Received November 14, 2018, accepted February 7, 2019, date of publication February 15, 2019, date of current version March 4, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2899873

Automated Verification and Optimization of SFQ Superconducting Circuits

ANDREW M. HASLAM, (Student Member, IEEE), KURT M. ENGLISH, (Student Member, IEEE), ALEXANDER DERRICKSON, (Student Member, IEEE), AND JOHN F. MCDONALD, (Life Senior Member, IEEE)

Center for Integrated Electronics, Rensselaer Polytechnic Institute, Troy, NY 12180, USA

Corresponding author: Andrew M. Haslam (haslaa@rpi.edu)

This work was supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via IARPA Contract #W911NF-14-C-0090.

ABSTRACT New tools have been created to allow a superconducting design flow for schematic design, verification, and optimization. These tools integrate with the Cadence design environment. In single flux quantum superconducting electronics, individual component values, such as wire inductances, Josephson junction critical currents, and bias currents, must be optimized to allow for maximum deviance from the designer value, which is also known as the device margin. One tool is used to create a description of the proper circuit behavior. Included with this tool is the ability to automatically create the description from a Cadence netlist. The other tool is an automated device margin circuit schematic verification and optimization tool, which widens device margins while maintaining proper circuit behavior derived from the first tool. Additionally, this optimization tool can automatically correct the circuit schematic using the proper circuit behavior description. In this paper, the functionality of the language used to create the description of the proper circuit behavior is presented. Several circuits are then verified and optimized based on their correct behavior.

INDEX TERMS Circuit verification, circuit optimization, Josephson junctions, RSFQ circuits, single flux quantum (SFQ), superconducting electronics.

I. INTRODUCTION

Historically, the state of CAD tools available to the superconducting electronics (SCE) design community has been both outdated and not suitable for complex designs [1], [2].

These CAD tools were only suitable for the design of chips of a modest complexity due to ''their lack of discipline and volume'' [3]. Major improvements to the currently available CAD tools are needed in order to increase the density, complexity, and performance of SCE chips [3]. While some work has been accomplished in this area [4], this is an improvement upon the old CAD tools and not a novel approach to the SCE optimization problem. With a new version of the hierarchical Single-Flux-Quantum Hardware Description Language (hSFQHDL), known as the hierarchical Single-Flux-Quantum Hardware Description Language in Cadence (hSFQHDLC), and a new single flux quantum circuit analysis and optimization tool, a novel approach to the problem of circuit optimization is presented. This tool targets the Rapid-Single-Flux-Quantum (RSFQ) [5] family of technology, where the presence or absence of a single quantum of magnetic flux is used to represent the ones and zeroes for logic operation, can operate at several tens of GHz speed with very low power consumption [6] and has been shown to be a contender for future energy-efficient large-scale computing systems [7]. This can be attributed to fundamental advantages SFQ circuits possess in low energy and high speed switching [8]. Success has been demonstrated in the field [9]–[11], including the development of RSFQ microprocessors such as SCRAM2 [12] and the CORE1 [13] series.

New energy-efficient versions of RSFQ, such as ERSFQ, eSFQ [14], [15] and LV-SFQ [16], have further reduced power consumption for systems built on this technology.

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

These new additions to the SFQ family show great promise, and success has already been demonstrated with these energy-efficient technologies as well [17]–[20].

However, the design of large Single-Flux-Quantum (SFQ) based circuits becomes challenging very quickly as circuit complexity scales compared to traditional CMOS based circuits. The switching action of each of the Josephson junctions (JJ), which is seen as a 2π change in quantum phase, within the circuit needs to be analyzed across the time domain of the simulation in order to verify proper operation of the circuit. SFQ circuits are also extremely sensitive to deviations from the ideal design values and the circuit must have the margins of each component maximized to increase the chances of a successful fabrication run. By verifying the switching action of all JJs within a circuit, the designer can than optimize the critical currents of the junctions, along with the values of other circuit components, and thus maximize the chances of a successful fabrication run.

Traditionally, the Single Flux Quantum Hardware Description Language (SFQHDL), parameter optimization, and circuit verification have been handled by the Personal Superconductor Circuit ANalyzer (PSCAN) [21] or, more recently, PSCAN2 [4]. The new hSFQHDLC integrates with the Cadence design suite, allowing for features such as automatic compilation from a Cadence netlist, structural netlist to hSFQHDLC conversion, and integration with a new optimization tool. Using a Cadence environment will offer the enhanced maintainability offered by a commercial EDA tool base, as well as allowing the designer to perform automated place and route, verification, and optimization all within the same tool suite. Any Cadence enhancements or improvements leading to increased program efficiency or accuracy would then be immediately available to the designer.

One piece of a modern integrated circuit (IC) design workflow is the schematic verification and optimization. A Cadence based SFQ circuit design workflow that addresses this functionality using hierarchical Single-Flux-Quantum Hardware Description Language in Cadence consists of the following steps (see Fig. 1). First, a netlist of the circuit under test is passed to Cadence Spectre, which is the tool used to perform the transient time-domain simulations. The same netlist is passed to the hSFQHDLC Automated Generation Tool, which uses the information contained in the netlist to automatically generate a hSFQHDLC description of the circuit under test. This hSFQHDLC description is then passed to the hSFQHDLC compiler, which generates a description of the correct circuit behavior. As Spectre simulates, the switching action of each junction is compared to the description of the correct circuit behavior in order to determine if the circuit operates correctly. This is done automatically using the new single flux quantum circuit analysis and optimization tool that is presented in this paper.

If the circuit passes this analysis, the margins of the circuit are calculated and then the parameters of the circuit can be modified in order to optimize the circuit margins. This can either be done manually by the designer or automatically

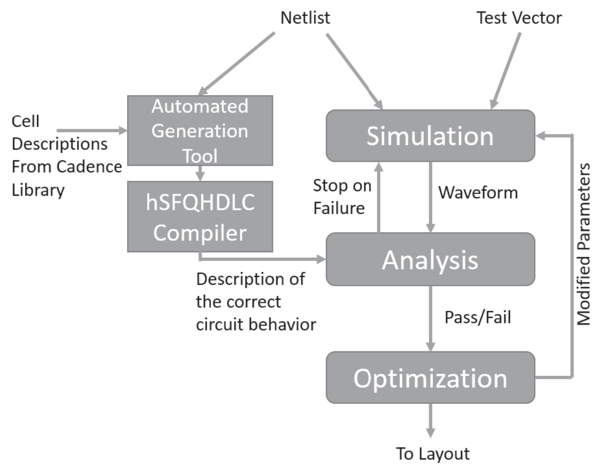

FIGURE 1. Design flow for superconducting circuit verification.

by using a tool such as the new single flux quantum circuit analysis and optimization tool, COWBoy [22], xopt [23], or MALT [24]. This new circuit is then analyzed again to verify that it still matches the previously established description of the correct circuit behavior. This process is repeated until the optimal circuit is achieved.

If the circuit fails the analysis, the circuit must be corrected until it meets the provided circuit description. While previously the designer was expected to do this manually, with the new single flux quantum circuit analysis and optimization tool, SFQ circuits which do not meet the initial circuit behavior description will have their component values intelligently changed by the program to create a working circuit.

As circuit complexity scales a cell-based [25] design approach can be used. Individual cells are optimized to have as little static interaction between each other in connection as possible. However there is still some interaction between cells, and so it is beneficial to the designer to optimize as large of a circuit as possible. By introducing a new, parallelized optimization tool that leverages Cadence Spectre for simulation larger circuits may be optimized than were previously possible.

This design flow seen in Figure 1 allows for fast and reliable verification of SFQ circuits in a Cadence design environment. In this paper hSFQHDLC and its functionality will be presented in Section III. The circuit analysis and optimization tool will then be used to analyze and optimize a circuit in Section IV. The ability of this tool to correct a circuit that does not meet its initial description will also be demonstrated in this section. Comparison will be given to an existing optimization tool in Section V and it will be shown how hSFQHLDC translates from a Structural Netlist in Section VI.

II. COMPONENT VALUES AND THE EFFECT ON CIRCUIT FUNCTIONALITY

The value of a component in a SFQ circuit may deviate from what the designer intended when creating a circuit layout.

This can be due to fabrication tolerances or the designer not drawing the layout to perfectly match the schematic. In order to prepare the circuit for layout and fabrication, each component value must be optimized to have the widest possible margin. This will allow each component to deviate from the schematic value and still produce a working circuit.

In order to move from one point to another, an SFQ pulse must propagate down a wire. In superconducting technology, a wire is an inductor and must be treated as such. Raising or lowering an inductance by changing the width or length of a wire can affect the operation of the circuit, even causing a complete failure of the circuit as a whole.

Figure 2 shows the optimized circuit schematic for a splitter [5]. This schematic uses the symbol "X" for a Josephson junction and those junctions are biased with current sources. Current sources are used as this is an RSFQ logic scheme. In layout, resistors would be used in place of the current sources to bias the circuit.

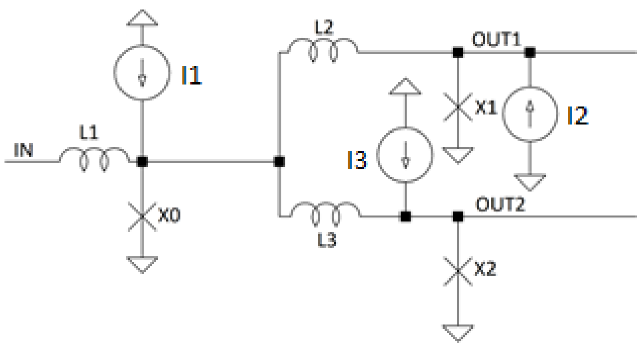

FIGURE 2. Splitter; Values: X0 = 300uA, X1 = 250uA, X2 = 250uA, L1 = 3p, L2 = 3p, L3 = 3p, I1 = 250uA, I2 = 187.5uA, I3 = 187.5uA.

In Figure 3, an SFQ pulse labeled "IN" is input into a splitter which creates two identical SFQ pulses at two different output terminals. The schematic for this design can be seen in Figure 2. By changing the inductance of the wire labeled "L2" the operation of the splitter can be modified to the point of failure.

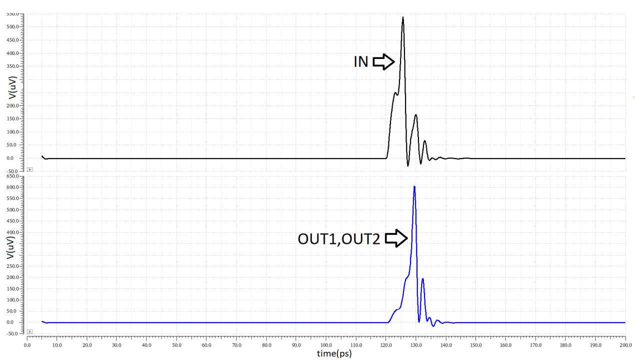

FIGURE 3. Simulation of splitter correct operation.

In Figure 3, the value of "L2" is 3pH. By simply increasing this value to 8p H, failure is seen in the circuit when the second output "OUT1" no longer outputs an SFQ pulse (Figure 4). Likewise, by decreasing "L2" to 0 pH failure is seen when "OUT2" no longer produces an SFQ pulse (Figure 5). This failure due to decreasing "L2" is only possible due to the fact that the junctions in this circuit are underbiased. The robustness of this inductor is affected by the values of the other components in the circuit. Thus, by optimizing the circuit the upper and lower bounds of the inductor "L2" will increase allowing for greater variance from the circuit values during manufacturing and layout.

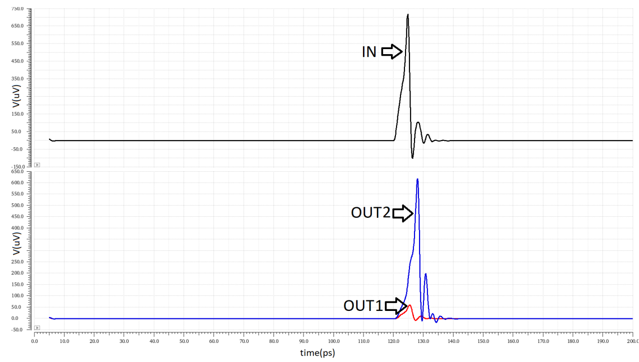

FIGURE 4. Simulation of splitter, increased inductance.

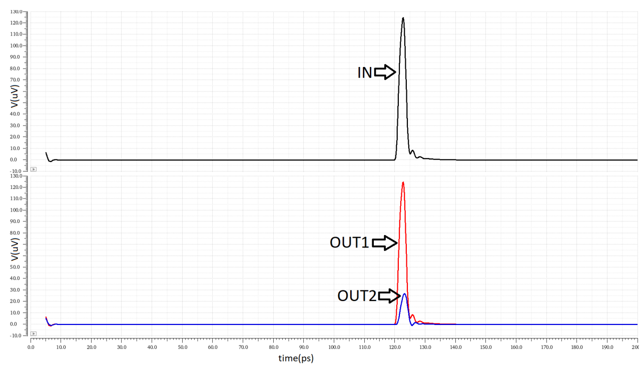

FIGURE 5. Simulation of splitter, decreased inductance.

III. HIERARCHICAL SFQHDLC

For SFQ circuits, it is not always easy to perform manual circuit verification via waveform or quantum phase analysis for complex circuits. As such, a tool is needed to automate circuit analysis. This tool will need to be provided with a description of the proper behavior of the circuit in order to verify the circuit functionality. SFQHDL was first introduced in 1991 [21] and later updated in 1996 [22] to include provisions for hierarchical constructs, i.e. circuits containing subcircuits. This language allows the designer to describe the behavior of a circuit in terms of the switching of the junctions within that circuit. This circuit is then passed to a simulator, such as Cadence Spectre [26], and the output of the circuit is compared to the description by a separate optimization and analysis tool. While PSCAN [22], JSIM [27], WRspice [28], or another simulator [29] could also be used, the simulator would have to be adapted to use hSFQHDLC.

Hierarchical SFQHDLC has many benefits over input/output checking, automated or manual. By describing the circuit in terms of the switching of each junction, the designer now has access to a high level of detailed information about their SFQ circuit. Checking junction switching events instead of inputs and outputs allows for detection of more circuit errors. For example, while a malfunctioning circuit may have produced a correct output, it may not have reset correctly to the base state, leading to corrupt output at a much later time. It also allows the designer to know exactly where in the circuit the problem occurred, speeding up the debugging process. This information can also be passed to a computer program, which can then be used to fix the circuit instead of the designer.

A. THE N-FUNCTION

A Josephson junction consists of two superconductors coupled by a weak link. The superconducting bound-electron pairs, known as Cooper pairs [30], tunnel though this insulator and the supercurrent is uninterrupted. These Cooper pairs will continue to tunnel though the insulator until the critical current of the junction, determined by the critical current density and size of the junction [31], is reached. At this point, the junction will oscillate in time and experience an AC voltage. Each pulse of voltage generated by the junction at this point also corresponds to the quantum phase across the junction rotating by 2π. This is known as a junction "switch" or "flip".

As hSFQHDLC relies on the switching action of each of the junctions to describe the circuit, it is important that this behavior is reported accurately and without false positives. This switching action is reported using an integer valued hysteretic function as seen in [22], known as the N-function. The N-function relates the number of flux quanta that have penetrated the junction to the quantum phase across the junction and reports this as an integer valued number. Effectively, this function reports the number of times the junction has switched.

The N-function allows for the creation of two more functions unique to hSFQHDL and hSFQHDLC, the INC and DEC [22] functions. These two functions return 1 at the moment that the N-Function increments or decrements, respectively. A small positive or negative change in the quantum phase will not trigger INC or DEC, only a $3/2\pi$ or greater phase change. At all other times the output of these functions is zero.

The functionality of the old hSFQHDL language has been recreated and improved upon to accommodate large-scale circuit verification. Hierarchical SFQHDLC has new syntax that is comparable to the Verilog Hardware Description Language.

B. RULE STATEMENTS

The most basic construct of the language is the rule statement, which is used to describe the correct behavior of a set of junctions which make up a circuit. A rule combines operators and operands to produce a result that is a statement that the circuit must obey. The operators and operands available in hSFQHDLC can be seen in Table 1.

TABLE 1. hSFQHDLC operators.

Symbol	Function
AND &&	Logical AND
OR ||	Logical OR
!	Logical Negation
==	Logical Equality
!=	Logical Inequality
+ - * /	Arithmetic
> >= < <=	Relational
INC	Increment Junction Phase
DEC	Decrement Junction Phase
N	N-Function

An hSFQHDLC description of any circuit is, at its most basic, a list of rules, otherwise known as a "rule deck". Together, this rule deck makes up a description of the proper behavior of a SFQ circuit.

C. RULE CONSTRUCTION

By combining the operators and operands in Table 1, the designer can create a set of rules for any SFQ circuit. A rule statement is written as follows:

RULE name (cause) effect : time;

The three main parts of this statement are the cause, the effect, and the time operator. The cause is the event which activates a rule statement, while the effect is the event which must occur to maintain normal operation of the circuit. The time operator specifies the amount of time that the effect has to occur after the cause takes place in order for the rule to be considered valid. The timing operator can be left blank if the user wishes to specify no timing failure.

To clarify, consider as an example the 2-stage Josephson Transmission Line (JTL) [5] in Figure 6. A RULE statement to describe the proper operation of this circuit could be written as:

RULE GO (INC(A)) INC(B) : 5p;

In English, this statement would be read as "If the quantum phase across the junction A increments, then the quantum phase across the junction B must increment within 5 picoseconds". GO is simply the name of the rule designated by the designer and ignored by the compiler. This rule is then passed to the analysis tool, which compares it to the quantum phases across the junctions to verify the circuit functionality.

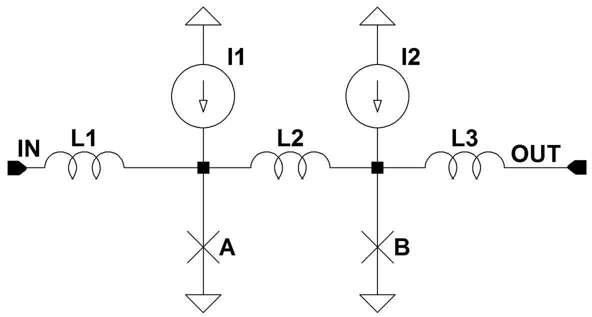

FIGURE 6. **2-Stage Josephson transmission line.**

D. INTERNAL STATES

An important feature in hSFQHDLC is the ability to keep track of the internal states of different cells. This is done through the use of the N-function described in Section III.A.

Consider as an example a DFF [5], seen in Figure 7. This cell has two states, and the switching order of the junctions changes depending upon the state of the cell. Hierarchical SFQHDLC handles these states through the use of relational logic and N-functions.

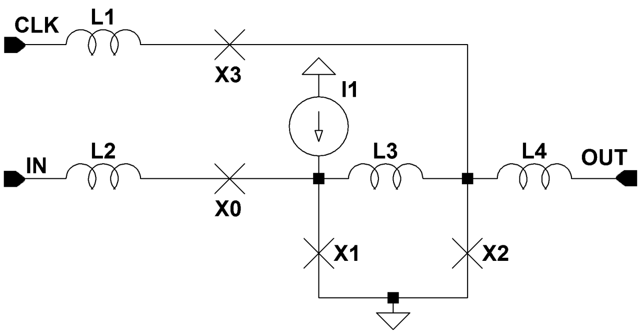

FIGURE 7. **D flip-flop.**

In the DFF, the state of the flip flop is stored in the relationship between the states of junctions X1 and X2. When the N-functions of the two junctions are equal, then the flip flop is at state ''0''. When the N-function of junction X1 is greater than that of junction X2, the flip flop is in state ''1'', or the storage state.

As an example, there are two actions that can take place upon the arrival of a clock pulse, depending upon the state of the flip flop. If the flip flop is in the ''0'' state, junction X3 will increment. Quantum phase is measured across the terminals of the junction model, thus the orientation of the model determines if the quantum phase across the junction increments or decrements. That rule would be written as follows:

RULE ((CLK) &&(N(X1) == N(X2))INC(X3);

This rule activates on the arrival of the clock input if and only if the N-function of junction X1 is equal to the N-function of junction X2. If the flip flop is in the ''1'' state instead, junction X2 will increment and an output will be produced. This is one way that this rule can be written:

RULE ((CLK) && (N(X1) == N(X2) + 1) INC(X2);

This rule activates on the arrival of the clock input if and only if the N-function of junction X1 is equal to the N-function of junction X2 plus one. A greater than sign could be used if the designer wished.

E. MULTIPLE EFFECTS

Not all causes have a single effect. In some cases, the switching order of the junctions may be neither clear nor relevant. In these cases, the switch of a single junction can cause multiple junctions to switch in any order. Take for example a binary decision diagram (BDD) [32] steering element constructed from a D2 flip-flop [33], as seen in Figure 8.

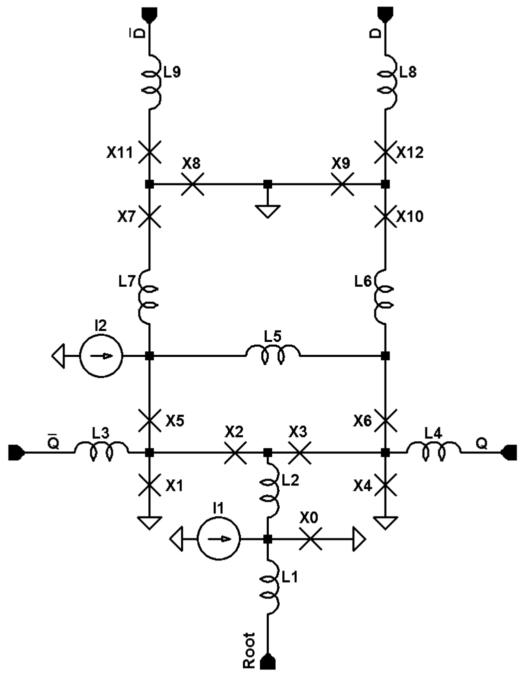

FIGURE 8. **BDD steering element.**

In this schematic, the arrival of a SFQ Pulse at the ''Root'' input will trigger the switching of the junction X0. In this case, the switching of X0 will then cause the switching of the junctions X1, X7, and X3 in any order. This would be written as:

RULE (INC(X0)) DEC(X3) && INC(X1) && DEC(X7);

If the designer cared about the order of the switching of junctions X1, X7, and X3 the multiple rule statements would be used instead. This would be written as:

RULE (INC(X0)) DEC(X3);
RULE (DEC(X3)) INC(X1);
RULE (INC(X1)) DEC(X7);

F. MULTIPLE QUANTUM PHASES

Another scenario which designers will encounter while designing SFQ circuits are junctions which generate multiple pulses. The simplest example of this is the SFQ to DC converter [5] seen in Figure 9.

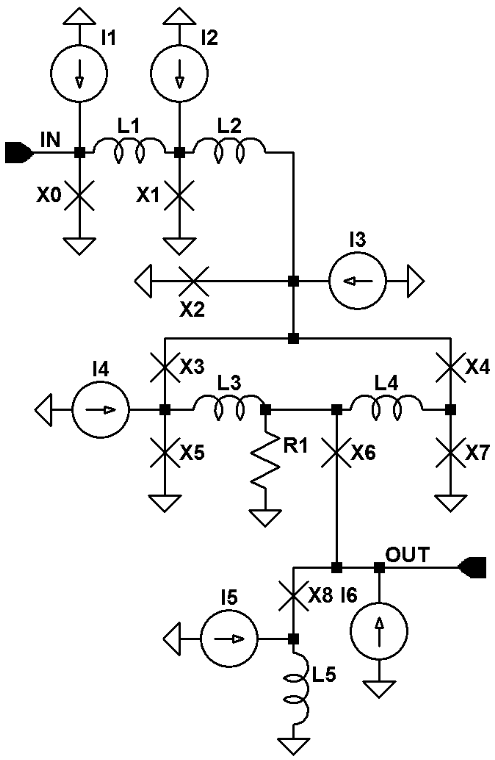

FIGURE 9. SFQ to DC converter.

In the SFQ to DC converter, a DC output is produced by setting the state of the internal T Flip-Flop [5] to "1" by increasing the N-function of junction X4 over that of junction X3. This initial setting action is accomplished through the arrival of a pulse at the input of the cell. While producing a DC output, junctions X6 and X8 will continuously switch until the state of the T Flip-Flop is reset to "0" by the arrival of a second input. This produces the DC output at the output terminal of the cell.

The rules for this are written by forming a loop, with the decrementation of junction X6 causing the incrementation of junction X8, which in turn would cause the decrementation of junction X6. The designer's initial attempt to represent this may look something like the following:

RULE (INC(X8) && (N(X4) = N(X3)+1)) DEC(X6);
RULE (DEC(X6) && (N(X4) = N(X3)+1)) INC(X8);

Which would be incorrect due to the fact that it would form an infinite loop. Any loop written in hSFQHDLC will need an exit case, which is accomplished through signer can create a set of eration seen in Table 1. As the "1" state (DC Output) of the internal T Flip-Flop is exited by incrementing the junction X3, this is used as the exit case. This results in the following rules:

RULE (INC(X8) &&!INC(X3) && (N(X4)
  == $N(X3) + 1$)) DEC(X6);
RULE (DEC(X6) &&!INC(X3) && (N(X4)
  == N(X3)+1)) INC(X8);

IV. USING hSFQHDLC FOR OPTIMIZATION

Section II showed that SFQ circuit functionality can be sensitive to component values and deviation from these schematic values can cause failure in circuit functionality. Hierarchical SFQHDLC can be combined with an optimization tool to maximize the margins of each component within a circuit.

A. D FLIP FLOP

The D Flip Flop seen in Figure 10 was created in Cadence Virtuoso using a JJ model created by the authors in Verilog-A. This model contains an extra output node to report the results of the N-Function discussed in Section III-A.

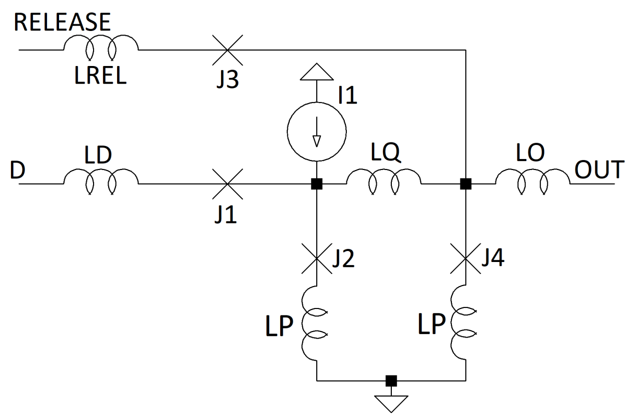

FIGURE 10. D flip flop with parasitic inductors (LP).

Cadence ADE was used to compile the Virtuoso schematic into a netlist and the netlist was simulated using Cadence Spectre. The initial simulation results reported by Cadence Spectre for the D flip flop are seen in Figure 11. The corresponding N-functions for this simulation are seen in Figure 12.

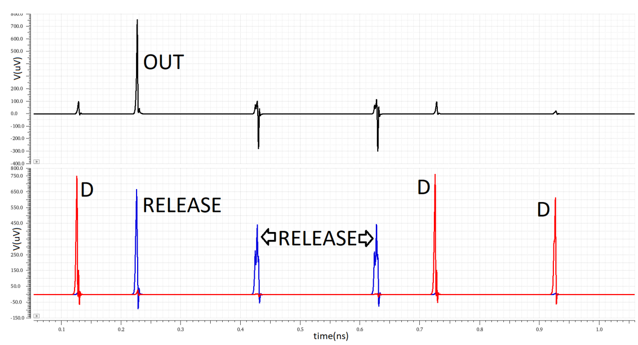

FIGURE 11. Simulation of a D flip flop in cadence spectre - pulses.

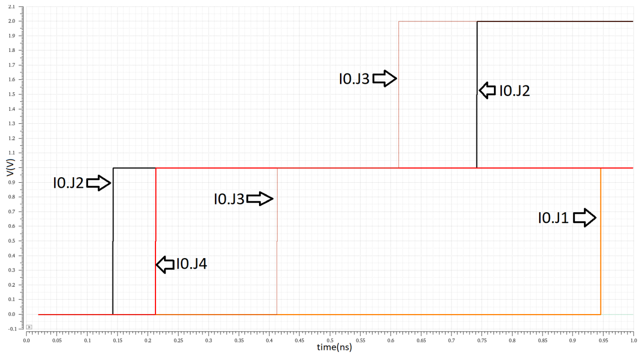

FIGURE 12. Simulation of D flip flop in cadence spectre - N functions of junctions.

The setting pulse to the flip flop can be seen in arriving first, labeled D, followed by the release pulse, labeled Release. The arrival of the release pulse causes the appearance of a pulse at the output of the flip flop, labeled OUT. The x-axis represents time in nanoseconds and the y-axis represents voltage in microvolts.

For optimization, the Cadence generated netlist for a schematic containing the D Flip Flop cell was passed to the Analysis and Optimization (AAO) tool, which added the appropriate DC to SFQ converters to the netlist. This netlist was then input to the hSFQHDLC automated generation tool. The automated generation tool links the user-generated hSFQHDLC descriptions from the designer's Cadence library to the netlist calls to produce the hSFQHDLC description. This description was then compiled into a list of rules by the hSFQHDLC Compiler.

This is done by a recursive flattening of the hierarchical Cadence netlist into a Cadence netlist where none of the cells contain other cells. The compiler then pulls the designer written SFQHDLC descriptions of each cell from the Cadence library and creates the circuit description, automatically linking inputs and outputs of cells appropriately. This creates a SFQHDLC file, which is then compiled into a list of rules. A flowchart showing this algorithm can be seen in Figure 13.

This rule list and the corresponding netlist were then passed back to the AAO tool. The AAO tool runs a time domain simulation within Cadence Spectre and verifies the results of the simulation against the supplied hSFQHDLC rule list. Test vectors are currently specified by the user and read in by the AAO tool via a separate text file.

When the "D" pulse arrives at the beginning of Figure 11, the analyzer recognizes the arrival as an input and detects the switching of junction I0.J2 at 139.3ps. The junction I0.J2 can be read as the junction J2 within the instance I0, where in this case the instance I0 is a D Flip Flop. When the "Release" pulse then arrives at 209.2ps, junction I0.J4 is then seen to switch, causing the output to appear. Another "Release" pulse is then seen to arrive at 413.9ps. Since the flip flop has not been set and there is nothing to release, the pulse that arrives on this input is rejected and junction I0.J3 switches. This happens again with the arrival of the next "Release" pulse. Lastly, two "D" pulses are seen to arrive. As the flip flop has already been set at the arrival of the second "D" pulse, this second "D" pulse is rejected and junction I0.J1 switches.

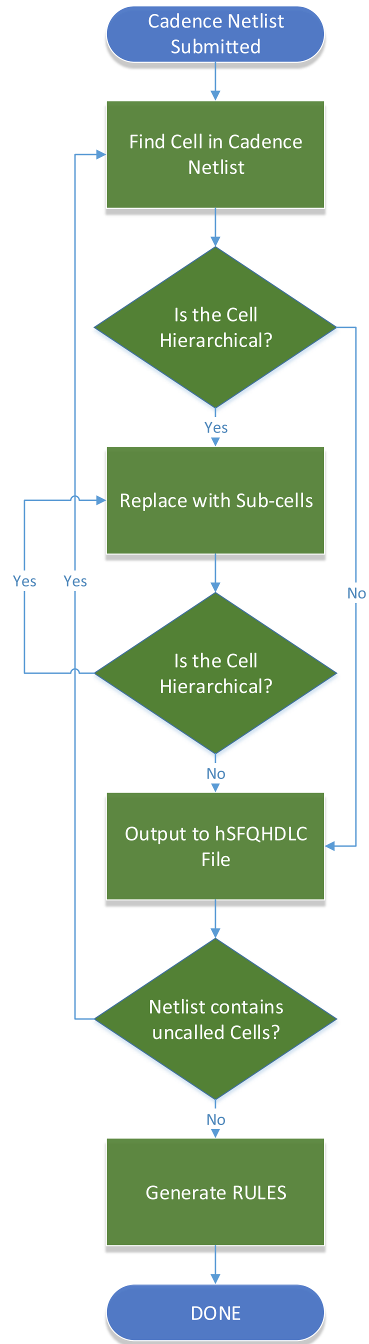

FIGURE 13. Automated Generation tool algorithm overview.

Since the simulation ran to completion without any junction switches out of order, e.g., I0.J2 incrementing without a "D" pulse arriving, any rules left uncompleted, or any timing failures, the circuit is said to have passed. The AAO tool currently reports this to the user via the system command prompt's text output: "Initial simulation is OK".

With the initial simulation having satisfied the rule deck, the AAO tool enters the mode where it begins to optimize the components of the circuit. The initial values of the components can be seen in Table 2, corresponding to Figure 10. The lower margin is the percent that a component value may decrease and the circuit will still work. The upper margin is the amount that a component value may increase and the circuit will still work.

TABLE 2. D flip flop starting values.

Component	Value	Lower Margin (%)	Upper Margin (%)
LD	5.136pH	> 90	12
LQ	3.945pH	10	44
LO	4.084pH	> 90	> 90
LREL	3.82pH	> 90	50
J1	257.5uA	20	4
J2	245uA	70	6
J3	243.75uA	72	14
J4	262.5uA	12	90
I1	150uA	10	42

The AAO tool is a new automated multithreaded critical and global margin optimization tool using Cadence Spectre. Global margin optimization has the largest impact on circuit yield [24], while critical margin optimization is still important for compensating for layout variation and fab errors. The AAO tool can maximize both global and critical margins in a circuit with no user interaction beyond initial setup. A flowchart showing the algorithm of the AAO tool can be seen in Figure 14.

Altered component values are tested in Spectre, and the AAO tool uses hSFQHDLC as well as input/output pulse verification to determine if the circuit operated correctly. The new global margin optimization algorithm maximizes the range that all inductors, bias currents, and critical currents can deviate from nominal values by user set margin targets while still resulting in a properly functioning circuit. The global margin is calculated for a component type by sweeping all component values of that type up and down by the same fraction from the nominal values. Critical margins are the upper and lower bound at which the circuit will still operate properly as each component's value is swept up and down individually, while all other components are held at nominal values. The AAO tool user specifies in a setup file critical and global margin targets for each component class.

The AAO tool's critical and global optimization algorithms are centered around the failure data provided by the hSFQHDLC rule set, which provide starting points in the circuit for the tool to work with. When calculating margins, after maximum and minimum margins of a component or component type are found, all junction failures that occur just beyond the operating margins are recorded. This includes junctions with required quantum phase changes that were not completed and junctions that had unexpected quantum phase changes at the time of circuit failure, determined by checking the simulation results against the hSFQHDLC. This typically is 1-3 junctions. These junctions provide a starting point for the AAO Tool to begin altering component values to look for margin improvement. The optimizer will begin by first only altering the values of this starting group of junctions, but over time will expand this failure list to also include all directly connected components, including those that are not junctions. When margin improvement is found, the list of components to be altered is set to whatever junctions functioned incorrectly according to the hSFQHDLC rules in the new improved circuit. The component type for global optimization is selected automatically based upon which group has the lowest global margin. Individual components are selected for critical margin optimization based upon how close the component's margins are to the minimum margin. Multiple components can be selected for optimization, and the component values from their failure lists will all be modified as a group.

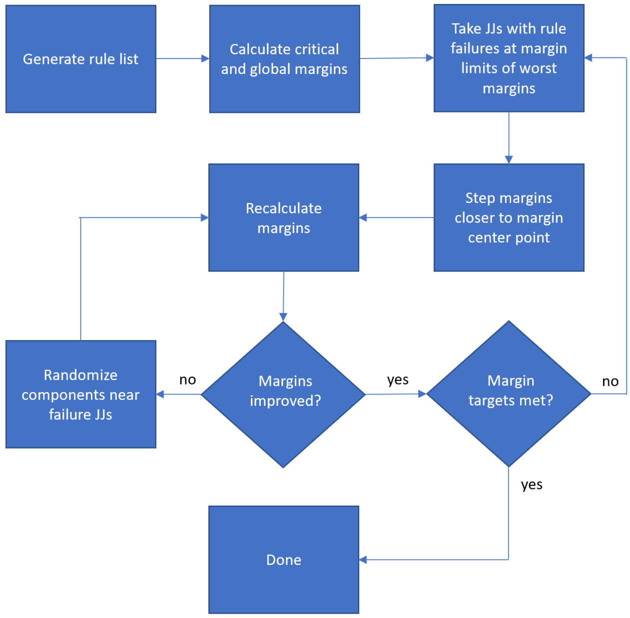

FIGURE 14. Optimization tool overview.

The results seen in Table 3 include parasitic inductors between a Josephson junction and the ground plane of .132p F. JTLs were present on all inputs and outputs.

B. PHASE DETECTOR

One common issue that a designer can run into when performing analysis and optimization on a circuit is that once a circuit is verified as having failed to have matched the provided description, the designer must then fix the circuit manually in order to make the circuit match the description provided to the verification tool. This can prove to be a very time consuming

TABLE 3. D flip flop optimized values.

Component	Value	Lower Margin (%)	Upper Margin (%)
LD	2.037pH	> 90	> 90
LQ	6.39pH	46	73
LO	4.34pH	> 90	> 90
LREL	2.092pH	> 90	> 90
J1	239.2uA	48	55
J2	176.6uA	77	66
J3	223.4uA	45	50
J4	247.9uA	55	45
I1	144.9uA	> 90	> 90

process, especially in larger and complex circuits. The AAO tool has a new and novel feature in which it will automatically take steps to change the component values to bring the circuit into compliance with the supplied hSFQHDLC rule list.

Consider the circuit seen in Figure 15, which represents a portion of the SFQ equivalent of a Bang-Bang style phase detector. This cell is made of two resettable D Flip-Flops, four splitters, and two JTLs [5].

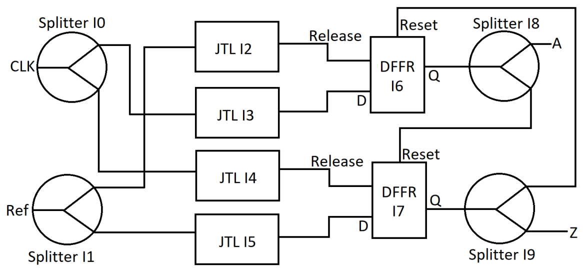

FIGURE 15. Phase detector schematic.

This Cadence generated schematic for this circuit was input into the hSFQHDLC automated generation tool, which generated the hSFQHDLC description of the circuit. The hSFQHDLC compiler then automatically compiled that hSFQHDLC description into a hSFQHDLC rule list.

This rule list and the corresponding Cadence netlist were then passed to the AAO tool. The AAO tool failed in its time domain simulation when it detected a failure on a rule with the DFFR labeled I7. The arrival of the "Release" pulse is supposed to cause the incrementation of junction I7.J4 and then junction I7.J6. This failure can be seen in Figure 16, which shows the arrival of the "D" pulse, the arrival of the "Release" pulse, and the appropriate N-function of the DFFR. The AAO then began to modify the circuit automatically to create a circuit which matched the provided circuit description. The optimizer uses the hSFQHDLC rules to determine the points of failure within the circuit by forming a

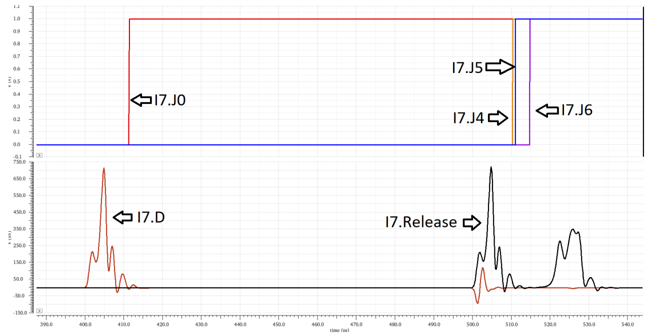

FIGURE 16. Incorrect action of the DFFR I7.J5 should not increment.

list of JJs with active hSFQHDLC rules when circuit failure was identified. Any JJ which failed to activate by having a required INC or DEC effect at the time of failure is considered to be a point of failure. The optimizer will then randomize the component values of all JJs within the points of failure list. The optimizer considers the circuit to be improved when the number of successful JJ INCs and DECs is greater than that of the unmodified circuit. Once improvement is found, the list of points of failure is cleared, and new points of failure are identified based upon the improved circuit. Component value randomization is now continued on this new list of points of failure.

If circuit improvement cannot be found, the list of randomized component values will spread to also include all components currently directly connected to those components which were already being randomized. This list of components are expanded slowly over time in the same manner until a circuit improvement is found.

The optimizer was able to progressively improve the circuit by making it further down the hSFQHDLC rule chain with successive simulations. The optimizer was able to identify that the critical current for the junction on the release input of the DFFRs needed to be slightly raised in order to correct

TABLE 4. COWBoy D flip flop starting values.

Component	Value	Lower Margin (%)	Upper Margin (%)
LD	5.136pH	> 90	11.2
LQ	3.945pH	6.6	43.6
LO	4.34pH	> 90	79.7
LREL	3.82pH	> 90	46.9
J1	257.5uA	18.3	4.2
J2	245uA	57.7	6.1
J3	243.75uA	62.3	13.6
J4	262.5uA	7	75.9
I1	150uA	9.4	30.9

for the rule failure. This created a circuit which matched the circuit description, allowing the circuit to be optimized. This corrected action of the DFFR can be seen in Figure 17.

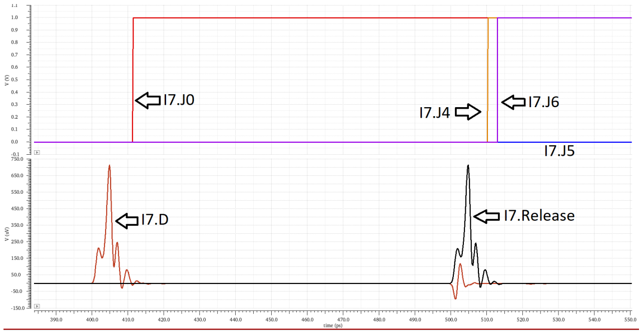

FIGURE 17. Correct action of the DFFR I7. I7J5 does not increment.

C. 32 BIT BDD ADDER

To further showcase the scalability and hierarchical capabilities of this tool, a 32 bit Binary Decision Diagram Adder (Figure 18) was created using two sixteen bit BDD Adders. Each sixteen bit adder was constructed from two eight bit adders. Each eight bit BDD Adder was in turn constructed from two four bit BDD Adders. Two different four bit adders were constructed, one from three BDD Adder [33] cells and one BDD Half Adder [33] cell. The other four bit adder cell was constructed from four BDD Adder cells.

This structure contains a total of 3592 junctions and five levels of hierarchy. The only cells in this structure with designer defined hSFQHDLC rules are the BDD, confluence buffer, splitter, and JTL. These cells are contained within the lowest level of the hierarchy. All rules for the upper level structures, such as the four; sixteen; and 32-bit adders, were automatically generated by the hSFQHDLC compiler. After compilation, the rule deck for this circuit contained 2518 rules that described the behavior of the circuit.

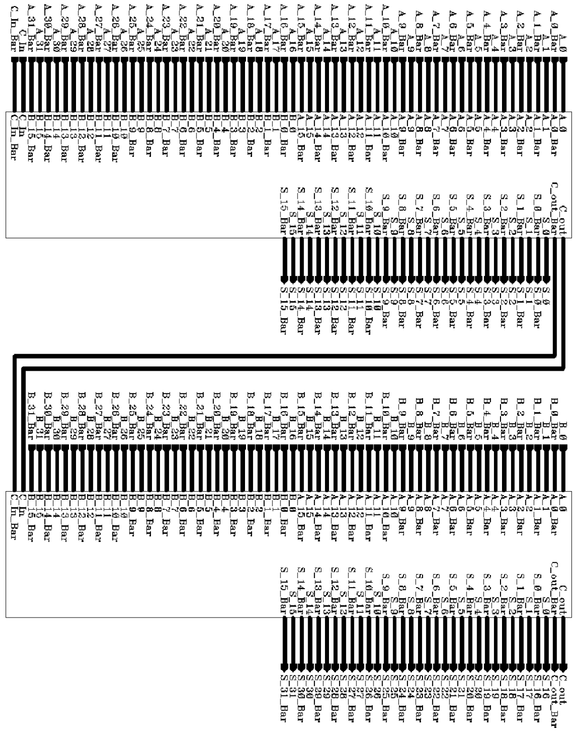

FIGURE 18. 32 Bit BDD adder constructed from two 16 bit BDD adders.

V. COMPARISON TO EXISTING OPTIMIZATION TOOLS

One of the main SFQ optimization tools currently being used is COWBoy using the PSCAN or PSCAN2 circuit simulator. COWBoy was run on the circuit seen in Figure 10 using PSCAN, which produced the initial margin table seen in Table 4. Results should not vary using COWBoy with PSCAN or PSCAN2.

This initial margin table pretty closely agrees with the initial margin table produced by the AAO tool presented in

TABLE 5. D flip flop optimized values comparison.

Component	COWBoy Value	COWBoy Lower Margin (%)	COWBoy Upper Margin (%)	AAOT Value	AAOT Lower Margin (%)	AAOT Upper Margin (%)
LD	4.00pH	> 90	> 90	2.037pH	> 90	> 90
LQ	5.34pH	45.9	> 90	6.39pH	46	73
LO	5.82pH	> 90	> 90	4.34pH	> 90	> 90
LREL	2.83pH	> 90	> 90	2.092pH	> 90	> 90
J1	206uA	38.9	36.6	239.2uA	48	55
J2	150.6uA	> 90	61.9	176.6uA	77	66
J3	239.8uA	38.9	36.6	223.4uA	45	50
J4	299.5uA	48	39.6	247.9uA	55	45
I1	122.4uA	79.7	> 90	144.9uA	> 90	> 90

```
NOR I0(.A(d[2]),.B(n2),.Z(n8),.A_b(d_b[2]),.B_b(n22_b),.Z_b(n8_b));
NOR I1(.A(n24),.B(d[0]),.Z(n9),.A_b(n33),.B_b(d_b[0]),.Z_bar(n9b));
OR I2(.A(n9),.A_b(n9b),.B(n25),.B_b(n25_b),.Z(t1),.Z_bar(t1));
OR I3(.A(t1),.A_bar(t1),.B(n8),.B_bar(n8_b),.Z(n10),.Z_bar(n10));
OR I4(.A(d[3]),.B(n18),.Z(n11),.A_b(d_b[3]),.B_b(n18_b),.Z_b(n11));
NAND I5(.A(n2),.B(d[1]),.Z(n13),.A_b(n20),.B_b(d_b[1]),.Z_b(n56));
NAND I6(.A(n15),.A_b(n15_b),.B(n13),.B_b(n56),.Z(t0),.Z_bar(t0_b));
NAND I7(.A(t0),.A_b(t0_b),.B(n11),.B_b(n11),.Z(n14),.Z_b(n14_b));
NAND I8(.A(n10),.B(n14),.Z(q),.A_b(n10),.B_b(n14_b),.Z_b(q_b));
SPLITTER I9(.A(select[1]),.Y(n16),.Z(n2));
SPLITTER I10(.A(select[0]),.Y(n25),.Z(n15));
SPLITTER I12(.A(n17),.Y(n2),.Z(n18));
SPLITTER I13(.A(n16),.Y(n17),.Z(n24));
SPLITTER I14(.A(select_b[1]),.Y(n16_b),.Z(n20));
SPLITTER I15(.A(select_b[0]),.Y(n25_b),.Z(n15_b));
SPLITTER I17(.A(n17_b),.Y(n22_b),.Z(n18_b));
SPLITTER I16(.A(n16_b),.Y(n17_b),.Z(n33));
```

FIGURE 19. Multiplexer structural netlist.

```
RULE LINK0 (INC(I9.J1)) INC(I12.J0);
RULE LINK1 (INC(I9.J2)) INC(I5.J0);
RULE LINK2 (INC(I10.J1)) INC(I2.J1);
RULE LINK3 (INC(I10.J2)) INC(I6.J0);
RULE LINK4 (INC(I11.J1)) INC(I0.J1);
RULE LINK5 (INC(I11.J2)) INC(I4.J23);
RULE LINK6 (INC(I12.J1)) INC(I11.J0);
RULE LINK7 (INC(I12.J2)) INC(I1.J0);
RULE LINK8 (INC(I13.J1)) INC(I16.J0);
RULE LINK9 (INC(I13.J2)) INC(I5.J23);
RULE LINK10 (INC(I14.J1)) INC(I2.J20);
RULE LINK11 (INC(I14.J2)) INC(I6.J23);
RULE LINK12 (INC(I15.J1)) INC(I0.J20);
RULE LINK13 (INC(I16.J1)) INC(I15.J0);
RULE LINK14 (INC(I16.J2)) INC(I1.J23);
RULE SPLIT0(INC(I9.J0)) INC(I9.J1) && INC(I9.J2) : 2p;
RULE SPLIT1(INC(I10.J0)) INC(I10.J1) && INC(I10.J2) : 2p;
RULE SPLIT2(INC(I11.J0)) INC(I11.J1) && INC(I11.J2) : 2p;
RULE SPLIT3(INC(I12.J0)) INC(I12.J1) && INC(I12.J2) : 2p;
RULE SPLIT4(INC(I13.J0)) INC(I13.J1) && INC(I13.J2) : 2p;
RULE SPLIT5(INC(I14.J0)) INC(I14.J1) && INC(I14.J2) : 2p;
RULE SPLIT6(INC(I15.J0)) INC(I15.J1) && INC(I15.J2) : 2p;
RULE SPLIT7(INC(I16.J0)) INC(I16.J1) && INC(I16.J2) : 2p;
RULE (INC(I5.J20) && !INC(I5.J1)) INC(I5.J5) && INC(I5.J7):20p;
RULE (INC(I5.J1) && !INC(I5.J20)) INC(I5.J3) && INC(I5.J7):20p;
RULE (INC(I5.J20) && INC(I5.J1)) INC(I5.J7):20p;
RULE (INC(I5.J0) && !INC(I5.J23)) INC(I5.J24) && INC(I5.J25):20p;
RULE (INC(I5.J23) && !INC(I5.J0)) INC(I5.J22) && INC(I5.J25):20p;
RULE (INC(I5.J0) && INC(I5.J23)) INC(I5.J25):20p;
RULE (INC(I5.J9) && (N(I5.J17) == N(I5.J18))) INC(I5.J17):20p;
…
RULE (INC(I4.J25) && INC(I4.J7)) INC(I4.J14):20p;
```

FIGURE 20. Multiplexer hSFQHDLC rule list.

```
jtl I20 (.jtl_in(n38),.jtl_out(net41));
jtl I31 (.jtl_in(net48),.jtl_out(Sum_Bar));
jtl I21 (.jtl_in(net27),.jtl_out(net57));
jtl I23 (.jtl_in(C_in),.jtl_out(n60));
jtl I30 (.jtl_in(net47),.jtl_out(Sum));
jtl I18 (.jtl_in(net40),.jtl_out(net52));
jtl I29 (.jtl_in(net46),.jtl_out(C_out));
jtl I28 (.jtl_in(net49),.jtl_out(C_out_Bar));
jtl I24 (.jtl_in(C_in_Bar),.jtl_out(n61));
jtl I19 (.jtl_in(n58),.jtl_out(net30));
jtl I26 (.jtl_in(B),.jtl_out(net54));
jtl I27 (.jtl_in(B_Bar),.jtl_out(net53));
jtl I25 (.jtl_in(A_Bar),.jtl_out(net59));
jtl I22 (.jtl_in(A),.jtl_out(net62));
con_buf I16 (.con_in1(n42),.con_in2(net51),.con_out(net48));
con_buf I15 (.con_in1(net50),.con_in2(net45),.con_out(net47));
con_buf I6 (.con_in1(n29),.con_in2(net26),.con_out(net32));
con_buf I17 (.con_in1(net44),.con_in2(net35),.con_out(net46));
con_buf I9 (.con_in1(net56),.con_in2(net55),.con_out(n33));
con_buf I14 (.con_in1(net43),.con_in2(net31),.con_out(net49));
bdd I0 (.D(net21),.D_Bar(net24),.Q(net27),.Q_Bar(n29),.Root(n60));
bdd I2 (.D(net34),.D_Bar(n37),.Q(n38),.Q_Bar(net40),.Root(net32));
bdd I1 (.D(net22),.D_Bar(net23),.Q(net26),.Q_Bar(n58),.Root(n61));
bdd I3 (.D(net36),.D_Bar(net39),.Q(net45),.Q_Bar(n42),.Root(n33));
splitter I13 (.split_in(net41),.out_a(net51),.out_b(net44));
splitter I8 (.split_in(net57),.out_a(net35),.out_b(net56));
splitter I12 (.split_in(net52),.out_a(net50),.out_b(net43));
splitter I7 (.split_in(net30),.out_a(net55),.out_b(net31));
splitter I5 (.split_in(net59),.out_a(net23),.out_b(net24));
splitter I10 (.split_in(net54),.out_a(net34),.out_b(net36));
splitter I4 (.split_in(net62),.out_a(net21),.out_b(net22));
splitter I11 (.split_in(net53),.out_a(net39),.out_b(n37));
```

FIGURE 21. 4-Bit adder structural netlist.

this paper, showing that these two tools agree on the point of starting margins. COWBoy was then allowed to optimize the margins of the circuit, the results of which can be seen in Table 5.

```
RULE LINK0 (INC(I20.X1)) INC(I13.J0);
RULE LINK1 (INC(I21.X1)) INC(I8.J0);
RULE LINK2 (INC(I23.X1)) INC(I0.X0);
RULE LINK3 (INC(I18.X1)) INC(I12.J0);
RULE LINK4 (INC(I24.X1)) INC(I1.X0);
RULE LINK5 (INC(I19.X1)) INC(I7.J0);
RULE LINK6 (INC(I26.X1)) INC(I10.J0);
RULE LINK7 (INC(I27.X1)) INC(I11.J0);
RULE LINK8 (INC(I25.X1)) INC(I5.J0);
RULE LINK9 (INC(I22.X1)) INC(I4.J0);
RULE LINK10 (INC(I16.J5)) INC(I31.X0);
RULE LINK11 (INC(I15.J5)) INC(I30.X0);
RULE LINK12 (INC(I6.J5)) INC(I2.X0);
RULE LINK13 (INC(I17.J5)) INC(I29.X0);
RULE LINK14 (INC(I9.J5)) INC(I3.X0);
RULE LINK15 (INC(I14.J5)) INC(I28.X0);
RULE LINK16 (INC(I0.X4)) INC(I21.X0);
RULE LINK17 (INC(I0.X1)) INC(I6.J3);
RULE LINK18 (INC(I2.X4)) INC(I20.X0);
RULE LINK19 (INC(I2.X1)) INC(I18.X0);
RULE LINK20 (INC(I1.X4)) INC(I6.J1);
RULE LINK21 (INC(I1.X1)) INC(I19.X0);
RULE LINK22 (INC(I3.X4)) INC(I15.J1);
RULE LINK23 (INC(I3.X1)) INC(I16.J3);
RULE LINK24 (INC(I13.J1)) INC(I16.J1);
RULE LINK25 (INC(I13.J2)) INC(I17.J3);
RULE LINK26 (INC(I8.J1)) INC(I17.J1);
RULE LINK27 (INC(I8.J2)) INC(I9.J3);
RULE LINK28 (INC(I12.J1)) INC(I15.J3);
RULE LINK29 (INC(I12.J2)) INC(I14.J3);
RULE LINK30 (INC(I7.J1)) INC(I9.J1);
RULE LINK31 (INC(I7.J2)) INC(I14.J1);
RULE GO0(INC(I20.X0)) INC(I20.X1);
RULE GO1(INC(I31.X0)) INC(I31.X1);
RULE GO2(INC(I21.X0)) INC(I21.X1);
RULE GO3(INC(I23.X0)) INC(I23.X1);
RULE GO4(INC(I30.X0)) INC(I30.X1);
RULE GO5(INC(I18.X0)) INC(I18.X1);
RULE GO6(INC(I29.X0)) INC(I29.X1);
RULE GO7(INC(I28.X0)) INC(I28.X1);
RULE GO8(INC(I24.X0)) INC(I24.X1);
RULE GO9(INC(I19.X0)) INC(I19.X1);
RULE GO10(INC(I26.X0)) INC(I26.X1);
RULE GO11(INC(I27.X0)) INC(I27.X1);
RULE GO12(INC(I25.X0)) INC(I25.X1);
RULE GO13(INC(I22.X0)) INC(I22.X1);
RULE GO1_14(INC(I16.J1)) INC(I16.J4) && INC(I16.J5);
RULE GO2_15(INC(I16.J3)) INC(I16.J2) && INC(I16.J5);
RULE GO1_16(INC(I15.J1)) INC(I15.J4) && INC(I15.J5);
RULE GO2_17(INC(I15.J3)) INC(I15.J2) && INC(I15.J5);
RULE GO1_18(INC(I6.J1)) INC(I6.J4) && INC(I6.J5);
RULE GO2_19(INC(I6.J3)) INC(I6.J2) && INC(I6.J5);
RULE GO1_20(INC(I17.J1)) INC(I17.J4) && INC(I17.J5);
RULE GO2_21(INC(I17.J3)) INC(I17.J2) && INC(I17.J5);
RULE GO1_22(INC(I9.J1)) INC(I9.J4) && INC(I9.J5);
RULE GO2_23(INC(I9.J3)) INC(I9.J2) && INC(I9.J5);
RULE GO1_24(INC(I14.J1)) INC(I14.J4) && INC(I14.J5);
…
RULE SPLIT26(INC(I13.J0)) INC(I13.J1) && INC(I13.J2) : 10p;
RULE SPLIT27(INC(I8.J0)) INC(I8.J1) && INC(I8.J2) : 10p;
RULE SPLIT28(INC(I12.J0)) INC(I12.J1) && INC(I12.J2) : 10p;
RULE SPLIT29(INC(I7.J0)) INC(I7.J1) && INC(I7.J2) : 10p;
RULE SPLIT30(INC(I5.J0)) INC(I5.J1) && INC(I5.J2) : 10p;
RULE SPLIT31(INC(I10.J0)) INC(I10.J1) && INC(I10.J2) : 10p;
RULE SPLIT32(INC(I4.J0)) INC(I4.J1) && INC(I4.J2) : 10p;
RULE SPLIT33(INC(I11.J0)) INC(I11.J1) && INC(I11.J2) : 10p;
```

FIGURE 22. 4-Bit adder hSFQHDLC rules.

This table shows that the margins calculated by the AAO tool presented in this paper are superior for this test case. COWBoy was not run on the circuit presented in Figure 15, as COWBoy does not have the ability to modify circuits that fails to match the provided hSFQHDL.

VI. TRANSLATION FROM STRUCTURAL NETLIST

It is possible to convert a structural netlist to hSFQHDLC. In an automatically placed and routed circuits, this is useful due to the lack of a Cadence netlist or schematic but the availability of a structural netlist, also known as structural Verilog. This hSFQHDLC description can then be compiled into a description of the correct circuit behavior for that circuit.

A. MULTIPLEXER

As an example of structural netlist to hSFQHDLC conversion take the structural netlist seen in Figure 19, which represents a SFQ multiplexer. The SFQ multiplexer was constructed out of 8 splitters, 4 NAND gates, 2 OR gates, and 2 NOR gates.

The multiplexer has the following structural netlist:

This netlist was input into a program which automatically converts a structural netlist to a hSFQHDLC rule list. The hSFQHDLC rule list can be seen in Figure 20.

This rule list, containing a total of 149 rules, is a comprehensive description of the correct behavior of the SFQ multiplexer circuit. Rules can be seen that correspond to instances in the structural netlist, such as RULE SPLIT0 which corresponds to the splitter I9. This rule then activates rules RULE LINK0 and RULE LINK1. These rules then activate further rules. In this manner the complete behavior of the structural netlist in Figure 17 is described.

B. 4 BIT ADDER

A BDD based differential 4 bit adder netlist can be seen in Figure 21. This circuit was constructed from 14 JTLs, 6 confluence buffers, 4 BDDs, and 8 splitters.

This adder has the following structural netlist:

This netlist was input into the program and automatically converted to a hSFQHDLC rule list containing 98 rules. The hSFQHDLC rule list can be seen in Figure 22.

VII. CONCLUSION

Hierarchical SFQHDLC provides several new features that have not been previously seen in SFQ design. These include conversion from a structural netlist, compilation from a Cadence netlist, and integration with a new and improved optimization tool. The margin optimization tool is a novel approach to circuit optimization with new features such as parallelization, optimization of circuits that do not match the provided rule list, and leverages Cadence Spectre as a simulation tool. When this is combined with hSFQHDLC, designers can quickly and accurately generate a list of rules for any circuit in a Cadence design environment. This combination of tools allows verification with much higher accuracy than simple input/output checking. If failure does occur, the exact point and reason of failure is reported back to the designer and the tool attempts to create a working circuit without designer effort.

FUTURE WORK: The RPI AAO Tool which utilizes hSFQHDLC is in the process of being expanded. Margin optimization is being improved to offer better optimization results with larger circuits. Cadence SKILL code is being written for optimizer setup and run commands, directly integrating into the Cadence graphical user interface (GUI).

ACKNOWLEDGEMENT

The authors would like to thank John Bulzacchelli for providing the COWBoy results seen in this paper. Thanks go out to Gerald Gibson for his support of the research group at RPI.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

REFERENCES

[1] C. J. Fourie and M. H. Volkmann, "Status of superconductor electronic circuit design software," *IEEE Trans. Appl. Supercond.*, vol. 23, no. 3, Jun. 2013, Art. no. 1300205. doi: 10.1109/tasc.2012.2228732.

[2] "Superconducting technology assessment," Nat. Secur. Agency Office Corporate Assessments, Defense Tech. Inf. Center, Fort Belvoir, VA, USA, Tech. Rep., Aug. 2005.

[3] A. Silver, P. Bunyk, A. Kleinsasser, and J. Spargo, "Vision for single flux quantum very large scale integrated technology," *Superconductor Sci. Technol.*, vol. 19, no. 5, pp. S307–S311, Mar. 2006. doi: 10.1088/0953-2048/19/5/S30.

[4] PSCAN2. (0.1.15). *Pavel Shevchenko*. Accessed: Oct. 5, 2017. [Online]. Available: http://pscan2sim.org/

[5] 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. doi: 10.1109/77.80745.

[6] S. Yorozu, Y. Kameda, H. Terai, A. Fujimaki, T. Yamada, and S. Tahara, "A single flux quantum standard logic cell library," *Phys. C, Supercond.*, vols. 378–3812, pp. 1471–1474, Oct. 2001. doi: 10.1016/S0921-4534(02)01759-8.

[7] D. S. Holmes, A. L. Ripple, and M. A. Manheimer, "Energy-efficient superconducting computing—Power budgets and requirements," *IEEE Trans. Appl. Supercond.*, vol. 23, no. 3, Jun. 2013, Art. no. 1701610. doi: 10.1109/TASC.2013.2244634.

[8] O. Mukhanov, "Digital Processing," in *Applied Superconductivity: Handbook on Devices and Applications*, P. Seidel, Ed. Weinheim, Germany: Wiley, 2015, pp. 1135–1162.

[9] S. V. Rylov and R. P. Robertazzi, "Superconducting high-resolution A/D converter based on phase modulation and multichannel timing arbitration," *IEEE Trans. Appl. Supercond.*, vol. 5, no. 2, pp. 2260–2263, Jun. 1995. doi: 10.1109/77.403035.

[10] M. Ito, K. Kawasaki, N. Yoshikawa, A. Fujimaki, H. Terai, and S. Yorozu, "20 GHz operation of bit-serial handshaking systems using asynchronous SFQ logic circuits," *IEEE Trans. Appl. Supercond.*, vol. 15, no. 2, pp. 255–258, Jun. 2005. doi: 10.1109/TASC.2005.849773.

[11] T. V. Filippov *et al.*, "20 GHz operation of an asynchronous wave-pipelined RSFQ arithmetic-logic unit," *Phys. Procedia*, vol. 36, pp. 59–65, 2012. doi: 10.1109/TASC.2013.2240755.

[12] Y. Nobumori *et al.*, "Design and implementation of a fully asynchronous SFQ microprocessor: SCRAM2," *IEEE Trans. Appl. Supercond.*, vol. 17, no. 2, pp. 478–481, Jun. 2007. doi: 10.1109/TASC.2007.898658.

[13] A. Fujimaki, "High-speed digital circuits," in *100 Years of Superconductivity*, H. Rogalla and P. H. Kes, Eds. Boca Raton, FL, USA: CRC Press, 2011, pp. 431–440.

[14] D. E. Kirichenko, S. Sarwana, and A. F. Kirichenko, "Zero static power dissipation biasing of RSFQ circuits," *IEEE Trans. Appl. Supercond.*, vol. 21, no. 3, pp. 776–779, Jun. 2011. doi: 10.1109/TASC.2010.2098432.

[15] O. A. Mukhanov, "Energy-efficient single flux quantum technology," *IEEE Trans. Appl. Supercond.*, vol. 21, no. 3, pp. 760–769, Jun. 2011. doi: 10.1109/TASC.2010.2096792.

[16] M. Tanaka, A. Kitayama, T. Koketsu, M. Ito, and A. Fujimaki, "Low-energy consumption RSFQ circuits driven by low voltages," *IEEE Trans. Appl. Supercond.*, vol. 23, no. 3, Jun. 2013, Art. no. 1701104. doi: 10.1109/TASC.2013.2240555.

[17] M. Tanaka, M. Ito, A. Kitayama, T. Kouketsu, and A. Fujimaki, "18-GHz, 4.0-aJ/bit operation of ultra-low-energy rapid single-flux-quantum shift registers," *Jpn. J. Appl. Phys.*, vol. 51, no. 5R, May 2012, Art. no. 053102. doi: 10.1143/JJAP.51.053102.

[18] M. H. Volkmann, A. Sahu, C. J. Fourie, and O. A. Mukhanov, "Implementation of energy efficient single flux quantum digital circuits with sub-aJ/bit operation," *Superconductor Sci. Technol.*, vol. 26, no. 1, Nov. 2012, Art. no. 015002. doi: 10.1088/0953-2048/26/1/015002.

[19] A. F. Kirichenko, I. V. Vernik, J. A. Vivalda, R. T. Hunt, and D. T. Yohannes, "ERSFQ 8-bit parallel adders as a process benchmark," *IEEE Trans. Appl. Supercond.*, vol. 25, no. 3, Jun. 2015, Art. no. 1300505. doi: 10.1109/TASC.2014.2371875.

[20] A. F. Kirichenko, I. F. Vernik, O. A. Mukhanov, and T. A. Ohki, "ERSFQ 4-to-16 decoder for energy-efficient RAM," *IEEE Trans. Appl. Supercond.*, vol. 25, no. 3, Jun. 2015, Art. no. 1301304. doi: 10.1109/TASC.2014.2385479.

[21] S. V. Polonsky, V. K. Semenov, and P. N. Shevchenko, "PSCAN: Personal superconductor circuit analyser," *Superconductor Sci. Technol.*, vol. 4, no. 11, pp. 667–670, Nov. 1991. doi: 10.1088/0953-2048/4/11/031.

[22] S. Polonsky, P. Shevchenko, A. Kirichenko, D. Zinoviev, and A. Rylyakov, "PSCAN'96: New software for simulation and optimization of complex RSFQ circuits," *IEEE Trans. Appl. Supercond.*, vol. 7, no. 2, pp. 2685–2689, Jun. 1997.

[23] T. Harnisch, J. Kunert, H. Toepfer, and H. F. Uhlmann, "Design centering methods for yield optimization of cryoelectronic circuits," *IEEE Trans. Appl. Supercond.*, vol. 7, no. 2, pp. 3434–3437, Jun. 1997. doi: 10.1109/77.622122.

[24] Q. P. Kerr and M. J. Feldman, "Multiparameter optimization of RSFQ circuits using the method of inscribed hyperspheres," *IEEE Trans. Appl. Supercond.*, vol. 5, no. 2, pp. 3337–3340, Jun. 1995. doi: 10.1109/77.403306.

[25] H. Hayakawa, N. Yoshikawai, S. Yorozu, and A. Fujimak, "Superconducting digital electronics," *Proc. IEEE*, vol. 92, no. 10, pp. 1549–1563, Oct. 2004. doi: 10.1109/JPROC.2004.833658.

[26] Spectre. (MMSIM15.1). *Cadence Design Systems, Inc*. Accessed: Oct. 5, 2017. [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/custom-ic-analog-rf-design/circuit-simulation/spectre-circuit-simulator.html

[27] E. S. Fang and T. V. Duzer, "A Josephson integrated circuit simulator (JSIM) for superconductive electronics application," in *Proc. Extended Abstr. Int. Supercond. Electron. Conf. (ISEC)*, Tokyo, Japan, Jun. 1989, pp. 407–410.

[28] WRspice Circuit Simulator. (4.3.8). *Whiteley Research Incorporated*. Accessed: Oct. 5, 2017. [Online]. Available: http://wrcad.com/wrspice.html

[29] J. G. Rollins, "Numerical simulator for superconducting integrated circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 2, pp. 245–251, Feb. 1991. doi: 10.1109/43.68411.

[30] L. N. Cooper, "Bound electron pairs in a degenerate Fermi gas," *Phys. Rev.*, vol. 104, no. 4, pp. 1189–1190, Nov. 1956. doi: 10.1103/PhysRev.104.1189.

[31] K. K. Likharev, *Dynamics of Josephson Junctions and Circuits*. Philadelphia, PA, USA: Gordon & Breach, 1984.

[32] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516, Jun. 1978. doi: 10.1109/TC.1978.1675141.

[33] N. Yoshikawa, H. Tago, and K. Yoneyama, "A new design approach for RSFQ logic circuits based on the binary decision diagram," *IEEE Trans. Appl. Supercond.*, vol. 9, no. 2, pp. 3161–3164, Jun. 1999. doi: 10.1109/77.783700.

ANDREW M. HASLAM (S'10) received the B.S. and M.S. degrees in electrical engineering from the Rensselaer Polytechnic Institute, Troy, NY, USA, in 2014 and 2017, respectively, where he is currently pursuing the Ph.D. degree in electrical engineering with the Department of Electrical and Computer Systems Engineering. His current research interests include single-flux quantum superconducting circuits, superconducting CAD tools, and high-speed SiGe circuits.

KURT M. ENGLISH (S'16) received the dual B.S. degree in electrical engineering and computer systems engineering from the Rensselaer Polytechnic Institute (RPI), Troy, NY, USA, in 2014, where he is currently pursuing the Ph.D. degree in computer systems engineering with the Department of Electrical and Computer Systems Engineering.

His current research interests include high-performance computing and single-flux quantum superconducting circuits.

ALEXANDER DERRICKSON (S'17) received the B.S. degree in computer and systems engineering from the Rensselaer Polytechnic Institute (RPI), Troy, NY, USA, in 2017, where he is currently pursuing the Ph.D. degree in computer and systems engineering.

His current research interests include high-performance computing, SiGe, and semiconductor device physics.

JOHN F. MCDONALD (S'63–M'71–SM'01–LSM'10) received the B.S.E.E. degree from the Massachusetts Institute of Technology, Cambridge, in 1962, and the M.Eng. and Ph.D. degrees in engineering and applied science from Yale University, New Haven, CT, USA, in 1965 and 1969, respectively.

He was a Member of Technical Staff with Bell Labs, in 1964. He was an Instructor with Yale University, in 1969, and an Assistant Professor, in 1970. In 1974, he joined the Department of Electrical, Computer and Systems Engineering, Faculty at Rensselaer Polytechnic Institute, Troy, NY, USA, as an Associate Professor, where he became a Full Professor, in 1985.