to be configured only to generate compatible IR. Bambu's LLVM support is focused on LLVM IR generated by Clang. As a result, Bambu is not well-tested on LLVM IR generated by other tools. While Bambu is able to process and compile them, the resulting designs might not be as optimized as they are with Clang. This can lead to bugs like the one described in Section 4.4.2. Rust also performs bounds checks on all memory accesses that could overflow. If these bound-checks cannot be removed during optimization, the resulting LLVM IR contains calls to panic which cannot be synthesized by Bambu. By default, Rust also performs overflow checks on all integer arithmetics, which also can result in panics that can be disabled. Another limitation is that only Rust functions with a C-compatible interface can be synthesized.

RustHLS, a framework for integrating the toolchain with RustHDL, was developed as part of this thesis. RustHLS makes it possible to create a single Rust crate containing both RTL and HLS descriptions. It also enables to write tests for the specification and the generated design side-by-side as Rust unit tests. It achieves this by searching the project for modules marked as HLS specifications, running the toolchain for the found specifications, and embedding the generated modules into the original project. If a function marked as HLS specification shows obvious problems, helpful error messages will be emitted before synthesis is performed. It also allows individual configuration of the toolchain for each specification.

6 FUTURE WORK

Future work on HLS from Rust should focus on improving support for Rust-generated LLVM IR in HLS tools. While Bambu can currently synthesize the LLVM IR generated by Rust, it seems possible to improve the generated designs in some cases. Finding these cases and improving Bambu to handle them should be possible.

It would also be interesting to see if the LLVM IR generated by Rust can be improved to generate better designs. Our toolchain builds every project dependency separately and then links them together. This is not ideal, as it does not allow the compiler to perform cross-crate link-time optimizations. This would probably require adjusting cargo or the Rust compiler to allow link-time optimizations when compiling to LLVM IR. The toolchain only configures the Rust compiler with the settings necessary for generating LLVM IR that Bambu can synthesize. Besides that, it uses rustc's default optimization profiles for speed and size. There are probably significant improvements possible by evaluating which optimizations are useful for HLS and which are not.

LIST OF ABBREVIATIONS

FPGA
Field-Programmable Gate Array

HLS
High-Level Synthesis

HDL
Hardware Description Language

SRAM
Static RAM

ADL
Accelerator Design Language

GPU
Graphics Processing Unit

LLVM IR
LLVM Intermediate Representation

RTL
Register-Transfer Level

DUT
Design/Device Under Test

ASIC
Application Specific Integrated Circuit

QoR
Quality of Results

CPU
Central Processing Unit

LUT
Look-Up Table

FF
Flip-Flop

DFF
D Flip-Flop

BRAM

Block RAM

DSP
Digital Signal Processor

CLB
Configurable Logic Block

LB
Logic Block

LE
Logic Element

RAII
Resource Acquisition Is Initialization / Scope-Bound Resource Management

HIR
High-level Intermediate Representation

THIR
Typed HIR

MIR
Mid-level Intermediate Representation

PAL
Programmable Array Logic

CFG
Control-Flow Graph

SSA
Static Single Assignment

GCC
GNU Compiler Collection

LLVM
LLVM is not an acronym

VHDL
Very High-Speed Integrated Circuit Hardware Description Language

REFERENCES

[Sta16] Stack Overflow
Stack Overflow Developer Survey 2016
[Online; accessed 5.7.23]
insights.stackoverflow.com/survey/2016

[Sta20] Stack Overflow
Stack Overflow Developer Survey 2020
[Online; accessed 5.7.23]
insights.stackoverflow.com/survey/2020

[Sta23] Stack Overflow
Stack Overflow Developer Survey 2023
[Online; accessed 5.7.23]
survey.stackoverflow.co/2023

[Bug22] William Bugden, Ayman Alahmar
Rust: The Programming Language for Safety and Performance
arXiv preprint
10.48550/arXiv.2206.05503 📁

[Sah22] Arash Sahebolamri, Thomas Gilray, Kristopher Micinski
Seamless Deductive Inference via Macros
ACM SIGPLAN International Conference on Compiler Construction
10.1145/3497776.3517779 📁

[Byc22] Andrey Bychkov, Vsevolod Nikolskiy
Rust Language for GPU Programming
Russian Supercomputing Days, Revised Selected Papers
10.1007/978-3-031-22941-1_38 📁

[Kyr22] Kyriakos-Ioannis D. Kyriakou, Nikolaos D. Tselikas
Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly.
Electronics
10.3390/electronics11193217 📁

[Cos19] Cosmin Cartas
Rust - The Programming Language for Every Industry
Economic Informatics Journal
10.12948/ei2019.01.05 📁

[Mil20] Roberto Millón, Emmanuel Frati, Enzo Rucci
A Comparative Study between HLS and HDL on SoC for Image Processing Applications
Revista elektron
10.37537/rev.elektron.4.2.117.2020 📁

[Smi21] Smiths Digital Forge, Samit Basu
A framework for writing FPGA firmware using the Rust Programming Language
[Online; accessed 5.7.23]
github.com/samitbasu/rust-hdl

[Nan16] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, Koen Bertels
A Survey and Evaluation of FPGA High-Level Synthesis Tools
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
10.1109/tcad.2015.2513673 📁

[Har22] David Hardin
Hardware/Software Co-Assurance using the Rust Programming Language and ACL2
arXiv preprint
10.48550/arXiv.2205.11709 📁

[Smi96] Douglas J. Smith
VHDL & Verilog compared & contrasted—plus modeled example written in VHDL, Verilog and C.
Annual Design Automation Conference
10.1145/240518.240664 📁

[Soz22] Emanuele Del Sozzo, Davide Conficconi, Alberto Zeni, Mirko Salaris, Donatella Sciuto, Marco D. Santambrogio
Pushing the level of abstraction of digital system design: A survey on how to program FPGAs
ACM Computing Surveys
10.1145/3532989 📁

[Fos15] Harry D. Foster
Trends in Functional Verification: A 2014 Industry Study
Annual Design Automation Conference
10.1145/2744769.2744921 📁

[Fla20] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, Simon J. Davidmann
Verilog HDL and its ancestors and descendants.
Proceedings of the ACM on Programming Languages
10.1145/3386337 📁

[Bot21] Andrew Boutros, Betz Vaughn
FPGA architecture: Principles and progression
IEEE Circuits and Systems Magazine
10.1109/MCAS.2021.3071607 📁

[Bal07] James Ball
Designing soft-core processors for FPGAs
Processor Design
10.1007/978-1-4020-5530-0_11 📁

[Mic95] Microprocessors and Microsystems
Critical path analysis for field-programmable gate arrays
Microprocessors and Microsystems
10.1016/0141-9331(95)90010-1 📁

[Lah19] Sakari Lahti, Panu Sjövall, Jarno Vanne, Timo D. Hämäläinen
Are We There Yet? A Study on the State of High-Level Synthesis
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
10.1109/TCAD.2018.2834439 📁

[Bar23] Benjamin L.C. Barzen, Arya Reais-Parsi, Eddie Hung, Minwoo Kang, Alan Mishchenko, Jonathan W. Greene, John Wawrzynek
Narrowing the Synthesis Gap: Academic FPGA Synthesis is Catching Up With the Industry
Design, Automation & Test in Europe Conference and Exhibition
10.23919/DATE56975.2023.10137310 📁

[Cou09] Philippe Coussy, Daniel D. Gajski, Michael Meredith, Andres Takach
An Introduction to High-Level Synthesis
IEEE Design & Test of Computers
10.1109/MDT.2009.69

[Lat04] Chris Lattner, Vikram Adve
LLVM: a compilation framework for lifelong program analysis & transformation
International Symposium on Code Generation and Optimization
10.1109/CGO.2004.1281665 📁

[Rus18] The Rust Project Developers
Rust Compiler Development Guide (rustc-dev-guide)
[Online; accessed 5.7.23]
github.com/rust-lang/rustc-dev-guide 📁

[Kla23] Steve Klabnik, Carol Nichols
The Rust programming language
[Online; accessed 5.7.23]
doc.rust-lang.org/stable/book

[Str12] Bjarne Stroustrup
Foundations of C++
Programming Languages and Systems
10.1007/978-3-642-28869-2_1 📁

[Jun17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, Derek Dreyer
RustBelt: Securing the Foundations of the Rust Programming Language
Proceedings of the ACM on Programming Languages
10.1145/3158154 📁

[Fer21] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, Antonino Tumeo
Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications
ACM/IEEE Design Automation Conference
10.1109/DAC18074.2021.9586110 📁

[Rus23] RustCrypto Developers
Pure Rust implementation of the KECCAK *sponge function, including the* KECCAK-f *and* KECCAK-p *variants*
[Online; accessed 5.7.23]
crates.io/crates/keccak

7 APPENDIX

Listing 15. `minmax` *function in C++*

CPP

```cpp
#include <limits.h>

void min_max(int *numbers, int numbers_length, int *out_max, int *out_min) {
  int local_max = INT_MIN;
  int local_min = INT_MAX;
  int i = 0;
  for (i = 0; i < numbers_length; i++) {
    if (numbers[i] > local_max) {
      local_max = numbers[i];
    }
    if (numbers[i] < local_min) {
      local_min = numbers[i];
    }
  }
  *out_max = local_max;
  *out_min = local_min;
}
```

Listing 16. `md5` *function in C++*

CPP

```cpp
#include <cstdint>

constexpr uint32_t SHIFT_PER_ROUND[64] = {
    7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
    5, 9,  14, 20, 5, 9,  14, 20, 5, 9,  14, 20, 5, 9,  14, 20,
    4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
    6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21,
};

constexpr uint32_t K[64] = {
    0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf, 0x4787c62a,
    0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
    0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562, 0xc040b340,
    0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
    0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8,
    0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
    0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70, 0x289b7ec6, 0xeaa127fa,
    0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
    0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92,
    0xffeff47d, 0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
    0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391,
};

constexpr uint32_t A0 = 0x67452301;
constexpr uint32_t B0 = 0xefcdab89;
constexpr uint32_t C0 = 0x98badcfe;
constexpr uint32_t D0 = 0x10325476;

uint32_t rotl(uint32_t n, int d){
     return (n << d)|(n >> (32 - d));
}

void md5(uint32_t *message_pointer, uint32_t *result_pointer) {
  uint32_t a = A0;
  uint32_t b = B0;
  uint32_t c = C0;
  uint32_t d = D0;

  for (int i = 0; i < 64; i++) {
    uint32_t f = 0;
    uint32_t g = 0;

    if (i <= 15) {
      f = (b & c) | ((~b) & d);
      g = i;
    } else if (i <= 31) {
      f = (d & b) | ((~d) & c);
      g = (5 * i + 1) % 16;
    } else if (i <= 47) {
      f = b ^ c ^ d;
      g = (3 * i + 5) % 16;
    } else if (i <= 63) {
      f = c ^ (b | (~d));
      g = (7 * i) % 16;
    }

    uint32_t current_message_data = message_pointer[g];
    uint32_t rot = f + a + K[i] + current_message_data;

    a = d;
    d = c;
    c = b;
    b = b + rotl(rot, SHIFT_PER_ROUND[i]);
  }

  a = a + A0;
  b = b + B0;
  c = c + C0;
  d = d + D0;

  result_pointer[0] = a;
  result_pointer[1] = b;
  result_pointer[2] = c;
  result_pointer[3] = d;
}
```

Listing 17. `md5` *implementation in Rust*

CPP

```
const SHIFT_PER_ROUND: [u32; 64] = [
    7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20,
    5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4,
    11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6,
    10, 15, 21,
];

const K: [u32; 64] = [
    0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf, 0x4787c62a,
    0xa8304613, 0xfd469501, 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
    0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821, 0xf61e2562, 0xc040b340,
    0x265e5a51, 0xe9b6c7aa, 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
    0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed, 0xa9e3e905, 0xfcefa3f8,
    0x676f02d9, 0x8d2a4c8a, 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
    0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70, 0x289b7ec6, 0xeaa127fa,
    0xd4ef3085, 0x04881d05, 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
    0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3, 0x8f0ccc92,
    0xffeff47d, 0x85845dd1, 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
    0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391,
];

const A0: u32 = 0x67452301;
const B0: u32 = 0xefcdab89;
const C0: u32 = 0x98badcfe;
const D0: u32 = 0x10325476;

/// Based on the pseudocode from https://en.wikipedia.org/wiki/MD5
pub unsafe extern "C" fn md5(
    message_pointer: *const u32,
    result_pointer: *mut u32,
) -> () {
    let message = std::slice::from_raw_parts(message_pointer, 16);
    let mut a = A0; // A
    let mut b = B0; // B
    let mut c = C0; // C
    let mut d = D0; // D

    // for i from 0 to 63 do
    for i in 0u32..64 {
        // var int F, g
        let f: u32;
        let g: u32;

        match i {
            // if 0 ≤ i ≤ 15 then
            //     F := (B and C) or ((not B) and D)
            //     g := i
            0..=15 => {
                f = (b & c) | ((!b) & d);
                g = i;
            }
            // else if 16 ≤ i ≤ 31 then
            //     F := (D and B) or ((not D) and C)
            //     g := (5×i + 1) mod 16
            16..=31 => {
                f = (d & b) | ((!d) & c);
                g = (5 * i + 1) % 16;
            }
            // else if 32 ≤ i ≤ 47 then
            //     F := B xor C xor D
            //     g := (3×i + 5) mod 16
            32..=47 => {
                f = b ^ c ^ d;
                g = (3 * i + 5) % 16;
            }
            // else if 48 ≤ i ≤ 63 then
            //     F := C xor (B or (not D))
            //     g := (7×i) mod 16
            48..=63 => {
                f = c ^ (b | (!d));
                g = (7 * i) % 16;
            }
            _ => {
                f = 0;
                g = 0;
            }
        }

        // // Be wary of the below definitions of a,b,c,d
        // F := F + A + K[i] + M[g]  // M[g] must be a 32-bit block
        let current_message_data = message[g as usize];
        let rot = f
            .wrapping_add(a)
            .wrapping_add(K[i as usize])
            .wrapping_add(current_message_data);

        // A := D
        a = d;
        // D := C
        d = c;
        // C := B
        c = b;
        // B := B + leftrotate(F, s[i])
        b = b.wrapping_add(rot.rotate_left(SHIFT_PER_ROUND[i as usize]));
        // end for
    }

    a = a.wrapping_add(A0);
    b = b.wrapping_add(B0);
    c = c.wrapping_add(C0);
    d = d.wrapping_add(D0);

    let result = std::slice::from_raw_parts_mut(result_pointer, 4 as usize);
    result[0] = a;
    result[1] = b;
    result[2] = c;
    result[3] = d;
}
```

Listing 18. `keccak` *implementation in C++*

CPP

```cpp
/*
 * The Keccak sponge function, designed by Guido Bertoni, Joan Daemen,
 * Michaël Peeters and Gilles Van Assche. For more information, feedback or
 * questions, please refer to our website: http://keccak.noekeon.org/
 * Implementation by the designers,
 * hereby denoted as "the implementer".
 * To the extent possible under law, the implementer has waived all copyright
 * and related or neighboring rights to the source code in this file.
 * http://creativecommons.org/publicdomain/zero/1.0/
 *
 */
typedef unsigned char UINT8;
typedef unsigned long long int UINT64;
#define nrRounds 24

#define GET_KRC_VAL(index) (KeccakRoundConstants[index])

static UINT64 KeccakRoundConstants[nrRounds] = {
    0x0000000000000001ULL, 0x0000000000008082ULL, 0x800000000000808aULL,
    0x8000000080008000ULL, 0x000000000000808bULL, 0x0000000080000001ULL,
    0x8000000080008081ULL, 0x8000000000008009ULL, 0x000000000000008aULL,
    0x0000000000000088ULL, 0x0000000080008009ULL, 0x000000008000000aULL,
    0x000000008000808bULL, 0x800000000000008bULL, 0x8000000000008089ULL,
    0x8000000000008003ULL, 0x8000000000008002ULL, 0x8000000000000080ULL,
    0x000000000000800aULL, 0x800000008000000aULL, 0x8000000080008081ULL,
    0x8000000000008080ULL, 0x0000000080000001ULL, 0x8000000080008008ULL};

#define nrLanes 25
static unsigned char KeccakRhoOffsets[nrLanes] = {
    0,  1,  62, 28, 27, 36, 44, 6,  55, 20, 3,  10, 43,
    25, 39, 41, 45, 15, 21, 8,  18, 2,  61, 56, 14};

#define index(x, y) (((x) % 5) + 5 * ((y) % 5))
#define ROL64(a, offset)                                                    \
  ((offset != 0) ? ((((UINT64)a) << offset) ^ (((UINT64)a) >> (64 - offset)))  \
                 : a)

void theta(UINT64 *A) {
  unsigned int x, y;
  UINT64 C[5], D[5];

  for (x = 0; x < 5; x++) {
    C[x] = 0;
    for (y = 0; y < 5; y++)
      C[x] ^= A[index(x, y)];
  }
  for (x = 0; x < 5; x++)
    D[x] = ROL64(C[(x + 1) % 5], 1) ^ C[(x + 4) % 5];
  for (x = 0; x < 5; x++)
    for (y = 0; y < 5; y++)
      A[index(x, y)] ^= D[x];
}

void rho(UINT64 *A) {
  unsigned int x, y;

  for (x = 0; x < 5; x++)
    for (y = 0; y < 5; y++)
      A[index(x, y)] = ROL64(A[index(x, y)], KeccakRhoOffsets[index(x, y)]);
}

void pi(UINT64 *A) {
  unsigned int x, y;
  UINT64 tempA[25];

  for (x = 0; x < 5; x++)
    for (y = 0; y < 5; y++)
      tempA[index(x, y)] = A[index(x, y)];
  for (x = 0; x < 5; x++)
    for (y = 0; y < 5; y++)
      A[index(0 * x + 1 * y, 2 * x + 3 * y)] = tempA[index(x, y)];
}

void chi(UINT64 *A) {
  unsigned int x, y;
  UINT64 C[5];

  for (y = 0; y < 5; y++) {
    for (x = 0; x < 5; x++)
      C[x] = A[index(x, y)] ^ ((~A[index(x + 1, y)]) & A[index(x + 2, y)]);
    for (x = 0; x < 5; x++)
      A[index(x, y)] = C[x];
  }
}

void iota(UINT64 *A, unsigned int indexRound) {
  A[index(0, 0)] ^= GET_KRC_VAL(indexRound);
}

void keccak(UINT64 A[25]) {
  unsigned int i;
  for (i = 0; i < nrRounds; i++) {
    theta(A);
    rho(A);
    pi(A);
    chi(A);
    iota(A, i);
  }
}
```

Listing 19. `keccak` *implementation in Rust*

```rust
/*
 * The Keccak sponge function, designed by Guido Bertoni, Joan Daemen,
 * Michaël Peeters and Gilles Van Assche. For more information, feedback or
 * questions, please refer to our website: http://keccak.noekeon.org/
 * Implementation by the designers,
 * hereby denoted as "the implementer".
 * To the extent possible under law, the implementer has waived all copyright
 * and related or neighboring rights to the source code in this file.
 * http://creativecommons.org/publicdomain/zero/1.0/
 */
macro_rules! get_krc_val {
    ($index:expr) => {
        KECCAK_ROUND_CONSTANTS[$index]
    };
}

const KECCAK_ROUND_CONSTANTS: [u64; 24] = [
    0x0000000000000001u64,
    0x0000000000008082u64,
    0x800000000000808au64,
    0x8000000080008000u64,
    0x000000000000808bu64,
    0x0000000080000001u64,
    0x8000000080008081u64,
    0x8000000000008009u64,
    0x000000000000008au64,
    0x0000000000000088u64,
    0x0000000080008009u64,
    0x000000008000000au64,
    0x000000008000808bu64,
    0x800000000000008bu64,
    0x8000000000008089u64,
    0x8000000000008003u64,
    0x8000000000008002u64,
    0x8000000000000080u64,
    0x000000000000800au64,
    0x800000008000000au64,
    0x8000000080008081u64,
    0x8000000000008080u64,
    0x0000000080000001u64,
    0x8000000080008008u64,
];

const NR_LANES: usize = 25;

const KECCAK_RHO_OFFSETS: [u8; NR_LANES] = [
    0, 1, 62, 28, 27, 36, 44, 6, 55, 20, 3, 10, 43, 25, 39, 41, 45, 15, 21, 8,
    18, 2, 61, 56, 14,
];

macro_rules! index {
    ($x:expr, $y:expr) => {
        (($x) % 5) + 5 * (($y) % 5)
    };
}

macro_rules! rol64 {
    ($a:expr, $offset:expr) => {
        (if ($offset != 0) {
            (((u64::from($a)) << $offset) ^ ((u64::from($a)) >> (64 - $offset)))
        } else {
            $a
        })
    };
}

unsafe fn theta(a: *mut u64) -> () {
    let mut c: [u64; 5] = [0; 5];
    let mut d: [u64; 5] = [0; 5];

    for x in 0..5 {
        for y in 0..5 {
            c[x] ^= *a.add(index!(x, y));
        }
    }
    for x in 0..5 {
        d[x] = rol64!(c[(x + 1) % 5], 1) ^ c[(x + 4) % 5];
    }
    for x in 0..5 {
        for y in 0..5 {
            *a.add(index!(x, y)) ^= d[x];
        }
    }
}

unsafe fn rho(a: *mut u64) -> () {
    for x in 0..5 {
        for y in 0..5 {
            *a.add(index!(x, y)) =
                rol64!(*a.add(index!(x, y)), KECCAK_RHO_OFFSETS[index!(x, y)]);
        }
    }
}

unsafe fn pi(a: *mut u64) -> () {
    let mut temp_a: [u64; 25] = [0; 25];

    for x in 0..5 {
        for y in 0..5 {
            temp_a[index!(x, y)] = *a.add(index!(x, y));
        }
    }

    for x in 0..5 {
        for y in 0..5 {
            *a.add(index!(0 * x + 1 * y, 2 * x + 3 * y)) = temp_a[index!(x, y)];
        }
    }
}

unsafe fn chi(a: *mut u64) -> () {
    let mut c: [u64; 5] = [0; 5];

    for y in 0..5 {
        for x in 0..5 {
            c[x] = *a.add(index!(x, y))
                ^ ((!*a.add(index!(x + 1, y))) & *a.add(index!(x + 2, y)));
        }
        for x in 0..5 {
            *a.add(index!(x, y)) = c[x];
        }
    }
}

unsafe fn iota(a: *mut u64, index_round: usize) -> () {
    *a ^= get_krc_val!(index_round);
}

pub unsafe extern "C" fn keccak(a: *mut u64) -> () {
    for i in 0..24 {
        theta(a);
        rho(a);
        pi(a);
        chi(a);
        iota(a, i);
    }
}
```

Listing 20. Idiomatic `keccak` implementation in Rust

```rust
const KECCAK_ROUND_CONSTANTS: [u64; 24] = [
    0x0000000000000001u64,
    0x0000000000008082u64,
    0x800000000000808au64,
    0x8000000080008000u64,
    0x000000000000808bu64,
    0x0000000080000001u64,
    0x8000000080008081u64,
    0x8000000000008009u64,
    0x000000000000008au64,
    0x0000000000000088u64,
    0x0000000080008009u64,
    0x000000008000000au64,
    0x000000008000808bu64,
    0x800000000000008bu64,
    0x8000000000008089u64,
    0x8000000000008003u64,
    0x8000000000008002u64,
    0x8000000000000080u64,
    0x000000000000800au64,
    0x800000008000000au64,
    0x8000000080008081u64,
    0x8000000000008080u64,
    0x0000000080000001u64,
    0x8000000080008008u64,
];

const KECCAK_RHO_OFFSETS: [u8; 25] = [
    0, 1, 62, 28, 27, 36, 44, 6, 55, 20, 3, 10, 43, 25, 39, 41, 45, 15, 21, 8,
    18, 2, 61, 56, 14,
];

macro_rules! index {
    ($x:expr, $y:expr) => {
        (($x) % 5) + 5 * (($y) % 5)
    };
}

macro_rules! rol64 {
    ($a:expr, $offset:expr) => {
        (if ($offset != 0) {
            (((u64::from($a)) << $offset) ^ ((u64::from($a)) >> (64 - $offset)))
        } else {
            $a
        })
    };
}

fn theta(a: &mut [u64; 25]) -> () {
    let mut c: [u64; 5] = [0; 5];
    let mut d: [u64; 5] = [0; 5];

    for x in 0..5 {
        for y in 0..5 {
            c[x] ^= a[index!(x, y)];
        }
    }
    for x in 0..5 {
        d[x] = rol64!(c[(x + 1) % 5], 1) ^ c[(x + 4) % 5];
    }
    for x in 0..5 {
        for y in 0..5 {
            a[index!(x, y)] ^= d[x];
        }
    }
```

```rust
}

fn rho(a: &mut [u64; 25]) -> () {
    for x in 0..5 {
        for y in 0..5 {
            a[index!(x, y)] =
                rol64!(a[index!(x, y)], KECCAK_RHO_OFFSETS[index!(x, y)]);
        }
    }
}

fn pi(a: &mut [u64; 25]) -> () {
    let mut temp_a: [u64; 25] = [0; 25];

    for x in 0..5 {
        for y in 0..5 {
            temp_a[index!(x, y)] = a[index!(x, y)];
        }
    }

    for x in 0..5 {
        for y in 0..5 {
            a[index!(0 * x + 1 * y, 2 * x + 3 * y)] = temp_a[index!(x, y)];
        }
    }
}

fn chi(a: &mut [u64; 25]) -> () {
    let mut c: [u64; 5] = [0; 5];

    for y in 0..5 {
        for x in 0..5 {
            c[x] = a[index!(x, y)]
                ^ ((!a[index!(x + 1, y)]) & a[index!(x + 2, y)]);
        }
        for x in 0..5 {
            a[index!(x, y)] = c[x];
        }
    }
}

fn iota(a: &mut [u64; 25], index_round: usize) -> () {
    a[0] ^= KECCAK_ROUND_CONSTANTS[index_round];
}

pub unsafe extern "C" fn keccak(a: *mut u64) -> () {
    let a: &mut [u64; 25] = std::mem::transmute(a);
    for i in 0..24 {
        theta(a);
        rho(a);
        pi(a);
        chi(a);
        iota(a, i);
    }
}
```

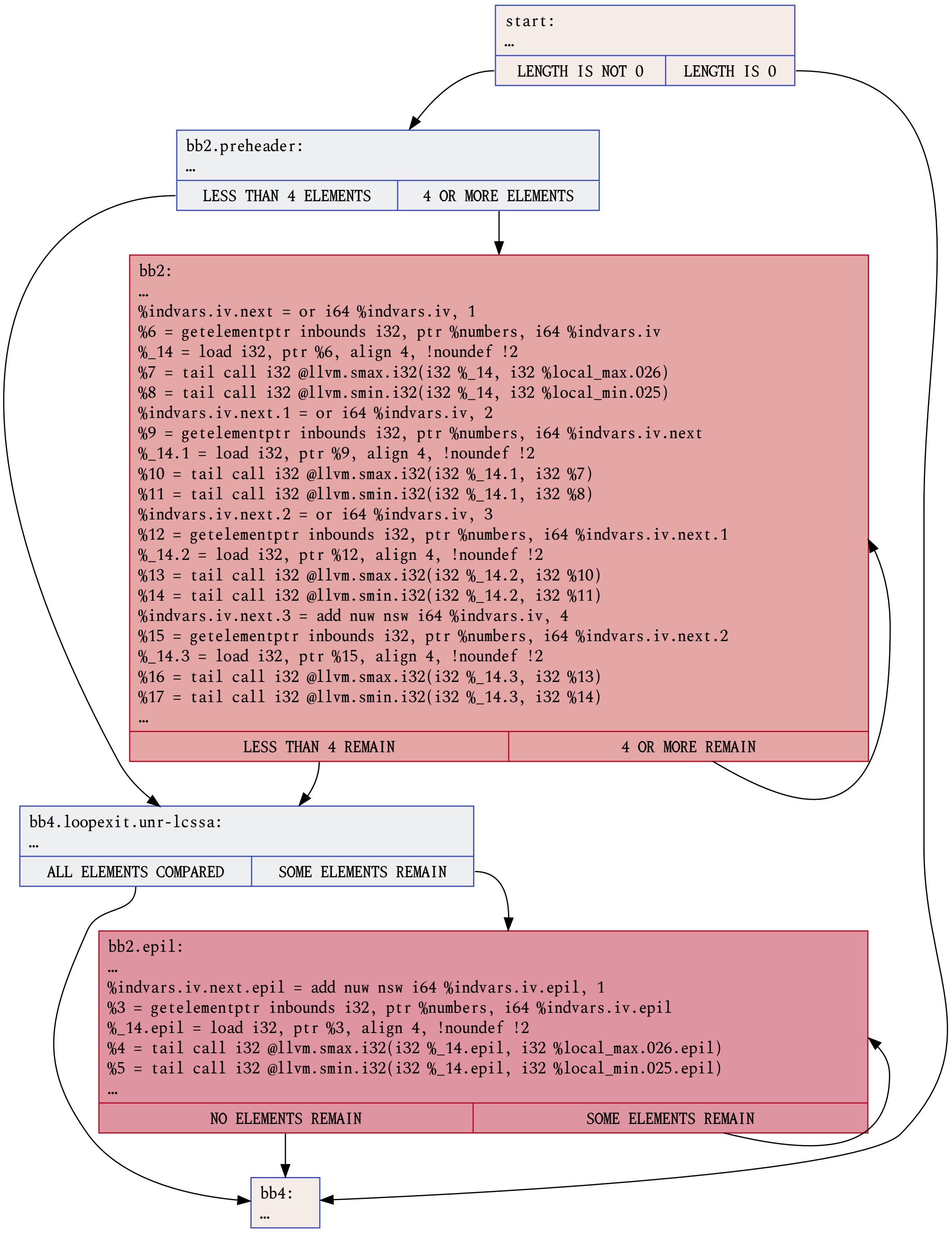
Listing 21. `keccak_p` *implementation from the keccak crate*

```rust
/// Source: https://github.com/RustCrypto/sponges/tree/master/keccak
///
/// Generic Keccak-p sponge function
pub fn keccak_p<L: LaneSize>(state: &mut [L; PLEN], round_count: usize) {
    if round_count > L::KECCAK_F_ROUND_COUNT {
        panic!("A round_count greater than KECCAK_F_ROUND_COUNT is not supported!");
    }

    // https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf#page=25
    // "the rounds of KECCAK-p[b, nr] match the last rounds of KECCAK-f[b]"
    let round_consts = &RC[(L::KECCAK_F_ROUND_COUNT - round_count)..L::KECCAK_F_ROUND_COUNT];

    // not unrolling this loop results in a much smaller function, plus
    // it positively influences performance due to the smaller load on I-cache
    for &rc in round_consts {
        let mut array = [L::default(); 5];

        // Theta
        unroll5!(x, {
            unroll5!(y, {
                array[x] ^= state[5 * y + x];
            });
        });

        unroll5!(x, {
            unroll5!(y, {
                let t1 = array[(x + 4) % 5];
                let t2 = array[(x + 1) % 5].rotate_left(1);
                state[5 * y + x] ^= t1 ^ t2;
            });
        });

        // Rho and pi
        let mut last = state[1];
        unroll24!(x, {
            array[0] = state[PI[x]];
            state[PI[x]] = last.rotate_left(RHO[x]);
            last = array[0];
        });

        // Chi
        unroll5!(y_step, {
            let y = 5 * y_step;

            unroll5!(x, {
                array[x] = state[y + x];
            });

            unroll5!(x, {
                let t1 = !array[(x + 1) % 5];
                let t2 = array[(x + 2) % 5];
                state[y + x] = array[x] ^ (t1 & t2);
            });
        });

        // Iota
        state[0] ^= L::truncate_rc(rc);
    }
}
```

Listing 22. Example of a generated RustHDL struct

```rust
/// This file was generated by rust_hls. Please do not edit it manually.
/// rust_hls hash: "fc1b10f200f5632694995e666ba00202"

extern crate verilated;
use ::rust_hdl::prelude::*;

#[allow(dead_code, unused)]
mod minmax_verilated {
    /// Bindings to the C++ library generated by Verilator go here
}

#[derive(::std::default::Default)]
pub struct Minmax {
    pub clk: Signal<
        In,
        Clock,
    >,
    pub reset: Signal<In, bool>,
    pub start_port: Signal<In, bool>,
    pub elements: Signal<In,Bits<32usize>>,
    pub num_elements: Signal<In,Bits<32usize>>,
    pub m_rdata_ram: Signal<In,Bits<32usize>>,
    pub m_data_rdy: Signal<In, bool>,
    pub done_port: Signal<Out, bool>,
    pub return_port: Signal<Out,Bits<64usize>,>,
    pub mout_oe_ram: Signal<Out, bool>,
    pub mout_we_ram: Signal<Out, bool>,
    pub mout_addr_ram: Signal<Out,Bits<32usize>>,
    pub mout_wdata_ram: Signal<Out,Bits<32usize>>,
    pub mout_data_ram_size: Signal<Out,Bits<6usize>>,
    verilated_module: Arc<Mutex<self::minmax_verilated::MinmaxVerilated>>,
}
unsafe impl Send for Minmax {}

#[automatically_derived]
```

```
impl Logic for Minmax {
    fn update(&mut self) {
        let mut verilated_module = match self.verilated_module.lock() {
            Ok(verilated_module) => verilated_module,
            Err(e) => panic!("Failed to aquire verilated_module lock: {}", e),
        };
        verilated_module.set_clk(if self.clk.val().clk { 1u8 } else { 0u8 });
        verilated_module.set_reset(if self.reset.val() { 1u8 } else { 0u8 });
        verilated_module.set_start_port(if self.start_port.val() { 1u8 } else { 0u8 });
        verilated_module.set_Pd61(self.elements.val().to_u32());
        verilated_module.set_Pd62(self.num_elements.val().to_u32());
        verilated_module.set_M_Rdata_ram(self.m_rdata_ram.val().to_u32());
        verilated_module.set_M_DataRdy(if self.m_data_rdy.val() { 1u8 } else { 0u8 });
        verilated_module.eval();
        self.done_port.next = verilated_module.done_port() != 0;
        self
            .return_port
            .next = to_bits::<
            64usize,
        >(verilated_module.return_port() & 18446744073709551615u64);
        self.mout_oe_ram.next = verilated_module.Mout_oe_ram() != 0;
        self.mout_we_ram.next = verilated_module.Mout_we_ram() != 0;
        self.mout_addr_ram.next = to_bits::<32usize,>(
          verilated_module.Mout_addr_ram() & 4294967295u32);
        self.mout_wdata_ram.next = to_bits::<32usize,>(
          verilated_module.Mout_Wdata_ram() & 4294967295u32);
        self.mout_data_ram_size.next = to_bits::<6usize,>(
          verilated_module.Mout_data_ram_size() & 63u8);
    }
    fn connect(&mut self) {
        self.done_port.connect();
        self.return_port.connect();
        self.mout_oe_ram.connect();
        self.mout_we_ram.connect();
        self.mout_addr_ram.connect();
        self.mout_wdata_ram.connect();
        self.mout_data_ram_size.connect();
    }
    fn hdl(&self) -> Verilog {
        Verilog::Wrapper(Wrapper {
            code: r#"minmax minmax_inst(
                      .clk(clk), .reset(reset), .start_port(start_port),
                      .done_port(done_port), .return_port(return_port),
                      .Pd61(elements), .Pd62(num_elements),
                      .M_Rdata_ram(m_rdata_ram), .M_DataRdy(m_data_rdy),
                      .Mout_oe_ram(mout_oe_ram), .Mout_we_ram(mout_we_ram),
                      .Mout_addr_ram(mout_addr_ram), .Mout_Wdata_ram(mout_wdata_ram),
                      .Mout_data_ram_size(mout_data_ram_size));"#
                .into(),
            cores: "verilog generated by bambu ...",
        })
    }
}
```



```
start:
…
```

```
bb2.preheader:
…
```

```
bb2:
…
%indvars.iv.next = or i64 %indvars.iv, 1
%6 = getelementptr inbounds i32, ptr %numbers, i64 %indvars.iv
%_14 = load i32, ptr %6, align 4, !noundef !2
%7 = tail call i32 @llvm.smax.i32(i32 %_14, i32 %local_max.026)
%8 = tail call i32 @llvm.smin.i32(i32 %_14, i32 %local_min.025)
%indvars.iv.next.1 = or i64 %indvars.iv, 2
%9 = getelementptr inbounds i32, ptr %numbers, i64 %indvars.iv.next
%_14.1 = load i32, ptr %9, align 4, !noundef !2
%10 = tail call i32 @llvm.smax.i32(i32 %_14.1, i32 %7)
%11 = tail call i32 @llvm.smin.i32(i32 %_14.1, i32 %8)
%indvars.iv.next.2 = or i64 %indvars.iv, 3
%12 = getelementptr inbounds i32, ptr %numbers, i64 %indvars.iv.next.1
%_14.2 = load i32, ptr %12, align 4, !noundef !2
%13 = tail call i32 @llvm.smax.i32(i32 %_14.2, i32 %10)
%14 = tail call i32 @llvm.smin.i32(i32 %_14.2, i32 %11)
%indvars.iv.next.3 = add nuw nsw i64 %indvars.iv, 4
%15 = getelementptr inbounds i32, ptr %numbers, i64 %indvars.iv.next.2
%_14.3 = load i32, ptr %15, align 4, !noundef !2
%16 = tail call i32 @llvm.smax.i32(i32 %_14.3, i32 %13)
%17 = tail call i32 @llvm.smin.i32(i32 %_14.3, i32 %14)
…
```

```
bb4.loopexit.unr-lcssa:
…
```

```
bb2.epil:
…
%indvars.iv.next.epil = add nuw nsw i64 %indvars.iv.epil, 1
%3 = getelementptr inbounds i32, ptr %numbers, i64 %indvars.iv.epil
%_14.epil = load i32, ptr %3, align 4, !noundef !2
%4 = tail call i32 @llvm.smax.i32(i32 %_14.epil, i32 %local_max.026.epil)
%5 = tail call i32 @llvm.smin.i32(i32 %_14.epil, i32 %local_min.025.epil)
…
```

```
bb4:
…
```

Figure 26. Simplified CFG of the LLVM IR of minmax Rust speed