Logic simulation tool for RSFQ circuits accepting arrivals of multiple pulses in a clock period

Nobutaka Kito¹, Shohei Udatsu¹, and Kazuyoshi Takagi²
¹ Department of Information Engineering, Chukyo University, Toyota, Aichi, Japan
² Graduate School of Engineering, Mie University, Tsu, Mie, Japan
E-mail: nkito@sist.chukyo-u.ac.jp

Abstract. A logic simulation tool for RSFQ circuits is proposed. It can treat RSFQ logic circuits containing signal lines accepting multiple pulses in a clock period. Logic circuits can be implemented in compact area by utilizing RSFQ specific gates and feeding multiple pulses into a gate input in a clock period. For treating circuits containing signal lines accepting multiple pulses in a clock period, a new description method is proposed. A logic simulation tool treating netlists described based on the proposed method was implemented. Examples of designs having signal lines accepting multiple pulses to be implemented in compact area are shown, and simulation results obtained with the tool are shown.

1. Introduction
RSFQ circuits [1] and their energy-efficient derivatives such as eSFQ [2], ERSFQ [3], and LV-RSFQ [4] are expected to realize high performance and energy-efficient computers.

RSFQ circuits use pulse logic. Namely, voltage pulses are used for calculating logic functions. The target frequency of RSFQ logic circuits is very high. Therefore, special considerations are necessary in circuit design and design automation optimized for RSFQ logic circuits is crucial. Design automation methods and tools have been developed [5] such as a technology mapping method [6], automatic layout methods [7, 8] and a design for testability method [9]. In this paper, we examine aggressive use of pulse, which was not supposed in those methods and tools.

Logic simulation of an RSFQ logic circuit is mainly carried out after designing a layout because the order of pulse arrivals at each gate affects its logic function and the order is determined in the layout design phase. In the design flow with CONNECT cell library [10], a traditional HDL netlist is extracted from a layout and logic simulation is performed mainly with it. Though design exploration in logic design phase is important to obtain a suitable design for its requirements, it is hard to evaluate various logic designs with this design flow. Methods to simulate a logic design without considering layout design are crucial.

To tame with the problem, a description method of RSFQ logic circuits was proposed in [11, 12]. In the method, the order of pulse arrivals is annotated for each gate to represent the circuit behavior explicitly. A logic simulation tool for circuit descriptions based on the method has also been shown. However, the method and the tool were designed for simple designs without complex ordering. Designs with complex order of pulse arrivals, i.e., arrivals of multiple pulses at a gate input in a clock period, cannot be represented with the method because it assumes a gate input receives at most one pulse in a clock period.
Logic functions can be implemented with fewer logic gates by utilizing complex ordering of pulse arrivals. For example, a logic function \(a \oplus b + c\) can be realized simply with a confluence buffer (CB) and a resettable D flip-flop (RDFF) having data, reset, and clock input terminals. Pulses of “a” and “c” are merged with a CB and are fed to the data terminal of an RDFF and a pulse of “b” is designed to be fed to its reset terminal between the pulse arrivals of “a” and “c.” This design cannot be described with the method in [11, 12] because at most two pulses appear in the data terminal in a clock period.

The method to be proposed is an extension of the description method to treat designs with the complex ordering of pulse arrivals, i.e., arrivals of multiple pulses at a gate input in a clock period. In the proposed method, we attach a list of integers or an integer for each gate input to represent the order of pulse arrivals. The integers in the list or the integer represent the order of each pulse arriving at the gate input.

We implemented a logic simulation tool. For logic simulation with the tool, we prepare a description of an RSFQ circuit in a description language based on the proposed description method. To ease describing circuits, the description language has some syntaxes such as module definition. We can describe netlists of RSFQ circuits with complex orders of pulse arrivals and we can perform simulation of them with the tool.

This paper is organized as follows. In the next section, the circuit description method of the previous research is briefly reviewed. In Section 3, an extension of the circuit description method for treating multiple pulses is proposed. In Section 4, the implemented logic simulation tool which employs a description language based on the proposed method is described. In Section 5, logic simulation results obtained with the implemented tool are shown. In Section 6, this paper is concluded.

2. Preliminaries
In RSFQ logic circuits, the behavior of a gate varies depending on the order of pulse arrivals to its input terminals. In an RSFQ logic gate, a pulse that arrives at a data input terminal after a pulse for its clock terminal is held for the next clock period, and the data input terminal works as if it has latching function. Therefore, it is important to represent the order of pulse arrivals of each gate.

A circuit description method of RSFQ circuits is proposed in [11, 12]. In the description method of those papers, a gate in a circuit is represented as follows.
\[
o = GATE \ G(w_1@p_1, \ w_2@p_2, \ldots, \ w_k@p_k);
\]
In the line, \(GATE\) is a primitive logic function, such as AND, OR, and XOR. \(G\) is the instance name of the gate. Each pair of a signal name and an integer described as \(w_i@p_i\) corresponds to an input of the gate. \(w_i\) and \(p_i\) (\(1 \leq i \leq k\)) are the signal name for the \(i\)-th input of the gate and the integer to represent the order of pulse arrivals, respectively. \(o\) is the output of the gate. When \(p_i\) is smaller than \(p_j\), it means the order that the pulse on \(w_i\) precedes the pulse on \(w_j\). Note that the order of gate inputs in parenthesis is determined in advance for each primitive logic function. In this paper, we describe inputs for data input terminals at first and describe an input for a clock terminal at last.

We show two circuits in Figure 1 as examples. Figures 1(a) and (b) show a circuit employing clock-follow-data clocking and one employing concurrent-flow clocking [13], respectively. In circuits employing clock-follow-data clocking, a clock pulse for a gate arrives after data pulses for the gate. In Figure 1(a), the value attached to “clock” of instance “g1” is greater than those attached to “x” and “y”. The same attached value for “x” and “y” means that no order is given for those inputs. In circuits employing concurrent-flow clocking, a clock pulse for a gate arrives before data pulses for the gate. In Figure 1(b), the value attached to “clock” of instance “g1” is smaller than those attached to “x” and “y”.

2
Figure 1. Examples of RSFQ logic circuits and their corresponding netlists.

Figure 2. Example of an RSFQ logic circuit which cannot be described with the method in [11, 12] (a), behavior of RDFF gate (b), and a naive design of the same logic function (c).

3. Circuit description method of RSFQ circuits accepting multiple pulses in a clock period

The circuit description method in [11, 12] can treat basic RSFQ logic circuits. However, it is not possible to describe netlists of RSFQ circuits utilizing pulse logic extensively. In other words, the method cannot treat signal lines accepting multiple pulses in a clock period.

Area efficient circuits can be derived utilizing pulse logic. For example, the circuit in Figure 2(a) calculates logic function \( a \overline{b} + c \) except the combination of \((a, b, c) = (1, 0, 1)\). Figure 2(b) shows the behavior of the resettable D flip-flop (RDFF) used in the circuit. The confluence buffer (CB) in the circuit converges pulses fed from “a” and “b” and multiple pulses may appear in its output. Figure 2(c) shows a naive design to calculate the same function and this design contains three logic gates and three D flip-flops. It is larger than the circuit in Figure 2(a). The circuit in Figure 2(a) assumes that a pulse fed to “a” arrives at the RDFF at first, then a pulse fed to “b” arrives, and a pulse fed to “c” arrives finally. Describing a signal line accepting multiple pulses is not considered in [11, 12]. There is no way to describe the circuit as a netlist.

In this paper, to treat RSFQ logic circuits containing signal lines accepting multiple pulses in a clock period, we propose the following circuit description method as an extension of the method in [11, 12],

\[
o = GATE \ G(w_1 @[p_{1,1}, \ldots, p_{1,n_1}], \ldots, w_k @[p_{k,1}, \ldots, p_{k,n_k}]);
\]

To describe the order of multiple pulses, we notate the order of pulse arrivals as lists of integers. If at most \( n_i \) pulses are on \( w_i \) in a clock period, the length of the list for \( w_i \) needs to be \( n_i \).
The \( i \)-th integer in the list represents the order for the \( i \)-th pulse of the input. We can use an integer in place of a list like the previous method when the number of pulses arriving at the input terminal is one, i.e. \( n_i = 1 \).

By introducing the extension, we can describe circuits containing signal lines accepting multiple pulses as netlists. As an example of the extended description method, we show the netlist for the circuit of Figure 2(a) in Figure 3. In the netlist, we attach an integer to each input of the CB. A pulse fed to “a” is converged to signal line “t” at first, then the a pulse fed to “c” is converged to the signal line. In line 2 of Figure 3, we attach a list for “t” to represent the order of pulse arrivals for the RDFF. The former integer in the list corresponds to the pulse fed from “a”, and the latter one corresponds to the pulse fed from “c”.

4. Logic simulation tool

We implemented a logic simulation tool\(^1\) that can treat netlists based on the proposed method. The tool is written with Python. It is easy to implement further extensions for the tool to treat specific needs of RSFQ logic simulation with utilizing rich Python packages. Though Python programs are considered to be slower than C/C++ programs, circuit size of RSFQ circuits is not very large currently and the execution time of logic simulation is fast. The tool employs a simple hardware description language based on the proposed description method. For logic simulation of an RSFQ logic circuit, input patterns for the circuit inputs are necessary. Thus, for the tool, we prepare a circuit description containing input patterns for the circuit in addition to a netlist of the circuit. The tool prints out the output patterns of the circuit corresponding to the given input patterns.

In Figure 4, we show an example of a circuit description for Figure 2(a). In the description, circuit inputs, a circuit output, and an internal signal line are defined with “input” statements, an “output” statement, and a “wire” statement, respectively. For each “input” statement, we attach a list of input values for the circuit input. Each element of the list is “1” or “0”. “1” and “0” represent the existence of a pulse and the absence of a pulse, respectively. In the description, all elements of the list for circuit input “clock” are “1” because we feed a pulse to the input

\(^1\) The tool is available on [http://nkito.sakura.ne.jp/logicsim/](http://nkito.sakura.ne.jp/logicsim/).
Figure 5. Circuit calculating $r_1$, $r_1r_2$, and $r_1r_2r_3$ and its circuit description.

```verilog
module PE(s_out, L_out, X_out, R_out)(y_in, s_in, L_in, X_in, R_in, clock);
  wire Y, PP, S0, C0, S1, C1T, C1;
  Y = ND G1_1( L_in@0, y_in@1, L_in@2 );
  PP = ND G2_1( L_in@0, Y@1, X_in@2 );

  S0 = XOR G3_1( PP@0, s_in@0, clock@1 );
  C0 = AND G4_1( PP@0, s_in@0, clock@1 );

  s_out = XOR G5_1( S0@0, C1@0, clock@1 );
  C1T = AND G6_1( S0@0, C1@0, clock@1 );
  C1 = CB G7_1( C1T@0, C0@0 );

  L_out = D G8_1( L_in@1, clock@0 );
  X_out = D G10_1( X_in@1, clock@0 );
  R_out = D G11_1( R_in@1, clock@0 );
endmodule
```

Figure 6. Netlist of a module for the processing element of the bit-serial multiplier in [14].

every clock period.

We show another example containing a signal line accepting multiple pulses in Figure 5. The circuit calculates $w_1 = r_1$, $w_2 = r_1r_2$, and $w_3 = r_1r_2r_3$ with utilizing non-destructive read-outs (NDROs). We feed a clock pulse to “set” terminals of NDROs to set their internal states, initially. Then, we feed data pulses to their “rst” terminals. Each NDRO outputs a pulse according to its internal state once it receives a pulse on its “clk” terminal. In the circuit description of the example, we use an “assign” statement to represent the connection between $r_1$ and $w_1$.

To ease describing a circuit, defining circuit modules is possible in the description language. As an example, we show a description of a module in Figure 6. The module corresponds to the processing element of the bit-serial multiplier in [14]. A definition of a module begins with “module” and ends up with “endmodule”. Outputs and inputs of a module are described in the former parentheses and the latter parenthesis in a “module” statement, respectively.

The circuit description of the 4-bit bit-serial multiplier based on [14] using the module definition is shown in Figure 7. The module is instantiated in lines 25, 28, 31, and 34. The connection between the outputs of an instance and signal lines is represented with the left-hand side of each instantiation statement.
# input and input pattern definitions
input reset_in [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
input load_in [0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0];
input y_in [0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0];
input x_in [0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,0,0,0,0,0];
input s_in [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
input clock [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1];

# output definitions
output reset_out ;
output load_out ;
output y_out ;
output x_out ;
output s_out ;

# definition of module PE in Figure 5 is omitted here.

# circuit definition
{

# wire definitions
wire L1, X1, R1, s_out1, s_in1;
wire L2, X2, R2, s_out2, s_in2;
wire L3, X3, R3, s_out3, s_in3;

(s_out1, L1, X1, R1) = PE PE0 (y_in, s_in, load_in, x_in, reset_in, clock);
s_in1 = NDRO ND0 ( R1@2, L1@2, s_out1@3);

(s_out2, L2, X2, R2) = PE PE1 (y_in, s_in1, L1, X1, R1, clock);
s_in2 = NDRO ND1 ( R2@2, L2@2, s_out2@3);

(s_out3, L3, X3, R3) = PE PE2 (y_in, s_in2, L2, X2, R2, clock);
s_in3 = NDRO ND2 ( R3@2, L3@2, s_out3@3);

(s_out, load_out, x_out, reset_out) = PE PE3 (y_in, s_in3, L3, X3, R3, clock);
assign y_out = y_in;
}

Figure 7. Description for the 4-bit bit-serial multiplier based on [14].

For exploring designs, “for” statement for generating variable numbers of modules or gates is available in the tool. Figure 8 is a description utilizing the statement for the bit-serial multiplier. In the description, we describe the definition of constant “WIDTH” and the width of signals is defined with it. By utilizing constant definition and “for” statement, we can easily change the bit-width of signals and the number of modules in a circuit and we can easily examine logic designs.

5. Simulation results
We show simulation results of the descriptions shown in the previous section with the implemented tool. At first, we show the logic simulation results of the descriptions of Figures 4 and 5 in Figures 9 and 10, respectively. As shown in the figures, the tool prints out the patterns of the circuit outputs and it also prints out the patterns of the internal signal lines. Because signal line “t” in both two circuits may contain multiple pulses in a clock period, the value of each clock period is represented as a list displayed with parentheses.

We also show the logic simulation result of the description of Figure 7 for the 4-bit bit-serial multiplier in Figure 11. We omitted patterns of the internal signal lines in the figure. Because the internal state of each gate is unknown initially, the output values of a circuit may be unknown. In the figure, those unknown values are depicted with “x”. Note that we also use the error value...
# definition of a constant
const WIDTH = 4;

# definitions of inputs and outputs and module PE are omitted here.

# circuit definition
{
    # wire definitions
    wire L[0:WIDTH-1], X[0:WIDTH-1], R[0:WIDTH-1];
    wire sout[1:WIDTH-1], sin[0:WIDTH-1];

    genvar i;

    assign sin[0] = s_in;
    assign L[0] = load_in;
    assign X[0] = x_in;
    assign R[0] = reset_in;

    for u0 (i=0; i < WIDTH-1; i += 1 ){
        (sout[i+1], L[i+1], X[i+1], R[i+1]) =
        PE PE0(y_in, sin[i], L[i], X[i], R[i], clock);
        sin[i+1] = NDRO ND0(R[i+1]@2, L[i+1]@2, sout[i+1]@3);
    }

    (s_out, load_out, x_out, reset_out) =
    PE PE3(y_in, sin[WIDTH-1], L[WIDTH-1], X[WIDTH-1], R[WIDTH-1], clock);
    assign y_out = y_in;
}

Figure 8. Description for the bit-serial multiplier utilizing “for” statement.

| Wires | t | : | [0, 0], [1, 0], [0, 0], [1, 0], [0, 1], [0, 1], [1, 1] |
|-------|---|---|--------------------------------------------------|
| Inputs| a | : | 0, 1, 0, 1, 0, 0, 1 |
|       | b | : | 0, 0, 1, 1, 0, 1, 1 |
|       | c | : | 0, 0, 0, 0, 1, 1, 1 |
|       | clock | : | 1, 1, 1, 1, 1, 1, 1 |
| Outputs| g | : | 0, 1, 0, 0, 1, 1, 1 |

Figure 9. Simulation result of the description of Figure 4.

| Wires | t | : | [0, 0], [0, 1], [1, 0], [1, 1], [0, 0], [0, 1], [1, 0], [1, 1] |
|-------|---|---|--------------------------------------------------|
| Inputs| r1 | : | 0, 1, 0, 1, 0, 0, 1 |
|       | r2 | : | 0, 0, 1, 1, 0, 1, 1 |
|       | r3 | : | 0, 0, 0, 0, 1, 1, 1 |
|       | clock | : | 1, 1, 1, 1, 1, 1, 1 |
| Outputs| w1 | : | 0, 1, 0, 1, 0, 0, 1 |
|       | w2 | : | 0, 0, 1, 0, 0, 1, 0 |
|       | w3 | : | 0, 0, 0, 1, 0, 0, 0 |

Figure 10. Simulation result of the description of Figure 5.

"-" in the logic simulation tool. For example, the error value is used for representing the output value of gates receiving forbidden inputs. Feeding multiple pulses for an input of basic logic gates in a clock period is forbidden. We use the error value "-" for the output of those gates. The tool uses four values in logic simulation.

In the figure, the circuit performs two multiplications, \([1111]_2 \times [1101]_2(=[11000011]_2)\) and \([1010]_2 \times [1110]_2(=[10001100]_2)\). From the figure, we can see that the truncated results...
[1100]_2 × 2^3 and [10001]_2 × 2^3 are obtained. By using the tool, we can examine and verify the complex design with the simple description without performing layout of the design.

6. Conclusion
We proposed a new description method of RSFQ circuits and a logic simulation tool. The proposed description method is an extension of our previous description method to describe designs containing signal lines accepting multiple pulses in a clock period. We implemented a logic simulation tool. The tool can treat circuit descriptions written in a description language based on the proposed method. With the method and the tool, we can treat RSFQ circuits utilizing pulse logic extensively as netlists and we can examine the behavior of them easily.

Acknowledgement
This work was supported by JSPS KAKENHI Grant Numbers JP19K11888 and JP18K11213.

References
[1] Likharev K and Semenov V 1991 IEEE Transactions on Applied Superconductivity 1 3–28
[2] Volkmann M H, Sahu A, Fourie C J and Mukhanov O A 2012 Supercond. Science and Technology 26 015002
[3] Kirichenko D E, Sarwana S and Kirichenko A F 2011 IEEE Transactions on Applied Superconductivity 21 776–79
[4] Tanaka M, Kitayama A, Koketsu T, Ito M and Fujimaki A 2013 IEEE Transactions on Applied Superconductivity 23 1701104
[5] Fourie C J 2018 IEEE Transactions on Applied Superconductivity 28 1300412
[6] Pasandi G and Pedram M 2019 IEEE Transactions on Applied Superconductivity 29 1300114
[7] Shahsavani S N, Lin T, Shafaei A, Fourie C J and Pedram M 2017 IEEE Transactions on Applied Superconductivity 27 1302008
[8] Kito N, Takagi K and Takagi N 2018 IEEE Transactions on Applied Superconductivity 28 1300105
[9] Krylov G and Friedman E G 2017 IEEE Transactions on Applied Superconductivity 27 1302307
[10] Yorozu S, Kameda Y, Terai H, Fujimaki A, Yamada T and Tahara S 2002 Physica C: Superconductivity 373-378 1471–74
[11] Kito N, Takagi K and Takagi N 2012 Proc. 17th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI2012) 319–24
[12] Takagi K, Kito N and Takagi N 2014 IEICE Transactions on Electronics E97-C 149–56
[13] Gaj K, Friedman E G and Feldman M J 1997 Journal of VLSI signal processing systems 16 247–76
[14] Hara H, Obata K, Park H, Yamanashi Y, Taketomi K, Yoshikawa N, Tanaka M, Fujimaki A, Takagi N, Takagi K and Nagasawa S 2009 IEEE Transactions on Applied Superconductivity 19 657–60

Figure 11. Simulation result of the description of the 4-bit bit-serial multiplier in Figure 7.