

SFQ回路設計のためのトポロジー制約付き 自動パイプライン合成

小林 晃^{1,a)} 大石 芽吹¹ 高前田 伸也^{1,2,b)}

概要：単一磁束量子 (SFQ) 回路は、超高速・低消費電力で動作する超伝導デバイスである。現在主流となっている CMOS 回路が電圧レベルでブール値を表現するのに対して、SFQ 回路の元となっている SFQ 論理は時間的に伝達されるパルスの有無を用いてブール値を表現する。このパルス由来の計算手法により、SFQ 回路はゲートレベルのパイプライン化やゲートのファンアウト制限など、SFQ 回路特有の制約を持つ。SFQ 回路向けのセルライブラリやシミュレータ、ゲートレベルの回路設計言語が開発されているものの、SFQ 回路を自動で合成する手法は依然として限定されており、結果として SFQ 回路の設計者は SFQ 特有の制約を満たすゲートレベルの回路構成を自力で探索する必要がある。この課題を解決するため、本研究では SFQ 回路設計のためのトポロジー制約付き自動パイプライン合成ツールを提案する。このツールは高レベルでの回路記述を SFQ のドメイン固有言語に変換し、さらに SFQ 特有のゲートレベルパイプライン化およびファンアウト制限といったトポロジー制約を自動で満たすようにする。バイナリニューラルネットワーク用の XNOR-popcount 回路の自動合成による評価から、回路が正しく動作することを実証し、最適化を通して 2.5% のゲート数削減と、30× のコード削減が達成されていることを確認した。

1. はじめに

現在のコンピュータのほとんどは CMOS テクノロジーを用いて作られている。1970 年代以降チップ内に集積されるトランジスタの数はおよそ 2 年ごとに倍増していき、チップの性能は飛躍的に向上している。しかし、物理的・経済的な制限を受けて CMOS 技術のスケールは急激に失速しており、CMOS に代わる計算技術が必要とされている。

その中で現在注目されている技術が超伝導テクノロジー、とりわけ単一磁束量子 (SFQ) 回路 [1–3] である。CMOS 回路が電圧レベルを用いて計算するのに対して、SFQ 回路は量子化された磁束を用いて計算を行い、この過程が電圧パルスとして表現される。ジョセフソン接合のスイッチングの際に消費されるエネルギーは 10^{-19} J 程度であり、CMOS 回路で電圧を保持する際に消費されるエネルギーと比較しても非常に小さい。また、SFQ 回路は数十～数百 GHz という高周波数で動作することができ [4, 5]、従来の CMOS 回路よりも速い。こうした SFQ 回路の特性は超低消費電力で高速なアプリケーション、例えば量子コンピュータにおける制御・エラー訂正回路として注目されて

いる。

このような利点があるにもかかわらず、SFQ 回路は依然として主流にはなっていない。第一に基となるデバイス技術が未発達であるという問題がある。第二に、SFQ 回路はパルスを用いた計算手法に起因する特有の設計制約を持っている。SFQ 回路のほとんどの論理ゲート (AND ゲートや NOT ゲートなど) はクロックパルスを用いてゲートレベルでパイプライン化される必要があり、正しい演算を得るために Delay Flip-Flop (DFF) を追加するなどして厳密にデータパスをバランシングしなくてはならない。また、SFQ パルスは自発的に分裂することがないため、SFQ ゲートのファンアウトは 1 に限定される。そのため、あるゲートの信号を複数のゲートが使用する場合には、分割ゲート (SPLIT) を明示的に挿入する必要がある。これらの制約は回路設計のオーバーヘッドを増加させ、人力での回路設計を非常に複雑で困難なものにさせる。

CMOS 技術においては高位合成ツールがハードウェア設計のために広く用いられている。設計者は C や C++ のような高級言語で回路のアルゴリズム的な振る舞いを記述し、それを高位合成ツールが自動でクロックサイクルレベルでの命令スケジューリングをしながらレジスタ転送レベル (RTL) の記述に変換する。これによって回路設計者はタイミングレベルでの設計をする必要がない。これに対し

¹ 東京大学

² 理化学研究所

^{a)} hkobayashi@is.s.u-tokyo.ac.jp

^{b)} shinya@is.s.u-tokyo.ac.jp

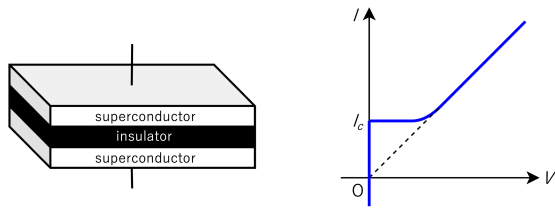


図 1 ジョセフソン接合の構造と電圧-電流特性

て、SFQ 回路向けのコンピュータ支援設計ツールは不十分であり、SFQ 回路の設計者は SFQ 特有の制約を意識しながらクロックを意識したゲートレベルでの設計を手動で行わなくてはならない。

本研究では SFQ 回路を効率的に設計するための、トポロジー制約付き自動パイプライン合成ツールを提案する。提案手法は SFQ 特有のトポロジー制約を、パスバランシングのために DFF を、ファンアウト制限を満たすために SPLIT ゲートをそれぞれ割り当てることで、自動的に解決する。さらに、グラフベースでの最適化を行うことでゲート数を減らし、面積やエネルギー効率を改善させる。提案手法がこれらの過程を自動化させることで、設計者は SFQ 特有の制約を明示的に扱うことなく機能の記述にのみ焦点をあてることができる。

本研究の主な貢献は以下のとおりである。

- 既存のオープンソースの高位合成ツールを拡張した SFQ 回路向けの自動パイプライン合成ツールを提案し、高レベルでの機能的な記述から SFQ 回路構成を生成できるようにする。
- ゲートレベルのパイプライン化およびゲートのファンアウト制限といった SFQ 特有のトポロジー制約を満たす SFQ 回路構成を生成するための合成手法を提案する。
- ケーススタディとして Binarized Neural Network 用の 16 ビット XNOR-popcount 回路を合成し、提案手法が機能的な正しさを保証しつつ、最適化を通して 2.5% のゲート数削減と、30× のコード削減を達成していることを示す。

2. 背景

2.1 SFQ 論理

SFQ 回路の基本構成要素となるのがジョセフソン接合とよばれる超伝導素子である。ジョセフソン接合は図 1 のように二つの超伝導層の間に薄い絶縁層が挟まれた構造となっている。このジョセフソン接合に流れる電流が I_c 以下のとき、ジョセフソン接合は抵抗なしで電流を流す。一方電流の値が I_c を超えるとジョセフソン接合は抵抗状態に切り替わり、絶縁層を磁束が通過する。

図 2 はジョセフソン伝送路 (JTL) と呼ばれる回路の構成を表している。JTL はジョセフソン接合を並べて電源か

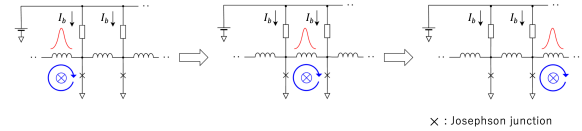


図 2 ジョセフソン伝送路

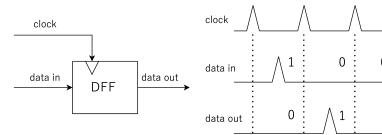


図 3 DFF ゲートの動作

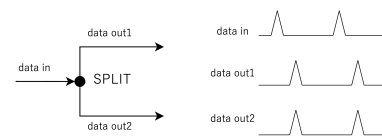


図 4 SPLIT ゲートの動作

らバイアス電流を加えたものとなっている。あるジョセフソン接合が SFQ パルスを受け取ると、パルスによって生じる周回電流とバイアス電流の和が I_c を超え、スイッチして隣のジョセフソン接合にパルスを伝える。SFQ パルスは電圧と時間の積分によって以下のように量子化される。

$$\int V dt = \Phi_0 \simeq 2.07 \text{mV} \cdot \text{ps}. \quad (1)$$

ここで、 Φ_0 は磁束を表す。SFQ 論理はこの量子化された磁束のパルスを計算に用いる。すなわち、パルスがあるとき 1 を、ないとき 0 を表すとして計算を行う。

このようなパルス由来の論理表現のために、SFQ 固有の制約がいくつか生じる。第一に、SFQ 論理では情報は時間的に伝達していくパルスの有無で表されるため、ほとんどの論理ゲート (AND, OR, NOT, ...) はその有無を判別するためにクロック信号を必要とする。そのため、SFQ 回路はゲートレベルでパイプライン化される。さらに、複数の入力を持つゲートは入力タイミングを揃える必要があるが、そのために SFQ 回路は図 3 に表される DFF を挿入する必要がある。第二に、SFQ 論理は情報を量子化されたパルスを用いて表現しているため、信号を複製することが簡単にできない。したがって、SFQ 回路は、「標準的なゲートは単一のゲートにのみ出力することができる」というファンアウト制約を持つ。もしある信号を複数のゲートに用いる場合は、図 4 に表される SPLIT ゲートと呼ばれる特別なゲートが必要になる。一般に、あるゲートから出る信号が n 個のゲートで使用される場合、 $n-1$ 個の SPLIT ゲートを用いて、葉の数が n の SPLIT ツリーが構成される。

2.2 SFQ 回路向けの設計ツール

SFQ 回路は約 4 K という極低温環境で動作するため、実際に SFQ 回路をチップ上で動かすのは困難である。加えて、SFQ 回路はタイミングやパラメータの変動、ジョセフソン接合の非線形ダイナミクスの影響を受けやすく、設計プロセスにおいて高精度なシミュレーションが不可欠である。したがって、シミュレーションツールは SFQ 回路の設計において欠かせない。RSFQlib [6] は SFQ 論理の標準セルライブラリを提供しており、定義済みのゲートを用いて論理レベルでの回路構成を可能にしている。JoSIM [7] はジョセフソン接合を含む超伝導素子を正確にモデル化できる SPICE ベースの回路シミュレータで、SFQ 回路の検証に広く用いられている。

また、一般的な CMOS 用のハードウェア記述言語 (HDL) はクロック分配やパイプライン化されたデータパスの記述には適していないため、正確に SFQ 回路を設計するためには SFQ 専用の HDL が欠かせない。RustSFQ [8] は Rust ベースの SFQ 回路のドメイン固有言語である。RustSFQ は、Rust の所有権と型システムを活用して、ファンアウト制限とワイヤ接続を静的に保証し、コンパイル時点で設計エラーを減らすことができる。

2.3 既存の SFQ 回路合成フレームワーク

これまでに、SFQ 回路の合成フレームワークがいくつか考案されている。SFQmap [9] は SFQ 回路向けのテクノロジーマッピングツールで、特にパスバランシング用の DFF の数と、最悪ケースのステージ遅延と回路の論理深度の積を削減する。qSeq [10] はフィードバックループを含む回路など、SFQ の順序回路を合成するためのアルゴリズムを提案している。Zhang らは、多数決論理 (MAJ) を用いた SFQ 回路向けの新たな論理合成フレームワークを提案した [11]。MAJ ゲートは $MAJ(A, B, C) = AB + BC + AC$ で定義されるゲートで、この 3-MAJ ゲートを一部の論理構造と置き換えることでパスの深さとゲート数を削減する。これらのフレームワークは全てブールネットワークまたは RTL の記述を入力として、SFQ 回路の論理レベル合成と最適化に焦点を当てている。

より低い抽象度レベルで見ると、SFQ 回路では、パルス由来の性質とゲートレベルのパイプライン処理のために、ゲート配置とクロック分配が重要かつ困難な課題となる。CMOS 回路とは異なり、SFQ 回路におけるクロックスキューは単なる動作周波数の低下にとどまらず、パルスの衝突やずれによって回路の動作そのものが意図しないものとなり得る。そこで、いくつかの研究では、物理的な配置と配線情報を明示的に考慮することで、SFQ 回路のクロックツリー合成に取り組んできた。Shahasavani らは、SFQ セルの詳細な物理的配置に基づいた最小スキュークロックツリー合成手法を提案した [12]。Wang らはこのア

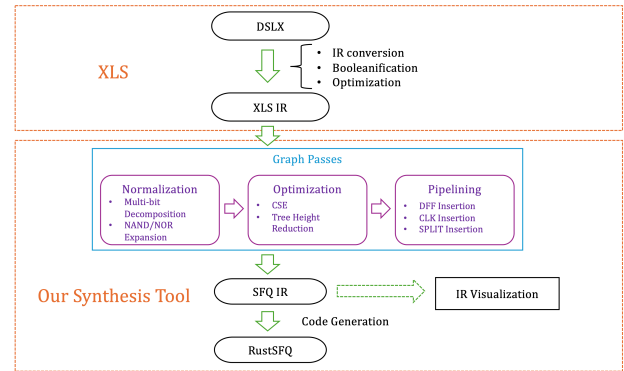


図 5 提案手法の概要図

プローチをさらに拡張して、クロックツリーのトポロジーと物理レイアウトを同時に最適化することでスキューとレイテンシを低減する手法を提案した [13]。これらのアプローチは、セルの配置、配線長、およびデバイスレベルでの遅延に関する正確な情報を必要とするため、物理設計段階で初めて適用可能となる。

3. 提案手法

本節では、SFQ 回路設計のための自動パイプライン合成手法について説明する。図 5 に提案手法の概要を示す。本研究は、主に Google が開発している XLS [14] というオープンソースの高位合成ツールを拡張している。提案手法は、DSLX という XLS 独自のドメイン固有言語を用いた回路記述を独自のグラフベースの中間表現 (IR) に変換する。データ依存グラフを構築する際に、ゲートレベルでのパイプライン処理や出力ポートのファンアウト制限などの、SFQ 固有の制約を明示的に考慮する。最後に、バックエンド出力として RustSFQ [8] を生成する。

3.1 グラフ表現

XLS で生成された IR は有向非巡回グラフに変換される。このグラフにおいて、ノードは IR 中の各演算に対応し、エッジは演算間の依存関係を表している。各ノードは、演算子タイプ、識別子、operand、consumer、depth などといった複数の属性を保持している。演算子集合には AND, OR, NOT, XOR といった基本的な論理演算だけでなく、JTL, DFF, SPLIT といった SFQ 回路用の演算も含まれる。operand 領域には、ノードに入力データを渡すノードへのポインタが格納され、consumer 領域にはノードの出力データを受け取るノードへのポインタが格納される。depth は SFQ 特有のタイミング特性を捉えるために本研究で導入された独自の属性であり、グラフにおいて入力ノードからそのノードへのパスにおける同期ゲートの数を表している。各グラフ変形の際、すべてのノードの depth が以下の式にしたがって計算され更新される。

$$depth(v) = \max_{u \in V: (u,v) \in E} depth(u) + \begin{cases} 1 & (v \text{ は同期ゲート}) \\ 0 & (\text{それ以外}). \end{cases} \quad (2)$$

ここで、 V, E はそれぞれノードとエッジの集合である。本研究では、グラフのトポロジカルな順序関係に基づいて、動的計画法を用いて $depth$ を計算する。この計算全体の時間計算量は $O(|V| + |E|)$ であり、グラフ変形中に繰り返し実行しても十分に高速である。グラフのエッジはノード間の依存関係を表している。すなわち、ソースノードの出力をデスティネーションノードが入力として用いることを示す。提案手法においては、エッジの種類が二つ定義されている。一つ目はデータエッジで、演算間のデータの依存関係を表している。二つ目がクロックエッジで、明示的にクロック信号の伝搬を表現してクロック同期の制約を明確にしている。

3.2 グラフパス

データフローグラフを構築したのち、一連のグラフパスを適用してグラフを変換する。これらのパスは主に、Normalization, Optimization, Pipelining という3つのフェーズに分けられる。

Normalization フェーズでは、IR とデータフローグラフは元の意味を保ったまま、RustSFQ バックエンドで有効な形式に再構築される。例えば、RustSFQ は XLS 組み込みの最適化によって生成される可能性のある NAND 演算や NOR 演算をサポートしていない。そこで、このフェーズで NAND 演算を AND 演算と NOT 演算の組み合わせに、NOR 演算を OR 演算と NOT 演算の組み合わせにそれぞれ展開する。

Optimization フェーズではグラフ最適化を行う。もともと XLS はグラフを生成する際に特定の最適化を行っているが、先述した Normalization パスでグラフ構造が変化するため、当初は達成されていた最適性が崩れてしまう可能性がある。そのため、このフェーズで改めて最適化処理を行い、グラフ表現の質を向上させる。例えば、共通部分式除去 (CSE) 処理では、同一の演算タイプと operand を持つノードをマージする。図に示すように、CSE は同一の演算ノードへのファンインに必要なクロック同期演算ゲートと SPLIT ゲートの数を削減する一方、マージされたノードからのファンアウトのために使用される SPLIT ゲートの数を増加させる可能性がある。通常、クロック同期の演算ゲートは SPLIT ゲートよりも多くのジョセフソン接合を含んでいるため、CSE は一般的には面積とエネルギー消費を削減する。

Pipelining フェーズは本研究独自のものである。このフェーズは DFF 挿入や SPLIT 挿入などの処理を含んでおり、生成される回路がゲートレベルのパイプライン化や

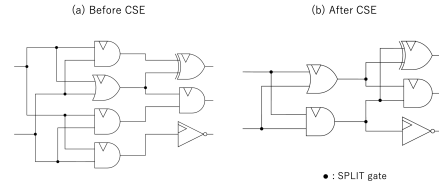


図 6 CSE の例

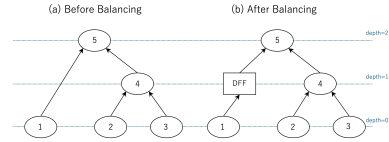


図 7 DFF 挿入の例

ファンアウト制限といった SFQ 特有の制約を見だし、正しく動作することを保証する。

DFF 挿入

DFF 挿入パスでは、グラフ全体でパイプラインの深さを均等にするために DFF を挿入する。チップ面積を削減するためには、挿入する DFF の総数を最小限にする必要がある。そこで、提案手法では以下のように DFF 挿入を行う。はじめに、グラフをトポロジカルソートして、ノードをボトムアップに走査する。すべてのエッジについて、ノード間の $depth$ の差を調べ、これが 1 より大きい場合にはタイミングがずれているため、追加で DFF を挿入する必要がある。必要な DFF の数を $DFD[i, j]$ とすると、これは

$$DFD[i, j] = depth[j] - depth[i] - 1, \quad (3)$$

で与えられる。ここで、 $depth[i]$ と $depth[j]$ はそれぞれ、ノード i, j の $depth$ を表す。図は DFF 挿入の例を示している。

グラフの走査中に各ノードとエッジは一度だけ処理されるため、このアルゴリズムの時間計算量は $O(|V| + |E|)$ となる (V, E はそれぞれノードとエッジの集合)。この手法により、グラフ全体で挿入される DFF の総数は最小化される。

SPLIT 挿入

SPLIT 挿入パスでは、SFQ 回路における単一ファンアウト制約を満たすために SPLIT ゲートが挿入される。あるノードが複数の consumer を持つ場合、複数のファンアウトをもつ。直接的なファンアウトを 1 つに制限するために、バランスド SPLIT ツリーが構築され、リーフの SPLIT ノードがそれぞれ 1 つの consumer に接続される。図は SPLIT 挿入の例を示している。このパスによる処理のあと、グラフは SFQ のファンアウト制約を満たす。

3.3 コード生成

すべてのグラフパス処理が完了したら、IR は HDL に変

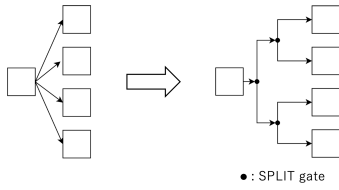


図 8 SPLIT 挿入の例

コード 1 DSLX で記述された HA および FA

```
fn halfadder(a: u1, b: u1) -> (u1, u1) {
  (a & b, a ^ b)
}

fn fulladder(a: u1, b: u1, cin: u1) -> (u1, u1) {
  let t1 = halfadder(a, b);
  let (cout1, sum1) = t1;

  let t2 = halfadder(sum1, cin);
  let cout2 = t2.0;
  let sum2 = t2.1;

  (cout1 | cout2, sum2)
}
```

換される。本研究では、ターゲット HDL として RustSFQ を採用する。リストは合成の例を示している。リストに示されている DSLX による半加算器 (HA) と全加算器 (FA) が最終的にリストに示されているような RustSFQ での記述に変形される。

4. 評価

本節では、提案手法の評価を行う。ケーススタディとして、バイナリニューラルネットワーク (BNN) における基本的な演算カーネルとして機能する 16 ビットの XNOR-popcount 回路を生成し、提案手法の有効性を実証する。

4.1 動機

BNN [15] は、重みと活性化値をバイナリ値 (通常は +1 と -1) に制限することで、従来のディープニューラルネットワーク (DNN) に比べて計算量を大幅に削減する。DNN は浮動小数点演算による積和 (MAC) 演算を多用し、GPU などの高負荷なハードウェア上で実行される。これに対して、BNN は +1 を論理値の 1, -1 を論理値の 0 にマッピングすることで、乗算を XNOR 演算、加算を population count (popcount)、とそれぞれ単純なビット演算に置き換える。すなわち、BNN 層の核となる計算は次の式のように表される。

$$a_1 = \text{popcount}(\text{xnor}(a_0, w_0)). \quad (4)$$

ここで、 a_0 と w_0 はそれぞれ入力となる活性化値と重みで、 a_1 は積和演算の結果を表している。

このアプローチは、エネルギー効率の観点から SFQ 論理に特に適している。2.1 節で説明したように、SFQ 回路は量子化されたパルスによって情報を表現し、ジョセフソン接合をスイッチングしてパルスを伝搬させる。そのため、ゲート数を削減することはエネルギー消費量の低減にそのまま貢献する。従来の DNN 実装はエネルギー消費の大きい浮動小数点乗算器に依存しているが、BNN は乗算器を完全に取り除き、単純な XNOR ゲートとビットカウント演算に置き換えている。これらの演算により、ゲート数と回路の複雑さが大幅に削減され、エネルギー効率が向上する。実装の観点から見ると、XNOR-popcount 回路は単純なビット演算と加算で構成されており、どちらも組み合わせ回路として実装可能である。さらに、この演算はゲートレベルでのパイプライン処理やファンアウト制限などの制約を満たすように適切に設計する必要がある。そのため、本研究では、この演算を提案する合成ツールの有効性を評価するためのベンチマークとして採用する。このケーススタディでは、16 ビットの XNOR-count 回路にフォーカスする。このサイズは現実的な設計を示すのに十分な大きさでありながら、詳細な解析を行う上でも扱いやすいサイズである。

4.2 実験方法

提案ツールは、公式のオープンソースレポジトリから入手した XLS [14] をベースに C++ で実装された。ビルドおよびバイナリ実行はすべて、AMD EPYC プロセッサ (合計 128 個の CPU コア) を搭載した Ubuntu 22.04.05 LTS 上で行われた。入力となる 16 ビット XNOR-count 回路の記述はリストに示されるように DSLX で記述されている。この DSLX による回路記述は、XLS に組み込まれているツールを用いてまず XLS IR (XLS がもつ IR) に変換され、最適化された。次に、この IR はビット単位の演算に分解され、再度最適化されて、提案ツールへの最終的な入力形式に変換された。

生成された回路の機能としての正しさは、RustSFQ の論理シミュレーションを用いて検証した。入力となる重みと活性化値はランダムに生成されたものを用いた。

4.3 結果

生成された XNOR-count 回路は 24 段のパイプラインで構成されており、これはこの回路のレイテンシが 24 クロックサイクルであることを示している。シミュレーションは Icarus Verilog を用いて行われ、結果は GTKWave を用いて可視化された。図はシミュレーション結果の一部を切り取ったものである。シミュレーションでは 28486 個のテストケース全てが正しく計算されており、この自動合成手法

コード 2 RustSFQ で出力された HA および FA

```
fn __fulladder_halfadder() -> Circuit<3, 0, 2, 0> {
    let inputs = ["a", "b", "clk"];
    let inputs_counter = [];
    let outputs = ["q1", "q0"];
    let outputs_counter = [];
    let (mut circuit, [a, b, clk], [], [q1, q0], []) =
        Circuit::create(inputs, inputs_counter, outputs, outputs_counter, "__fulladder_halfadder");

    let (split_6_0, split_6_1) = circuit.split(a % 0);
    let (split_7_0, split_7_1) = circuit.split(b % 0);
    let (split_5_0, split_5_1) = circuit.split(clk % 0);
    let and_2 = circuit.and(split_6_0 % 1, split_7_0 % 1, split_5_0 % 0);
    let xor_3 = circuit.xor(split_6_1 % 1, split_7_1 % 1, split_5_1 % 0);

    circuit.unify(and_2, q1);
    circuit.unify(xor_3, q0);

    return circuit;
}

fn __fulladder_fulladder(__fulladder_halfadder: &Circuit<3, 0, 2, 0>) -> Circuit<4, 0, 2, 0> {
    let inputs = ["a", "b", "cin", "clk"];
    let inputs_counter = [];
    let outputs = ["q1", "q0"];
    let outputs_counter = [];
    let (mut circuit, [a, b, cin, clk], [], [q1, q0], []) =
        Circuit::create(inputs, inputs_counter, outputs, outputs_counter, "__fulladder_fulladder");

    let (split_16_0, split_16_1) = circuit.split(clk % 0);
    let (split_18_0, split_18_1) = circuit.split(split_16_0 % 0);
    let (split_17_0, split_17_1) = circuit.split(split_16_1 % 0);
    let (split_20_0, split_20_1) = circuit.split(split_17_0 % 0);
    let (split_19_0, split_19_1) = circuit.split(split_17_1 % 0);
    let ([cout1, sum1], []) = circuit.subcircuit(__fulladder_halfadder, [a % 0, b % 0, split_20_0 % 0], []);
    let dff_12 = circuit.dff(cin % 1, split_20_1 % 0);
    let dff_13 = circuit.dff(cout1 % 1, split_19_0 % 0);
    let ([cout2, sum2], []) = circuit.subcircuit(__fulladder_halfadder, [sum1 % 0, dff_12 % 0, split_19_1 % 0], []);
    let or_9 = circuit.or(dff_13 % 1, cout2 % 1, split_18_0 % 0);
    let dff_14 = circuit.dff(sum2 % 1, split_18_1 % 0);

    circuit.unify(or_9, q1);
    circuit.unify(dff_14, q0);

    return circuit;
}
```

の妥当性が実証された。

表 1 は 3.2 節で説明した最適化フェーズにおける、最適化ありとなしのゲート数の比較を示している。最適化の結果、AND ゲートと SPLIT ゲートの数が減少し、合計ゲート数は 945 から 921 へと、2.5%減少した。最適化パスによりゲート数は幾分か減少しているが、その割合は比較的小

さい。

コード削減を含む自動合成ツールの有効性を評価するために、コード行数 (LOC) を `wc -l` コマンドで計測した。表 2 は DSLX で記述された高レベルの XNOR-count 回路記述と、提案ツールが生成した、RustSFQ で記述されたゲートレベルでの厳密な SFQ 回路記述の両方についての

コード 3 16-bit XNOR-count description in DSLX

```
// 16bit popcount
fn popcount16(x: u16) -> u5 {
    (x[0:1] as u5) + (x[1:2] as u5) + (x[2:3] as u5) +
    (x[3:4] as u5) +
    (x[4:5] as u5) + (x[5:6] as u5) + (x[6:7] as u5) +
    (x[7:8] as u5) +
    (x[8:9] as u5) + (x[9:10] as u5) + (x[10:11] as u5)
    + (x[11:12] as u5) +
    (x[12:13] as u5) + (x[13:14] as u5) + (x[14:15] as
    u5) + (x[15:16] as u5)
}

// 16 bit BNN neuron
fn bnn_neuron_16(x: u16, w: u16) -> u1 {
    // XNOR
    let xnor = !(x ^ w);

    // popcount
    let count = popcount16(xnor);

    // threshold (test: >= 8)
    count >= u5:8
}
```

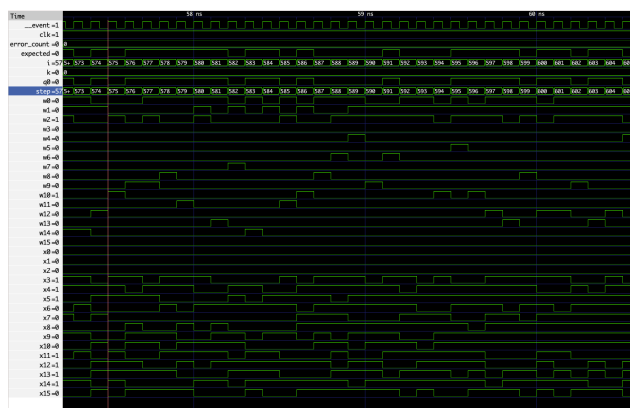


図 9 シミュレーション結果を GTKWave を用いて可視化した様子

表 1 最適化の有無によるゲート数の比較

#ゲート	最適化なし	最適化あり
AND	101	93
DFF	166	166
NOT	60	60
OR	42	42
SPLIT	544	528
XOR	32	32
Total	945	921

LOC を示している． RustSFQ による記述では 935 LOC なのに対して， DSLX による高レベルでの記述ではわずか 30 LOC であり， 30 倍以上のコード削減を達成している． これは提案手法が回路設計の生産性向上に大きく貢献して

表 2 回路記述レベルによる LOC の比較

Circuit Description	LOC
High-level design in DSLX	30
Synthesized design in RustSFQ	935

いることを示している．

5. 結論

本研究では，既存の高位合成ツールを拡張して，効率的な SFQ 回路設計に特化した，新しい自動パイプライン合成ツールを提案した． このツールは，高レベルの機能的な記述をゲートレベルの SFQ 回路記述に変換すると同時に， DFF 挿入によるパスのバランス調整や SPLIT 挿入によるファンアウト制限の解消など， SFQ 固有の制約を満たす回路構成を自動で合成する． バイナリニューラルネットワーク用の XNOR-popcount 回路の自動合成による評価から，合成された回路の正しさと，最適化を通じた 2.5% のゲート数削減および 30× のコード削減が実証された．

謝辞

本研究の一部は， JSPS 科研費 23H00467 および JSPS 科研費 26K21734 の支援による．

参考文献

- [1] K. Nakajima, Y. Onodera, and Y. Ogawa. Logic design of Josephson network. *Journal of Applied Physics*, 47(4):1620–1627, 04 1976.
- [2] K.K. Likharev and V.K. Semenov. RSFQ logic/memory family: a new Josephson-junction technology for sub-terahertz-clock-frequency digital systems. *IEEE Transactions on Applied Superconductivity*, 1(1):3–28, 1991.
- [3] D. E. Kirichenko, S. Sarwana, and A. F. Kirichenko. Zero Static Power Dissipation Biasing of RSFQ Circuits. *IEEE Transactions on Applied Superconductivity*, 21(3):776–779, 2011.
- [4] Feng Li, Yuto Takeshita, Daiki Hasegawa, Masamitsu Tanaka, Taro Yamashita, and Akira Fujimaki. Low-power high-speed half-flux-quantum circuits driven by low bias voltages. *Superconductor Science and Technology*, 34(2):025013, jan 2021.
- [5] W. Chen, A.V. Rylyakov, V. Patel, J.E. Lukens, and K.K. Likharev. Rapid single flux quantum T-flip flop operating up to 770 GHz. *IEEE Transactions on Applied Superconductivity*, 9(2):3212–3215, 1999.
- [6] Lieze Schindler, Johannes A. Delpert, and Coenrad J. Fourie. The ColdFlux RSFQ Cell Library for MIT-LL SFQ5ee Fabrication Process. *IEEE Transactions on Applied Superconductivity*, 32(2):1–7, 2022.
- [7] Johannes Arnoldus Delpert, Kyle Jackman, Paul le Roux, and Coenrad Johann Fourie. JoSIM—Superconductor SPICE Simulator. *IEEE Transactions on Applied Superconductivity*, 29(5):1–5, 2019.
- [8] Mebuki Oishi, Sun Tanaka, and Shinya Takamaeda-Yamazaki. RustSFQ: A Rust-based Domain-Specific Language for Efficient SFQ Circuit Design. *IEEE Micro*, pages 1–9, 2026.
- [9] Ghasem Pasandi, Alireza Shafaei, and Massoud Pedram.

- SFQmap: A Technology Mapping Tool for Single Flux Quantum Logic Circuits. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.
- [10] Ghasem Pasandi and Massoud Pedram. qSeq: Full Algorithmic and Tool Support for Synthesizing Sequential Circuits in Superconducting SFQ Technology. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 133–138, 2021.
- [11] Junyao Zhang, Paul Bogdan, and Shahin Nazarian. A Majority Logic Synthesis Framework For Single Flux Quantum Circuits, 2023.
- [12] Soheil Nazar Shahsavani and Massoud Pedram. A Minimum-Skew Clock Tree Synthesis Algorithm for Single Flux Quantum Logic Circuits. *IEEE Transactions on Applied Superconductivity*, 29(8):1–13, 2019.
- [13] Ching-Cheng Wang and Wai-Kei Mak. A novel clock tree aware placement methodology for single flux quantum (sfq) logic circuits. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.
- [14] Google and Contributors. XLS: Accelerated HW Synthesis. <https://github.com/google/xls>, 2020. Accessed: 2026-03-30.
- [15] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.