State Access for RSFQ Test and Analysis

Marly Roncken, *Member, IEEE*, Ebelechukwu Esimai, *Graduate Student Member, IEEE*, Vivek Ramanathan, Warren A. Hunt Jr., and Ivan Sutherland

(Invited Paper)

***Abstract*—We present means to initialize, to propagate, and to examine states in an RSFQ circuit that are useful for design as well as for functional test and analysis. Our RSFQ test strategy distinguishes states by the information they carry from computation to computation, and saves costs by ignoring information-free states. To start, stop, and stall operations that are asynchronous, we developed a new variety of RSFQ stateholder, called *MrGO* after its CMOS counterpart. We include two simulated examples, a clocked pipelined adder for which we test functionality, and an asynchronous ring FIFO for which we analyze throughput.**

***Index Terms*—RSFQ circuit, initialization, test and analysis, information-free state, information mobility, asynchronous.**

I. Introduction

STATES are so abundant in Rapid Single Flux Quantum (RSFQ) circuits [2], [12], [22] that it may be too expensive to access them all for functional test and analysis [1]. In this paper, we distinguish states by (1) the information they carry from computation to computation, and (2) the mobility of that information. We avoid test access to information-free states that change temporarily during a computation but remain the same between computations. We judiciously use nondestructive state readouts to create *fixed* rather than *moving* test targets. To support design, test, and analysis of asynchronous circuits, we introduce new varieties of stateholders that provide the flow control necessary to (1) start and stop an operation, and (2) stall an operation until progress conditions are met.

We simulate two examples in an (R)SFQ MIT Lincoln Laboratory process geometrically equivalent to 270 nm CMOS. The clocked pipelined adder in Fig. 1 provides test access *to* the latch states that store to-add or added data but *not to* the information-free states in the adder logic. The asynchronous ring FIFO in Fig. 3 uses the new stateholders to implement its protocols and fix locations for initialization and test access.

Manuscript received 9 November 2022; revised 16 February 2023; accepted 20 February 2023. Date of publication 2 March 2023; date of current version 21 April 2023. This work was supported in part by private sponsors through the Portland State University Foundation and in part by the Mayo Clinic under Grant SPPDG-052 for "Computing Systems Based on the Link-Joint Paradigm." *(Corresponding author: Marly Roncken.)*

Marly Roncken, Ebelechukwu Esimai, and Ivan Sutherland are with the Asynchronous Research Center, Maseeh College of Engineering and Computer Science, Portland State University, Portland, OR 97207 USA (e-mail: marly.roncken@gmail.com).

Vivek Ramanathan and Warren A. Hunt Jr. are with the Department of Computer Science, The University of Texas at Austin, Austin, TX 78712 USA.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TASC.2023.3251949.

Digital Object Identifier 10.1109/TASC.2023.3251949

One can read this paper two ways. The main text serves as storyline for the examples and key ideas, and the motivations and strategy behind them. Figures and captions give enough detail on circuits and simulations for replication.

II. Terminology and Notation

The examples in this paper are partitioned into *Links* that store and transfer information, and *Joints* that compute *on* and control the flow *of* information. We use a similar partition in our CMOS designs [7], [17], [18], [19]. The dual-rail data encoding in the adder example uses two signals per bit, called *T* and *F*. A pulse comes into either the *T* bit signal or the *F* bit signal but never both. A pulse on *T*, *F*, or neither, denoted as *[T,F]=[1,0]*, *[0,1]*, or *[0,0]*, represents value 1, 0, or *nodata*, respectively. All current values in this paper are given in normalized units, *Inorm*, of 125e-6 A. All inductance values are given in normalized units, *Lnorm*, of 2.632e-12 H. In our RSFQ circuits, we mark grounded Josephson junctions with symbol ⊙ and flying ones with symbol ×.

III. Example 1: Clocked Pipelined Adder

Pipelining is attractive for streaming long arithmetic calculations at high throughput. The pipelined adder in Fig. 1 adds 2-bit numbers, *x* and *y*, to 1-bit carry input, *c*, and generates 3-bit sum output, *s*. The first and leftmost pipeline stage adds the least significant *x*, *y* bits to *c* and generates a 1-bit carry output that serves as carry input for the second and rightmost stage, which adds the most significant bits. By increasing the number of stages, we can add numbers with increasingly more bits. The pipeline stages operate in parallel. While the second stage adds the most significant bits, the first stage adds the least significant bits of the next *x*, *y* to the next *c*. A demonstration follows in the first simulation half of Fig. 2, where we show two consecutive *x*, *y*, and *c* additions.

Links store data to be added or already added. Because these data change with each addition, Links make superb entry and exit points for testing functionality of the adder logic, located in ADD Joints. To this end, we store data in D2 latches [12], which have two separate input and two separate propagate-and-output options for normal versus test operations — see Fig. 1(b). Joint states are information-free, changing during but not between ADD operations — see Appendix. As a result, we can initialize and test the Joints by controlling and observing Link latches. Test access inside ADD Joints is unnecessary. A demonstration follows in the second simulation half of Fig. 2, where we proffer

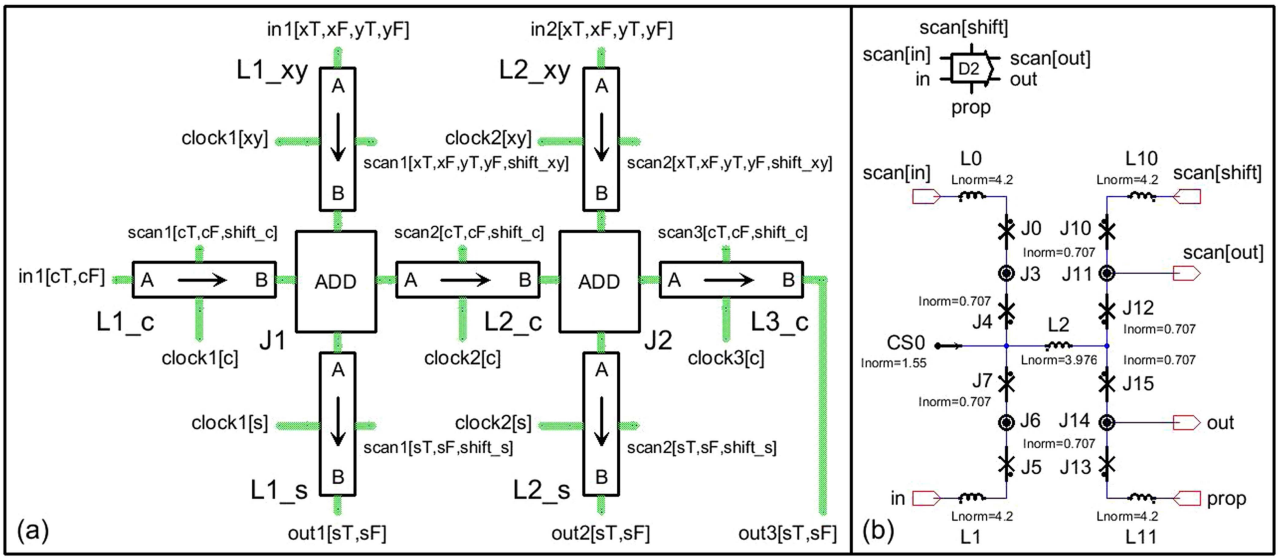

Fig. 1. Clocked adder with two pipeline stages (a). The first stage has two input Links, *L1_xy* and *L1_c*, that store 1-bit dual-rail encoded data that they transfer by means of a clock to an ADD Joint, *J1*, which adds the bits. The Joint sends the resulting sum and carry to two output Links, *L1_s* and *L2_c*, that store the 1-bit dual-rail results for later transfer. The second stage is similar, and consists of input Links *L2_xy*, *L2_c*, ADD Joint *J2*, and output Links *L2_s*, *L3_c*. Normally, the first stage acts first. It adds a 1-bit carry input to the least significant bits of 2-bit data inputs x, y. The second pipeline stage must wait until the first stage has generated a carry bit for it to add to the two most significant x, y bits. The Links use D2 latches to store their data. Panel (b) shows the RSFQ circuit and icon for a D2 latch. During normal operation, the data in a D2 latch come and go through the Link. However, for initialization, test, and analysis, we can redirect D2 data to come from or go to a scan chain. The picture omits a global *clock* signal and its fork connections to local Link clocks and D2 *prop* signals. Also omitted are a global *scanshift* signal and its fork connections to the local D2 *scan[shift]* signals in each Link. Both are present in the simulation for Fig. 2. Absent from our simulation are serial scan chain connections between D2 *scan[in]* and *scan[out]* signals. The RSFQ circuit for Joint ADD is based on Patra et al. [16], and replicated in the Appendix. Note that ADD has neither a clock nor scan test access.

two consecutive D2-stored test inputs to both Joints, and observe their D2-stored test responses.

We can extend this approach to larger parts, e.g., test the 2-stage adder as one part and avoid test access to Joints *J1*, *J2* and Link *L2_c* whose states are information-free for 2-stage additions. Testing larger parts increases test time but decreases circuit area, especially when parts have many internal Links.

The ADD Joints use self-timed dual-rail data and are therefore clockless. The Links use local clocks that connect to a global *clock* to propagate data simultaneously to and from each pipeline stage. The maximum *clock* frequency might decrease if we increase the number of pipeline stages — a problem absent in fully asynchronous and globally asynchronous locally synchronous (GALS) designs.

IV. Example 2: Asynchronous Ring FIFO

First-In-First-Out (FIFO) buffers are useful for maintaining throughput between system parts operating at different speeds and for on-chip at-speed test runways [8], [14]. The ring FIFO in Fig. 3 copies data from Link to Link around the ring. As in the adder in Fig. 1, the Links store their data in D2 latches. But unlike the adder, the ring FIFO lacks a *clock* that we can stop to obtain exclusive latch access for scanning test data in or out. In the ring FIFO, we obtain exclusive test access by stopping one or more Joints. For this purpose, each Joint has a *go-nogo* gate, called *MrGO* and pronounced "Mister GO" [19].

A. MrGO and Related Stateholders

The purpose of MrGO is to enable or disable Joint action. In RSFQ, MrGO combines a nondestructive readout (NDRO), called *STATE*, with a gate called *SYNC* that combines a rendezvous and latch (see Figs. 4–5(a)). The RSFQ behavior of MrGO can be specified with pulse logic—a Boolean logic where *true* indicates the presence and *false* the absence of a pulse. We specify our gates using guarded commands [5], but a finite state machine diagram [12] works equally well. The guarded commands for MrGO combine those for STATE and SYNC following next (parameters are as in Fig. 4).

STATE (*set state*, *reset_state*, *read_state*, *ans_state*, s_1)
  set_state $\rightarrow$ s_1:=*true*; *set_state:=false*
  reset_state $\rightarrow$ s_1:=*false*; *reset_state:=false*
  read_state $\rightarrow$ *ans_state:=*s_1; *read_state:=false*
SYNC (*R*, *killR*, *goneR*, *ansA*, *ansB*, s_1)
  s_2 : bool
  R $\rightarrow$ s_2:=*true*; *R:=false*
  killR $\rightarrow$ *goneR:=*s_2; s_2:=*false*; *killR:=false*
  $s_1 \wedge s_2$ $\rightarrow$ *ansA:=true*; *ansB:=true*; s_2:=*false*

Guarded commands "guard $\rightarrow$ command" execute atomically, in mutual exclusion, and only when their guard is valid [5].

B. Turn-Taking in the Link-Joint Protocol

In good conversations, one listens while the other speaks. As the conversation progresses, listener and speaker take turns. Following good conversation practice, the asynchronous Link-Joint protocol lets Joints take turns updating the one-to-one Links connecting them. Each Link keeps track of whose turn it is, and shares this information with both Joints [7].

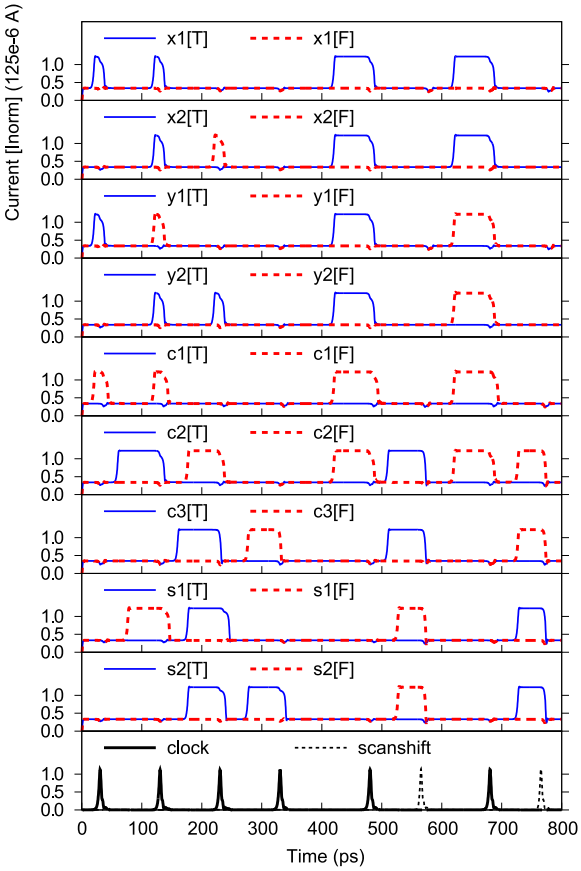

Fig. 2. Simulated results for the clocked pipelined adder of Fig. 1.

- From 0 to 400 ps, the adder performs *x1x2* + y1y2 + *c1* twice, using a 10 GHz *clock*, adding 11+11+0 (3+3+0) then 10+01+0 (1+2+0) with sum *s1s2c3*=011 (6), 110 (3), in binary (decimal) values, least significant bit left as in Fig. 1.
- Upon the first clock pulse, the first pipeline stage adds the dual-rail encoded least significant bits of the first pattern, *x1[T,F]*, *y1[T,F]*, *c1[T,F]*, stored in D2 latches of Links *L1_xy*, *L1_c*. The waveforms show the current through the corresponding quantizing inductor, *L2* (see Fig. 1(b)). Before the second clock pulse, sum and carry results are stored in the D2 latches of Links *L1_s*, *L2_c* as *s1[T,F]*, *c2[T,F]*. Note that dual-rail (decimal) values *x1[T,F]=[1,0] (1)*, *y1[T,F]=[1,0] (1)*, *c1[T,F]=[0,1] (0)* yield expected sum value *s1[T,F]=[0,1] (0)*. In the first clock period, the second pipeline stage remains idle, because it receives *nodata* for dual-rail inputs *x2[T,F]*, *y2[T,F]*, and *c2[T,F]*.
- Upon the second clock pulse, the most significant bits of the first pattern are available in *L2_xy*, *L2_c*, for addition by the second pipeline stage, which yields *s2[T,F]=[1,0] (1)*, *c3[T,F]=[1,0] (1)*. Meanwhile, the first stage adds the least significant bits of the second pattern, yielding: *s1[T,F]=[1,0] (1)*.
- Upon the third clock pulse, the second pipeline stage adds the most significant bits of the second pattern, yielding: *s2[T,F]=[1,0] (1)* and *c3[T,F]=[0,1] (0)*. The first stage, having finished its patterns, receives *nodata* and is therefore idle. The fourth clock pulse clears all D2 latches for the next test.
- From 400 to 800 ps, two scan patterns test two single-bit additions *xi*+*yi*+*ci* (1+1+0 then 1+0+0) in both pipeline stages (*i=1..2*), simultaneously, using 5 GHz *clock* and *scanshift* signals. We use *scanshift* to clear D2 latches before scanning in new data, which is crucial for *c2[T,F]* that serve as test output for the first pipeline stage and as test input for the second stage.

Connections occur at ports. Each Link transfers data from its port *A* to its port *B*, while each COPY Joint copies data from its port *in* to its port *out*. Link port *A* is compatible with COPY port *out*, and Link port *B* with COPY port *in*.

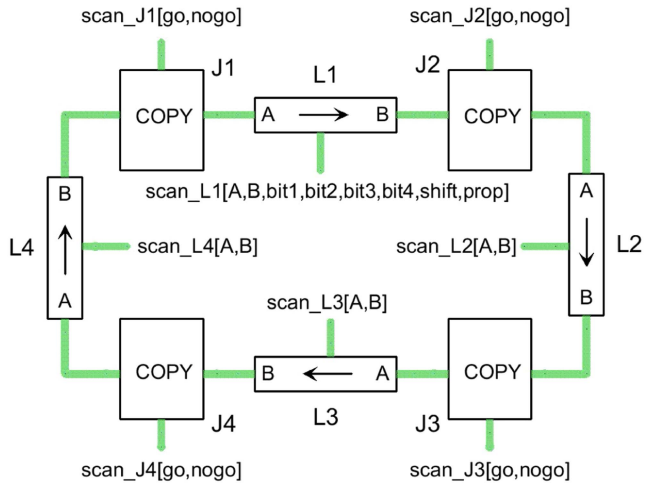

Fig. 3. Asynchronous ring FIFO with storage capacity of up to four 4-bit data items. The protocols exchange data for space. The ring FIFO can be idle (1) for lack of data, when it stores zero data items, or (2) for lack of space, when it stores four data items, or become idle (3) because we disable some Joint to act. When not idle, the ring circulates data at full-native speed. Four Links store and transfer data. Four COPY Joints manipulate data and flow control by copying incoming to outgoing data. Data flow in the direction of the Link arrows. Scan test connections allow us to control and observe (1) protocol settings in each Link, (2) *go–nogo* settings in each Joint, and (3) data in Link *L1*.

The RSFQ Link implementation in Fig. 5(b) uses two instances of STATE of Fig. 4(a), *TURNA* and *TURNB*, to track and share turn information. *TURNA* shares *A*'s turn via *A[s]* and *A[me]*. *TURNB* shares *B*'s turn via *B[s]* and *B[me]*. Signals *A[s]*, *A[me]*, *B[s]*, and *B[me]* are shared with the corresponding Joint ports, e.g., *A[s]* and *B[me]* of Link *L1* in Fig. 3 alias *out[s]* of *J1* and *in[me]* of *J2*, respectively.

The RSFQ Joint implementation in Fig. 5(a) connects stateholder *TURN*, a reduced version of SYNC of Fig. 4(b), to *MrGO*, discussed in Section IV-A. *TURN* waits for turns on *in* and *out* before triggering *MrGO* to relinquish both turns.

Having two STATE gates in the Link makes it possible to keep related STATE (Link.*TURNA* or Link.*TURNB*) and SYNC (Joint.*TURN*) gates together in the physical layout.

C. State Access for Throughput Analysis

Our throughput analysis comes from the Weaver, an asynchronous crossbar switch in 40 nm CMOS, with measured ring FIFO throughput up to 6 Giga data items per second [17]. Fig. 6 outlines the approach and shows simulated throughput up to 10 Giga data items per second for the RSFQ ring FIFO. We *avoid* test access to the following stateholders:

- D2 latches in *L2*, *L3*, and *L4*, because their data come around to Link *L1* for test control and observation;
- Joint *TURN*s, because a *TURN* state is the same before and after a COPY operation and can be controlled and observed via scan access to Link *TURNA*s and *TURNB*s;
- Joint *MrGO.SYNC*s, because our approach to throughput analysis works with rather than against the protocols.

Crucially, Link *TURNA* and *TURNB* are STATE gates with nondestructive readout. Their states *outlive* read operations. In contrast, states *vanish* to move along with the gate output for

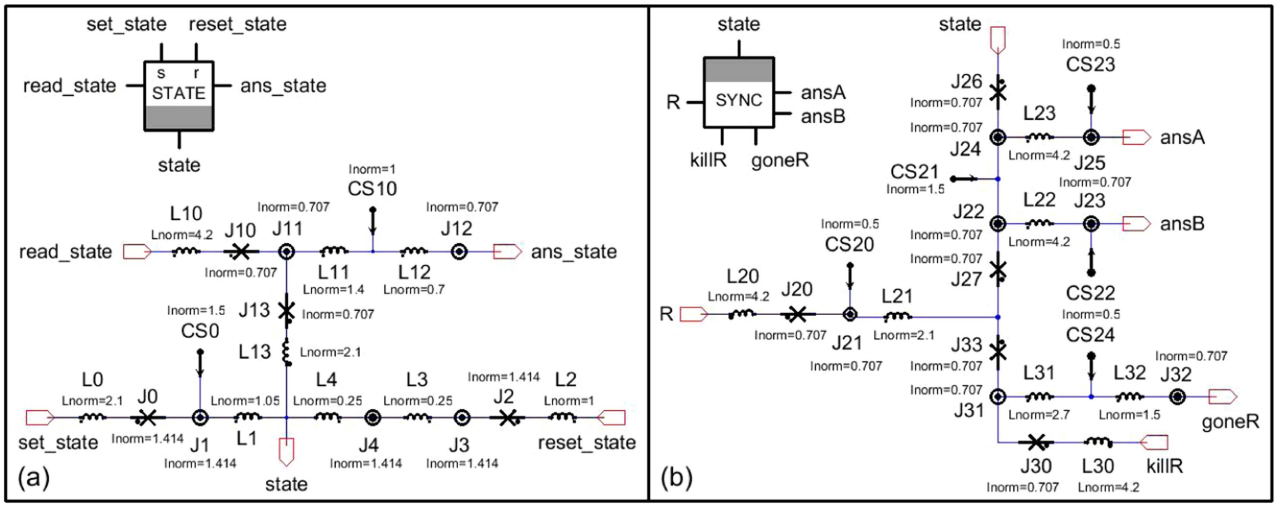

Fig. 4. For NDRO gate STATE (a) we represent the presence or absence of circulating current through quantizing inductor *L1* as state s_1 being *true* or *false* (see Section IV-A). As in other NDRO gates, s_1 can be set *true*, reset *false*, and read out nondestructively from *read_state* to *ans_state*. In addition, s_1 is shared externally via signal *state* — both logically, as Boolean, and electrically, by the presence or absence of a trickle current. Similarly, s_2 relates to quantizing inductor *L21* in gate SYNC (b), is set via *R*, and read out destructively from *killR* to *goneR*. Upon $(state \wedge s_2)$, SYNC generates a pulse on *ansA* and *ansB*, and resets s_2. In terms of RSFQ gates, SYNC combines a rendezvous (between *R*, *state*, *ansA/B*) with a latch (between *R*, *killR*, *goneR*). A reduced version of SYNC, without *killR–goneR*, uses Lnorm=2.8 for *L21*. Margin simulations show correct operation when we vary individual current sources *CS0*, *CS10* in STATE within [−30%,+25%], [−20%,+0%], and *CS20–CS24* in SYNC within [−40%,+30%], [−20%,+10%], [−100%,+45%], [−100%,+45%], [−50%,+70%], respectively.

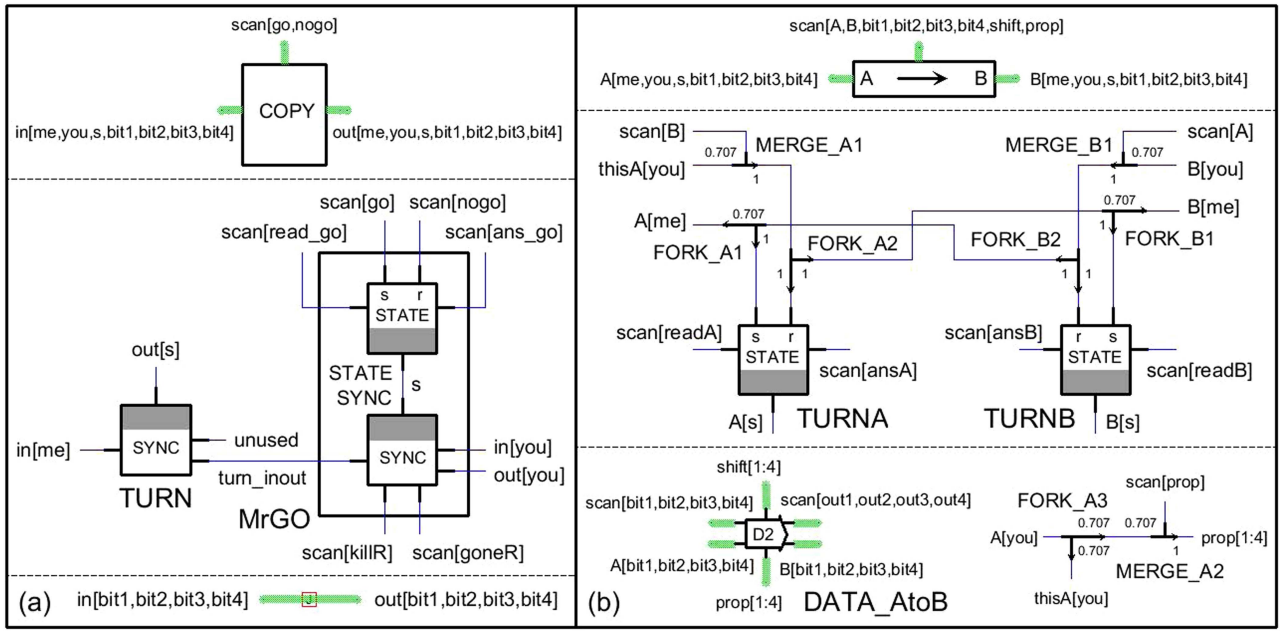

Fig. 5. RSFQ circuit and icon for the COPY Joint (a) and communication Link (b) used in the ring FIFO in Fig. 3.

(a) The Joint acts conditionally, waiting for (1) a pulse on *in[me]*, indicating that the data received over *in[bit1,bit2,bit3,bit4]* are valid and complete and that port *in* has the turn for its connecting Link, (2) a trickle current on *out[s]*, indicating that port *out* has the turn for its connecting Link, and (3) a trickle current on *MrGO.s*, giving the Joint permission to act. The waiting sequence is irrelevant, because the SYNC gates in *TURN* and *MrGO* rendezvous their inputs — see Fig. 4 — and because (in normal operation) the Link-Joint protocol guarantees persistent trickle currents for as long as the Joint waits. When the wait is over, the Joint acts by generating a pulse on *in[you]* and *out[you]*, relinquishing both Link turns. By then, the Joint has also copied *in[bit1,bit2,bit3,bit4]* to *out[bit1,bit2,bit3,bit4]*.

(b) The Link has two mutually exclusive STATE gates, *TURNA* and *TURNB*, that together store whose turn it is to propagate Link data or change the turn. Before we set one STATE we reset the other. During normal operation, *B[you]* resets *TURNB* and gives the turn to Link port *A*, and thereby to the Joint port connected to *A*. For initialization, test, and analysis, we use *scan[A]* to do the same. Giving the turn to *A* not only sets *TURNA*, causing a trickle current on *A[s]*, but also generates a pulse on *A[me]*. The Joint uses one or the other to determine if *A* has the turn. Likewise, *A[you]* and *scan[B]* reset *TURNA*, set *TURNB*, and generate a pulse on *B[me]*, but also propagate data stored in D2 latches of *DATA_AtoB*. The picture omits the forks in *prop[1:4]* and between *scan[shift]* and *shift[1:4]*.

Note that the ring FIFO in Fig. 3 uses only a subset of the scan connections provided by its Links and Joints.

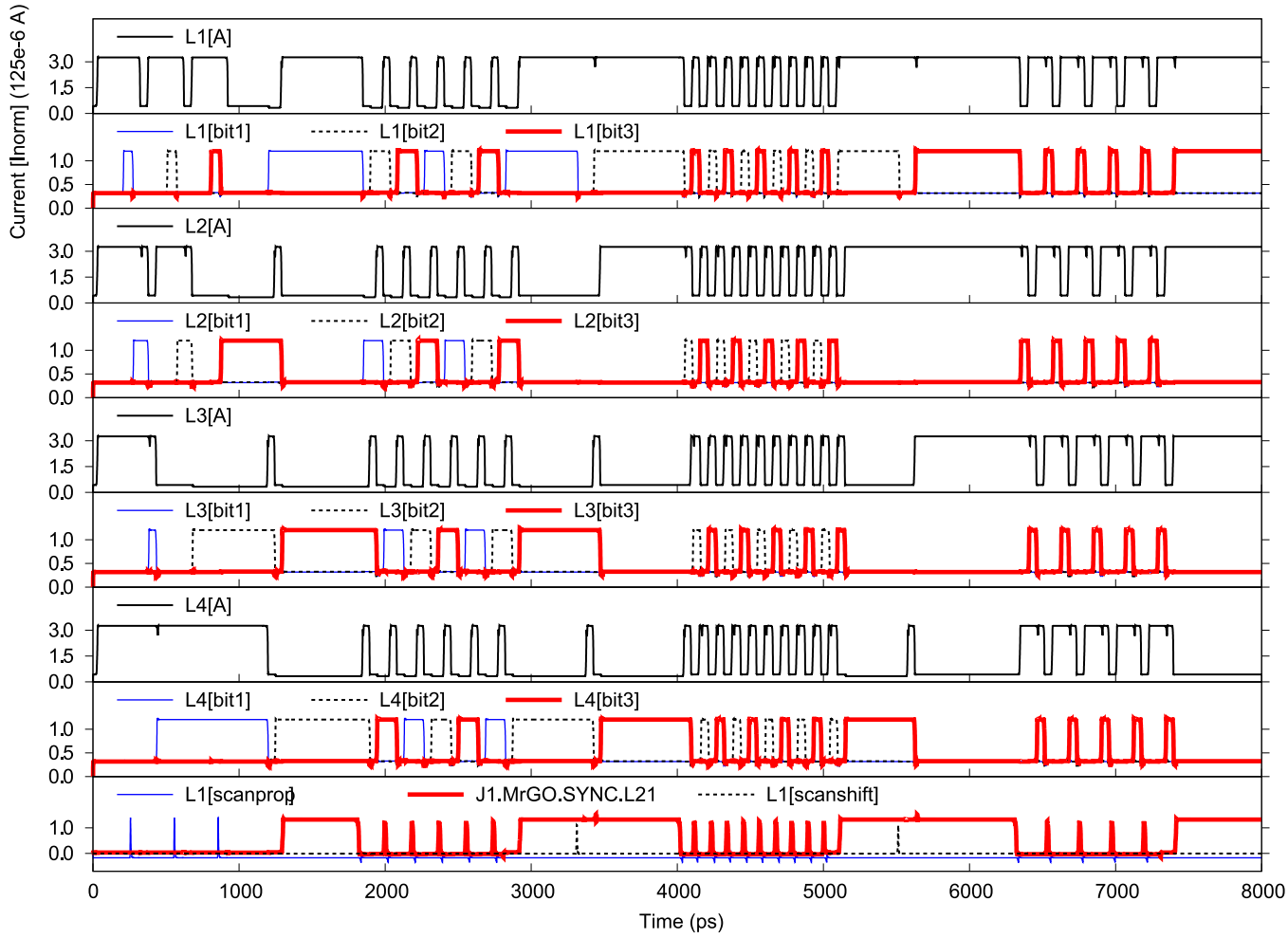

Fig. 6. Simulated waveforms showing *throughput* of the asynchronous ring FIFO in Fig. 3 versus *occupancy*, the number of data items in the ring. The waveforms for *Li[A]* and *Li[bitj]*, *i =1..4*, *j=1..3*, show the current through inductors *Li.TURNA.L1* and *Li.DATA_AtoB.L2[j]* — see Figs. 1(b), 4(a), 5(b). The waveform for *L1[scanprop]* shows the current through the *scan[prop]* branch inductor in *L1.MERGE_A2* — see Fig. 5(b). The waveform for *L1[scanshift]* shows the current through an inductor in the fork tree connecting *scan_L1[shift]* to *L1.shift[1:4]* — see Figs. 3, 5(b). The current for *J1.MrGO.SYNC.L21* speaks for itself.

- The most important simulation parts are the three 1000 ps run periods, starting around 1800, 4000, and 6300 ps, during which the ring FIFO operates at full-native speed. These run periods are confined between "blocks" of stable high current levels for *J1.MrGO.SYNC.L21* (bottom row) during which COPY Joint *J1* is disabled. For the first run period, 1800–2800 ps, we count 6 pulse intervals for *J1.MrGO.SYNC.L21*, which indicates that Joint *J1* acted six times. We also count 6 data pulses on Link *L2*, all from *L2[bit1]*, *L2[bit2]*, and *L2[bit3]* (fourth row from top). This indicates that *L2* transferred data six times, in lockstep with *J1.MrGO.SYNC.L21*, and that the ring FIFO contains three data items. For the second run period, around 4000–5000 ps, we count 10 data pulses on *L2*, all from *L2[bit2]* and *L2[bit3]*. For the third run period, 6300–7300 ps, we count 5 data pulses on *L2*, all from *L2[bit3]*. From this information, we can determine the throughput of the ring FIFO as a function of "the number of data items in the ring" also known as *occupancy*. The waveforms show that for occupancies 3, 2, and 1, the ring FIFO runs at 6, 10, and 5 Giga data items per second.
- In future on-chip measurements, the task of counting how often a data item passes a given location in the ring FIFO can be fulfilled by a scan-testable ripple counter [13], [17], triggered by the currently unused *TURN* output in one of the COPY Joints — see Fig. 5(a).
- From 0 to 1800 ps, we scan three data items into the ring. The ring FIFO has only partial (*L1*) scan access to data, which forces us to merge scan and normal operations to get more than one data item in the ring. After power-up, all Joints are disabled and cannot interfere with scan operations. Around 5 ps, we generate a pulse on *scan_Li[A]*, *i =1..4*, giving the turn to each Link port *A* and thereby to the output port of each Joint. The waveforms for *Li[A]* now show a stable high current. Around 105 ps, we generate a pulse on *scan_Ji[go]*, *i=2..3*, to enable Joints *J2* and *J3*, and prepare a runway for the data to move through the ring as far as *L4*. The Joints remain idle, because only their output ports have the turn. We use three scan steps to "walk in" each data item. First, we scan in a data value by generating the right pulses on *scan_L1[bit1,bit2,bit3,bit4]*. Second, we propagate this value to Joint *J2* by generating a pulse on *scan_L1[prop]*. Third, we start *J2* and mobilize the runway by generating a pulse on *scan_L1[B]*. We "walk in" data at 200, 500, and 800 ps, using 1-hot *scan_L1[bit1,bit2,bit3,bit4]* values 1000, 0100, and 0010, represented here as *Li[bit1]*, *Li[bit2]*, and *Li[bit3]*, *i=1..4*. Note that around 1000 ps, the first value is stored in Link *L4*, the second in *L3*, the third in *L2*. Around 1200 ps, we enable *J4*, which causes all data to move to the next Link and puts *J1* and *L1* in charge of start, stop, and data removal for each 1000 ps run.
- We start and stop each 1000 ps run by generating a pulse on *scan_J1[go]* and 1000 ps later on *scan_J1[nogo]*. After the run has petered out, the data are queued up counterclockwise from *L1*, and the turn for all Links with data is with *B*. To remove a data item we generate first a pulse on *scan_L1[scanshift]*, clearing the data in *L1*, and then on *scan_L4[A]*, changing the turn in *L4* to *A* to propagate data from *L4* to *L1* and forward any data in *L3* and *L2* by one Link. We remove a data item around 3300–3500 ps and 5500–5700 ps.

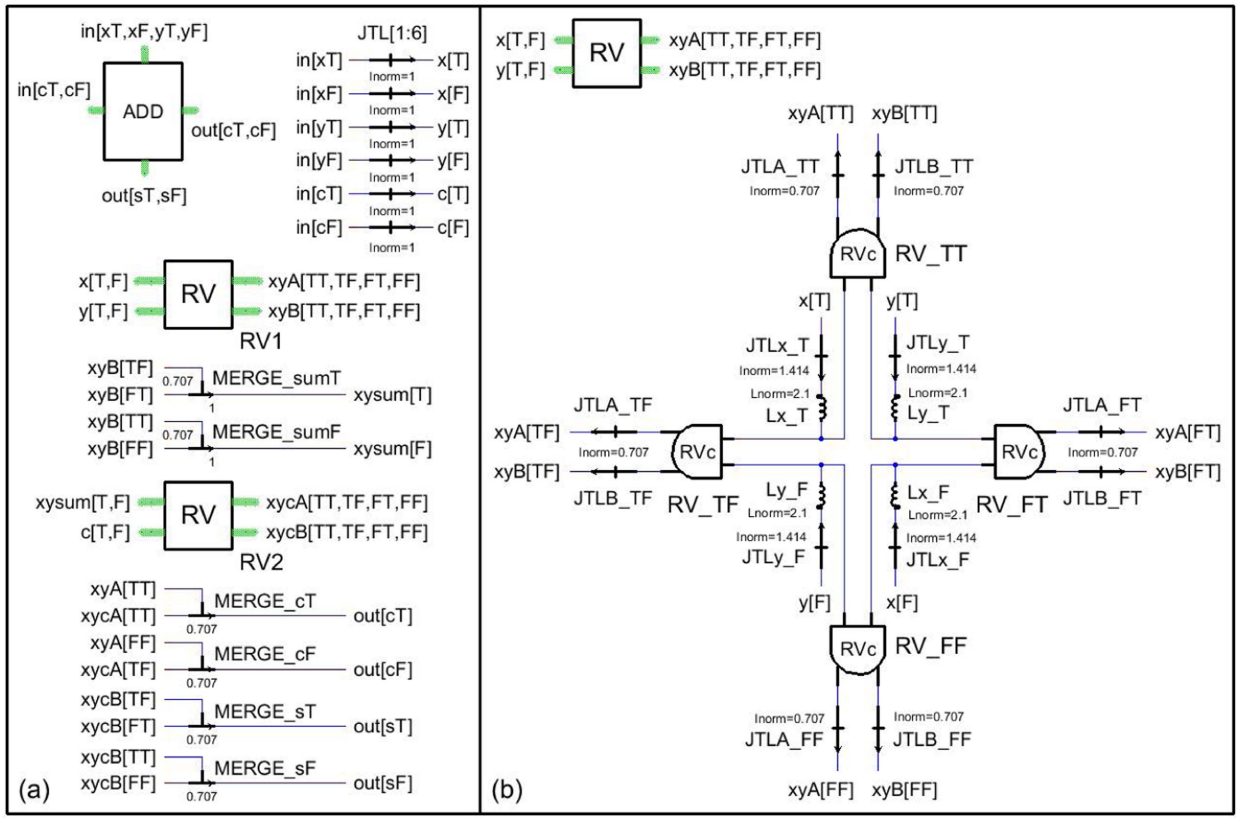

Fig. 7. RSFQ circuit and icon for bit-wise adder Joint ADD of Fig. 1(a) and its rendezvous gate RV (b). ADD takes 1-bit inputs *in[xT,xF]*, *in[yT,yF]*, *in[cT,cF]*, and generates 1-bit sum and carry outputs, *out[sT,sF]*, *out[cT,cF]*. Inputs and outputs are dual-rail encoded, with bit value 1 represented by a pulse on *T* and bit value 0 by a pulse on *F*. As elsewhere in this paper, we design conservatively and amplify signals as needed using stepup factors of approximately $\sqrt{2}$ per Josephson junction. For instance, we amplify all ADD inputs using size 1 (125e-6 A) Josephson transmission lines *JTL[1:6]* before passing them on to 1.414 transmission lines in *RV1* and *RV2*. A single stepup suffices, because ADD inputs come from 0.707 D2 latches. Core rendezvous gate RVc in (b) matches subcircuit *J26-J24-J22-J27* in Fig. 4(b) with output connections at *J24*, *J22* and current source *CS21* — except that in RVc, *J26*, *J27*, *CS21* accommodate size 1 currents.

latch gates like the D2 latch in Fig. 1(b) and for rendezvous gates like SYNC in Fig. 4(b) and RVc in Fig. 7(b). Where latch and rendezvous create moving test targets, STATE provides a fixed location to (1) start, stop, and stall Joint operations, and (2) examine, initialize and propagate Link information.

V. Conclusion

We have shown a test strategy for RSFQ circuits in enough detail to enable replication and practicality. We gave two examples, reflective of state-of-the-art superconducting designs [4], [6], [9], [11]—a clocked pipelined adder for which we tested functionality, and an asynchronous ring FIFO for which we analyzed throughput—both designed in Electric [20], and simulated with equivalent JoSIM [3] and VWSIM [10] results.

As in CMOS [19], our RSFQ examples use *Links* to store and transfer information between *Joints* for computation and flow control. In our asynchronous example, each Joint has a *go-nogo* gate, *MrGO*, to enable or disable Joint action. Our RSFQ test strategy combines scan-based test access to Link information with a *clock* or with scan-based *MrGO* control.

The examples show scan access points but avoid tying them to scan chains, because (1) the chain sequence is irrelevant to our test strategy and any functional or structural test generation we might use [23], and (2) shifting information in and out of scan chains would dominate the simulations and obscure the results. Future on-chip measurements require scan chains, a ripple counter, and a scan test interface [13], [15], [17], [21].

Appendix

Fig. 7 shows the RSFQ circuit for Joint ADD in Fig. 1. The circuit is based on the clockless dual-rail bit-wise full adder by Patra et al. [16, Fig. 13]. In our version, each rendezvous gate has two outputs, avoiding the explicit forks in the adder version of Patra et al.

For sound and complete dual-rail inputs, with a pulse on the *T* or *F* bit signal but never both, the circuit will generate sound and complete dual-rail outputs and leave no pulse behind. Likewise, *nodata* dual-rail inputs, with a pulse on neither the *T* nor the *F* bit signal, yield *nodata* dual-rail outputs and leave no pulse behind. Our simulation of the clocked pipelined adder, with results shown in Fig. 2, uses both these ADD properties.

Other input combinations are illegal and may leave pulses behind that can be removed with an extra reset signal.

Acknowledgment

The authors thank Gary Delp, Bart McCoy, Quinn Morgan, and Glenn Shirley for feedback and comradery. They thank Steve Rubin for making the open-source Electric CAD system conversant with RSFQ.

REFERENCES

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Hoboken, NJ, USA: Wiley-IEEE Press, 1990.

[2] P. Bunyk, K. Likharev, and D. Zinoviev, "RSFQ technology: Physics and devices," *Int. J. High Speed Electron. Syst.*, vol. 11, no. 1, pp. 257–305, 2001.

[3] J. A. Delport, K. Jackman, P. le Roux, and C. J. Fourie, "JoSIM – Superconductor SPICE simulator," *IEEE Trans. Appl. Supercond.*, vol. 29, no. 5, pp. 1–5, Aug. 2019.

[4] Z. J. Deng, N. Yoshikawa, S. R. Whiteley, and T. Van Duzer, "Self-timing and vector processing in RSFQ digital circuit technology," *IEEE Trans. Appl. Supercond.*, vol. 9, no. 1, pp. 7–17, Mar. 1999.

[5] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[6] M. Dorojevets, C. L. Ayala, N. Yoshikawa, and A. Fujimaki, "16-Bit wave-pipelined sparse-tree RSFQ adder," *IEEE Trans. Appl. Supercond.*, vol. 23, no. 3, Jun. 2013, Art. no. 1700605.

[7] E. Esimai and M. Roncken, "Flexible active-passive and push-pull protocols," *IEEE Embedded Syst. Lett.*, vol. 14, no. 3, pp. 139–142, Sep. 2022.

[8] Q. P. Herr and P. Bunyk, "Implementation and application of first-in first-out buffers," *IEEE Trans. Appl. Supercond.*, vol. 13, no. 2, pp. 563–566, Jun. 2003.

[9] D. S. Holmes, A. M. Kadin, and M. W. Johnson, "Superconducting computing in large-scale hybrid systems," *Computer*, vol. 48, no. 12, pp. 34–42, 2015.

[10] W. A. Hunt Jr., V. Ramanathan, and J. Strother Moore, "VWSIM: A circuit simulator," in *Proc. Int. Workshop ACL2 Theorem Prover Appl.*, 2022, pp. 61–75.

[11] S. Kundu, G. Datta, P. A. Beerel, and M. Pedram, "qBSA: Logic design of a 32-bit block-skewed RSFQ arithmetic logic unit," in *Proc. IEEE Int. Supercond. Electron. Conf.*, 2019, pp. 1–3.

[12] K. K. Likharev and V. K. Semenov, "RSFQ logic/memory family: A new Josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Trans. Appl. Supercond.*, vol. 1, no. 1, pp. 3–28, Mar. 1991.

[13] O. A. Mukhanov and S. V. Rylov, "Time-to-digital converters based on RSFQ digital counters," *IEEE Trans. Appl. Supercond.*, vol. 7, no. 2, pp. 2669–2672, Jun. 1997.

[14] O. A. Mukhanov, "Superconductive single-flux quantum technology," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 1994, pp. 126–127.

[15] O. A. Mukhanov, "Rapid single flux quantum (RSFQ) shift register family," *IEEE Trans. Appl. Supercond.*, vol. 3, no. 1, pp. 2578–2581, Mar. 1993.

[16] P. Patra, S. Polonsky, and D. S. Fussel, "Delay insensitive logic for RSFQ superconductor technology," in *Proc. IEEE Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, 1997, pp. 42–53.

[17] M. Roncken and I. Sutherland, "Design and test of high-speed asynchronous circuits," in *Asynchronous Circuit Appl.*. J. Di and S. C. Smith, Eds. London, U.K.: The Inst. Eng. Technol., ch. 7, 2020, pp. 113–171.

[18] M. Roncken et al., "How to think about self-timed systems," in *Proc. Asilomar Conf. Signals, Syst., Comput.*, 2017, pp. 1597–1604.

[19] M. Roncken, S. Mettala Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, "Naturalized communication and testing," in *Proc. IEEE Int. Symp. Asynchronous Circuits Syst.*, 2015, pp. 77–84.

[20] S. M. Rubin, *Using the Electric VLSI Des. System*. Portola Valley, CA, USA: R. L. Ranch Press, 2016, [Online] https://www.staticfreesoft.com

[21] I. Sutherland, Q. Morgan, W. A. Hunt Jr., V. Ramanathan, and M. Roncken, "An IEEE-compatible JTAG test-access-port controller for RSFQ logic and systems," *IEEE Trans. Appl. Supercond.*, early access, Mar. 6, 2023, doi: 10.1109/TASC.2023.3251942.

[22] M. Vratislav, E. Baggetta, M. Aurino, S. Bouat, and J.-C. Villegier, "Superconducting RSFQ logic: Towards 100 GHz digital electronics," in *Proc. IEEE Int. Conf. Radioelektronika*, 2011, pp. 1–8.

[23] F. Wang and S. K. Gupta, "An effective and efficient automatic test pattern generation (ATPG) paradigm for certifying performance of RSFQ circuits," *IEEE Trans. Appl. Supercond.*, vol. 30, no. 5, pp. 1–11, Aug. 2020.