Rust-Based Domain-Specific Language for SFQ Circuit Design

Mebuki Oishi
The University of Tokyo
Japan
mebuki@is.s.u-tokyo.ac.jp

Sun Tanaka
The University of Tokyo
Japan
st@is.s.u-tokyo.ac.jp

Shinya Takamaeda-Yamazaki
The University of Tokyo
Japan
shinya@is.s.u-tokyo.ac.jp

ABSTRACT

Cell-based design of a single-flux-quantum (SFQ) digital circuit requires *input–output consistency*; every output signal must be consumed *only once* by the input of the following component, which is a unique constraint, unlike the traditional CMOS digital circuit design. While there are some cell libraries and simulation tools for SFQ circuit development, they do not verify the input–output consistency, and designers have significant responsibilities to ensure it manually. Additionally, designers have to carefully manage net names without unintended duplication and correct connectivity among nets in a netlist for simulations.

We propose a domain-specific language (DSL) embedded in Rust that automatically ensures the input–output consistency in the SFQ circuit by leveraging the *ownership system* of Rust. Each SFQ circuit element is represented as a function while wires are represented as instances, and the Rust compiler verifies that multiple elements do not share a single wire through the ownership system. Circuit descriptions in the proposed DSL are successfully compiled into low-level netlists for both analog and digital circuit simulations, and the DSL provides higher productivity than the conventional design flow.

1 INTRODUCTION

Single-flux-quantum (SFQ) circuits are superconducting circuits operating at cryogenic temperatures. Due to their extremely high-speed operation and low power consumption, they attract significant interest as next-generation computing devices [1].

SFQ circuits are driven by voltage *pulses* unlike CMOS circuits, which are driven by voltage *levels*. Each pulse corresponds to a single magnetic flux quantum, implying it does not spontaneously split or disappear. Thus, SFQ digital circuits must satisfy *input–output consistency*, a one-to-one correspondence between inputs and outputs. In other words, every output signal must be consumed *only once* by the input of the following component, which is a unique constraint, unlike the traditional CMOS digital circuit design.

Another characteristic of SFQ circuits is that logic gates, such as AND or NOT, are clock-synchronous, as well as flip-flops. Consequently, while SFQ circuits can achieve extremely high clock frequency through deep gate-level pipelining, it requires careful consideration of how clock signals are supplied to each logic gate [2, 3].

For higher productivity in SFQ circuit development, cell-based design [4] is preferred over full custom design using primitive circuit elements like resistors and inductors, as is done in CMOS circuit development. However, due to the pulse-driven nature, designers must ensure that the SFQ circuit satisfies input–output consistency.

There is an analog circuit simulator [5] tailored to SFQ circuit evaluation, and such analog simulator requires a netlist of an SFQ circuit described in the SPICE format as the input. However, directly describing a netlist is a significant burden for designers because a unique net name without unintended duplication should be assigned manually to each signal, and careful connectivity management is required.

To address these problems in SFQ circuit design, we propose a domain-specific language (DSL) embedded in Rust. Rather than other programming languages, we specifically use Rust to leverage its ownership system to facilitate static checking of input–output consistency. Since designing a high-performance SFQ circuit requires fine-tuning of clock timings, this DSL is designed to describe a circuit at the gate level, close to the netlist. Moreover, the DSL contributes to efficient circuit design because it can generate netlists for both digital and analog simulation models with proper net names automatically assigned.

LATTE '25, March 30, 2025, Rotterdam, The Netherlands

2 IMPLEMENTATION AND BASIC DESIGN

The DSL is provided as a Rust crate, enabling circuit descriptions to be written as fully valid Rust code. The primary data types are `Wire` and `Circuit`. Methods on `Circuit` instantiate logic gates, and `Wire` is used to connect gates. Each `Wire` can only be used once by leveraging Rust's ownership system.

The `Wire` struct internally holds only a single `String`. By wrapping it in a struct, constructor calls and cloning are restricted. The `Circuit` struct has a list of gate instances; when a gate function is invoked, a new gate instance is added to this list.

Listing 1: SFQ Component Abstraction

```rust
pub struct Wire {name: String}
enum Gates {
  And {name: String, a: String, b: String, clk: String, q: String},
  Split {name: String, a: String, q0: String, q1: String},
  ...
}
impl<const N: usize, ... > Circuit<N, M, L, O, P> {
  pub fn and(&mut self, a: Wire, b: Wire, clk: Wire, q: Option<&str
      >) -> Wire {
    let gate = Gates::And { ... };
    self.gates.push(gate);
    return Wire::new( ... );
  }
  ...
}
```

As a primary example, an SFQ-based half-adder circuit and its DSL description are shown in Figure 1 and Listing 2, respectively.

A `Circuit` instance, which corresponds to a subcircuit in SPICE and a module in Verilog, is created by invoking the `Circuit::create()` function with its input and output ports specified. At the time, `Wire` instances for the inputs (`a`, `b`, and `clk` in Listing 2) are also created.

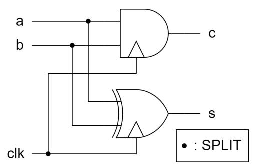

Figure 1: Half adder circuit

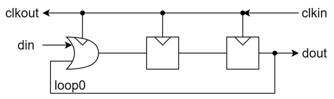

Figure 2: Counter-flow clocking circuit with a loop

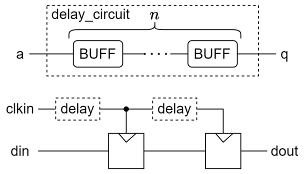

Figure 3: Parametrized delay circuit

Listing 2: DSL Description for the circuit in Figure 1

```
let inputs = ["a", "b", "clk"];
let outputs = ["c", "s"];
let (mut circuit, [a, b, clk], [], []) =
    Circuit::create(inputs, outputs, [], [], [], "HalfAdder");
let [clk1, clk2] = circuit.split(clk, None, None);
let [a1, a2] = circuit.split(a, None, None);
let [b1, b2] = circuit.split(b, None, None);
let c = circuit.and(a1, b1, clk1, Some("c"));
let s = circuit.xor(a2, b2, clk2, Some("s"));
circuit.set_outputs([c, s]);
```

Logic gates are instantiated by invoking dedicated functions on the `Circuit`. Each function takes the `Wire`s for the gate's inputs and takes ownership of them. Consequently, the `Wire` cannot be used again. The gate function then returns a newly generated `Wire` for its output, which is used as an input for the subsequent gate.

A `Wire` can only be obtained from the circuit's input port or the gates' output, and can only be used exactly once. This mechanism ensures the input–output consistency required in SFQ circuits.

The `Wire`s corresponding to the circuit's output port are passed to the `set_output()` function, which consumes their ownership. If any `Wire` remains unused, it violates the input–output consistency, and the compiler issues a warning about an unused variable.

The `Circuit` has two functions `to_spice()` and `to_verilog()`, which generate netlists in SPICE or Verilog format as strings. Conversion of the DSL code into netlists is completed by compiling and executing the code as a Rust code.

3 ADVANCED DESIGN

In the proposed DSL, circuits are described from upstream to downstream, but certain types of circuits cannot be described in this manner. These include circuits with feedback loops and circuits employing counter-flow clocking, in which the clock signal travels in the opposite direction of the data flow.

To describe a circuit with loops, `Wire`s at the upper end of the loop are created along with the `Circuit`. The `Wire`s at the lower end are identified as the upper end using the `set_loops()` function.

To describe counter-flow clocking circuits, the `CounterWire` type is introduced. The DSL provides only BUFF and SPLIT gate functions for `CounterWire`, which are used in clock lines. In contrast to standard gate functions, these functions receive the output `CounterWire` and return the input `CounterWire`. Listing 3 and Figure 2 illustrate these descriptions and show the resulting circuit structures.

The DSL enables circuits to be parametrized, which is difficult in netlists. For example, a delay circuit comprising a specified number of BUFF gates is described in Listing 4. Here, Rust's ownership system is sufficiently strong to know that the `Wire` instance in

Listing 3: DSL Description for the circuit in Figure 2

```
let (mut c, [din], [loop0], [clkout]) = Circuit::create(
  ["din"], ["dout"], ["loop0"], ["clkout"], ["clkin"], "Advanced");

let (clk, clk0) = c.counter_split(clkout, None, None);
let d = c.or(din, loop0, clk0, None);
let (clk, clk0) = c.counter_split(clk, None, None);
let d = c.dff(d, clk0, None);
let (clkin, clk0) = c.counter_split(clk, Some("clkin"), None);
let d = c.dff(d, clk0, None);
let [dout, loop0] = c.split(d, Some("dout"), Some("loop0"));
c.set_outputs([dout]);
c.set_loops([loop0]);
c.set_counter_inputs([clkin]);
```

variable `a` is consumed and created in every iteration; thus, the input–output consistency is still ensured.

A `Circuit` can be reused multiple times in another `Circuit` using the `subcircuit()` function. The function takes the reused `Circuit` and `Wire`s for inputs, then returns `Wire`s for outputs. The number of inputs and outputs is statically checked with the circuit type, which includes the port counts.

Listing 4: DSL Description for the circuit in Figure 3 ($n = 5$)

```
fn delay_circuit(n: u32) -> Circuit<1, 1, 0, 0, 0> {
    let name = format!("delay{}", n);
    let (mut c, [mut a], [], []) =
    Circuit::create(["a"], ["q"], [], [], [], &name);
    for i in 0..n {
        let name = if i == n - 1 { Some("q") } else { None };
        a = c.buff(a, name);
    }
    c.set_outputs([a]);
    return c;
}
fn main() {
    let delay5 = delay_circuit(5);
    let (mut c, [d, clk], [], []) = Circuit::create(["din", "clk"],
          ["dout"], [], [], [], "main");
    let ([clk], []) = c.subcircuit(&delay5, [clk], [], [None], []);
    let [clk, clk1] = c.split(clk, None, None);
    let d = c.dff(d, clk1, None);
    let ([clk], []) = c.subcircuit(&delay5, [clk], [], [None], []);
    let d = c.dff(d, clk, Some("dout"));
    c.set_outputs([d]);
    println!("{}", c.to_spice().join("\n"));
}
```

4 FUTURE WORK

The current DSL supports only cell-based bit-level representations of SFQ circuits. For higher productivity, supporting multi-bit signals and corresponding operations will be an essential feature improvement. Unlike CMOS circuits, careful management of clock signals, such as clock reach ordering to multiple gates [3], is mandatory. We will develop the enhancement for automatic verification of clock-related constraints.

REFERENCES

[1] Konstantin K Likharev and Vasilii K Semenov. RSFQ logic/memory family: A new Josephson-junction technology for sub-terahertz-clock-frequency digital systems. *IEEE Transactions on Applied Superconductivity*, 1(1):3–28, 1991.

[2] Koki Ishida, Ilkwon Byun, Ikki Nagaoka, Kosuke Fukumitsu, Masamitsu Tanaka, Satoshi Kawakami, Teruo Tanimoto, Takatsugu Ono, Jangwoo Kim, and Koji Inoue. SuperNPU: An extremely fast neural processing unit using superconducting logic devices. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 58–72. IEEE, 2020.

[3] Kazuyoshi TAKAGI, Nobutaka KITO, and Naofumi TAKAGI. Circuit Description and Design Flow of Superconducting SFQ Logic Circuits. *IEICE Transactions on Electronics*, E97.C(3):149–156, 2014. doi: 10.1587/transele.E97.C.149.

[4] Lieze Schindler, Johannes A. Delport, and Coenrad J. Fourie. The ColdFlux RSFQ Cell Library for MIT-LL SFQ5ee Fabrication Process. *IEEE Transactions on Applied Superconductivity*, 32(2):1–7, 2022. doi: 10.1109/TASC.2021.3135905.

[5] Johannes Arnoldus Delport, Kyle Jackman, Paul le Roux, and Coenrad Johann Fourie. JoSIM—Superconductor SPICE Simulator. *IEEE Transactions on Applied Superconductivity*, 29(5):1–5, 2019. doi: 10.1109/TASC.2019.2897312.