A Buffer and Splitter Insertion Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits

Ruizhe Cai*, Olivia Chen†, Ao Ren*, Ning Liu*, Nobuyuki Yoshikawa†, Yanzhi Wang*
*Department of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115, USA
{cai.ruiz, ren.ao, liu.ning}@husky.neu.edu, yanz.wang@northeastern.edu
† Institute of Advanced Sciences
Yokohama National University
Yokohama, Kanagawa 2408501, Japan
{chen-olivia-pg, nyoshi}@ynu.ac.jp

***Abstract*—Adiabatic Quantum-Flux-Parametron (AQFP) logic is an adiabatic superconductor logic that has been proposed as alternative to CMOS logic with extremely high energy efficiency. In AQFP technology, gates are driven by AC-power, which also serves as clock signal to synchronize the outputs of all gates in the same clock phase. As a matter of fact, AQFP circuits may require huge amount of buffers and splitters to be inserted to allow inputs to any gate having equal delay. Existing buffer and splitter insertion method does not deliver optimization, which could lead to huge space and delay overhead. A better automated buffer and splitter framework is imminent for more efficient AQFP circuits design. In this paper, we propose an automated buffer and splitter insertion method that is capable of adding optimized amount of buffers and splitters to any given gate-level netlist to achieve equal delay for all gates. The proposed method achieve equal delay by inserting buffers and splitters with any library limitation on the size of splitters. Experimental results suggest that the proposed method can deliver better results compared with the existing method, with up-to 40.84% less in size and 3.13% less in delay when splitter fan-out size is limited to four.**

I. Introduction

Being widely-known for low energy dissipation and ultra-fast switching speed, Josephson Junction (JJ) based superconductor logic families have been proposed and implemented to process analog and digital signals [1]. It can operate at clock frequencies of several tens of gigahertz and even hundreds of thousands of times as energy efficient as its CMOS counterparts, thanks to its construction of resistance-less wires and ultra-fast switches. It has been perceived to be an important candidate to replace state-of-the-art CMOS due to the superior potential in operation speed and energy efficiency, as recognized by the U.S. IARPA C3 and SuperTools Programs and Japan MEXT-JSPS Project.

The design and fabrication of superconducting circuits have already been established [2], [3], [4]. In addition, a prototype superconducting microprocessor "Core 1" has been demonstrated in 2004 [3], which is able to execute instructions at a high clock frequency of several tens of gigahertz, and with extremely low-power dissipation. These achievements make superconducting electronics highly promising for future high-performance computing applications.

As one of the most matured superconducting technology, the Rapid-Single-Flux-Quantum (RSFQ) technology is proposed by K. Likharev, O. Mukhanoc, V. Semenov in 1985 [1]. Despite its capability to be operated at an ultra-high speed of hundreds of GHz while maintaining extremely low switching energy (10^{-19}J), it suffers from an increasing static power due to on-chip resistors that are required for constant DC bias supply for the main RSFQ circuit. Numerous methods have been proposed to resolve the static power dissipation problem of RSFQ, including low-voltage RSFQ (LV-RSFQ) [5], reciprocal quantum logic (RQL) [6], LR-biased RSFQ [7] and energy-efficient single-flux quantum (eSFQ) [8].

The *Adiabatic Quantum-Flux-Parametron* (AQFP) technology, on the other hand, uses AC bias/excitation currents as both multi-phase clock signal and power supply [9] to mitigate the power consumption overhead of DC bias while operating at a frequency of few GHz. Consequently, AQFP is remarkably energy efficient compared to RSFQ, albeit operating at a lower frequency. The energy-delay-product (EDP) of the AQFP circuits fabricated using processes such as the AIST standard process 2 (STP2) and the MIT-LL SFQ process [10], [11], is at least 200 times smaller than those of the other energy-efficient superconductor logics and is only three orders of magnitude larger than the quantum limit [9]. Physical testing results of an AQFP 8-bit carry-look-ahead adder and large scale circuits consisting up-to 10,000 AQFP logic gates have demonstrated the AQFP being a promising technology that is robust against circuit parameter variations [12].

Despite the high application potential of AQFP in VLSI circuits, a systematic, automatic synthesis framework for AQFP is imminent. There are two features of AQFP that restrict conventional CMOS synthesis methods being directly applied on AQFP. In spite of And-Or-Inverter(AOI) based

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00067

representation, which conventional CMOS circuits highly relies on, AQFP circuits prefer majority gates. In fact, its two inputs AND and OR gates are also built with three inputs majority gate with one input being constant. In addition, given its clock-synchronized data propagation nature, AQFP technology requires all inputs to any gate having equal delay. In order to meet this balanced timing requirement, splitters and buffers need to be inserted to the circuit. As a matter of fact, some circuit size can be doubled even with very little amount of buffers and splitters inserted. The buffer and splitter insertion method can have a huge impact on the overall resource consumption. As the design complexity increases, an unoptimized buffer and splitter insertion method could result in huge amount of unnecessary buffers and splitters added. While there are methods exploring majority synthesis results for majority logic based technologies[13], [14], [15], [16], [17], an automated buffer and splitter insertion framework for AQFP technology that seeks for an optimized solution is still imminent, as buffer and splitter insertion can be very challenging, especially for complex designs.

In this paper, we propose an automated buffer and splitter insertion framework that can achieve very promising results. The proposed method can find an optimized amount of buffers and splitters to be inserted to achieve balanced paths under any splitter fan-out capability restrictions. Experimental results on multiple logic designs suggest that the proposed method can deliver very promising results, as it delivers up-to 40.84% less in size and 3.13% less in delay with splitter fan-out size limited to four, compared to the existing insertion method.

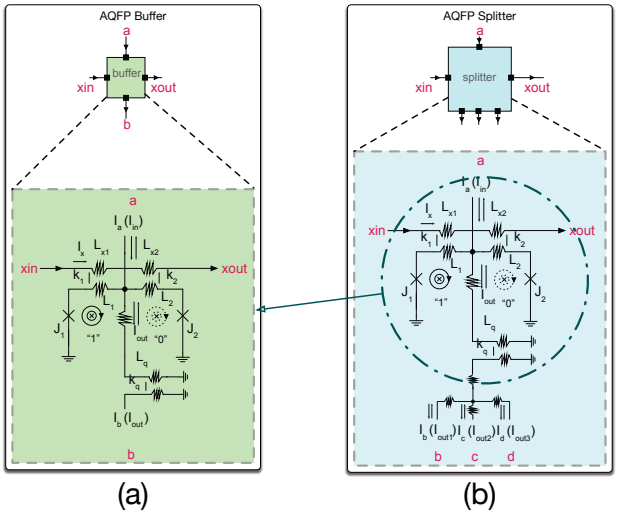

Fig. 1. (a) AQFP buffer junction level schematic. (b) AQFP splitter junction level schematic.

II. Background

A. AQFP Superconducting Logic

Fig. 1 (a) shows the basic logic structure of an AQFP buffer consisting of a double-Josephson-Junction SQUID [18]. As mentioned above, an AQFP logic gate is mainly driven by AC-power, which serves as both excitation current and power supply. The excitation current I_x is typically in the order of hundreds of micro-amperes. Excitation fluxes are applied to the superconducting loops via inductors L_1, L_2, L_{x1} and L_{x2}. Either the left or the right loop stores one single flux quantum based on the value of the small input current I_{in}. The storage position of the quantum flux in the left or the right loop can be encoded as logic '1' or '0' as the directions of current on the output are opposite. Consequently, by viewing the direction of the output current I_{out} as the logic state, The device is capable of acting as a buffer.

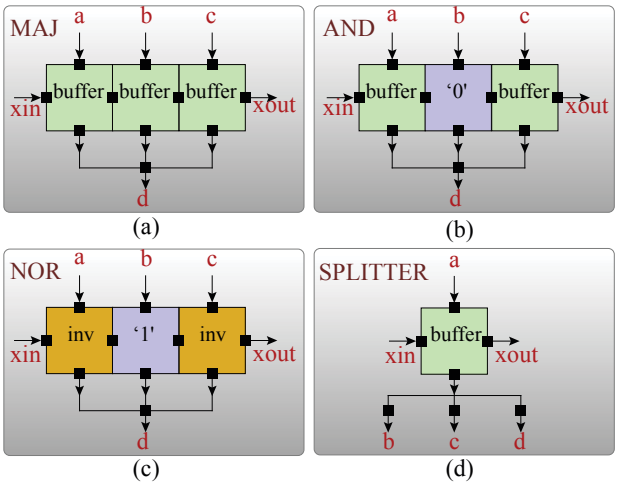

Fig. 2. (a). An AQFP majority gate, which consists of 3 buffers; (b). An AQFP AND gate, which consists of 2 buffers and a constant '0' gate; (c). An AQFP NOR gate, which consists of 2 inverters and a constant '1' gate; (d). An AQFP 1-to-3 splitter, which consists of a buffer.

Both the AQFP inverter and constant cells are designed base on the AQFP buffer. By negating the coupling coefficient of the output transformer in the AQFP buffer, it behaves as an inverter. The constant cell in AQFP is designed using asymmetry excitation flux inductance in the AQFP buffer. The AQFP splitter is also implemented based on the AQFP buffer as shown in Fig. 1 (b). Unlike CMOS gates, which are connected to fan-out gates directly, all AQFP gates need to be connected to splitters for fan-out more than one.

The AQFP standard cell library is built via the *minimalist design approach* [12] by designing more complicated gates using a bottom-up manner. As shown in Fig. 2 (a), the very basic multi-input logic gate in AQFP is the majority gate. The output, 'd' is decided by number of '1's at inputs, 'a' to 'c'. A variation of the majority gate (such as the minority gate) can be produced by replacing any buffer with inverter. AND/OR gates in AQFP are designed from majority-based gate by replacing one of the buffers or inverters with a constant gate. As shown in Fig. 2, (b)-(c), an AQFP AND gate is implemented by two buffers and one constant '0' while an AQFP NOR gate is implemented by two inverters and one constant '1'. As three-input majority-based gates and two-input AND/OR based gates have the same area and delay,

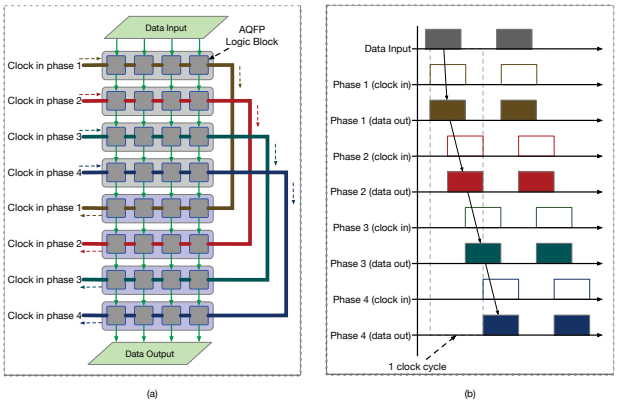

Fig. 3. (a). 4-phase clocking scheme for AQFP circuits; (b). Data propagation between neighbouring clock phases.

the former kind is more preferred in AQFP technology logic design.

B. AQFP Data Synchronization

The AC-power in AQFP logic does not only drive AQFP logic gates, but also serves as clock signal to synchronize the outputs of all gates in the same clock phase. Therefore, data propagation in AQFP circuits requires overlapping of clock signals from neighboring phases. Fig. 3 presents an example of typical clocking scheme of AQFP circuits and data flow between neighboring clock phases. Such clock-based synchronization characteristic also requires that all inputs for each gate should have the same delay (clock phases) from the primary inputs. Consequently, splitters and buffers are needed in AQFP circuits for proper data propagation.

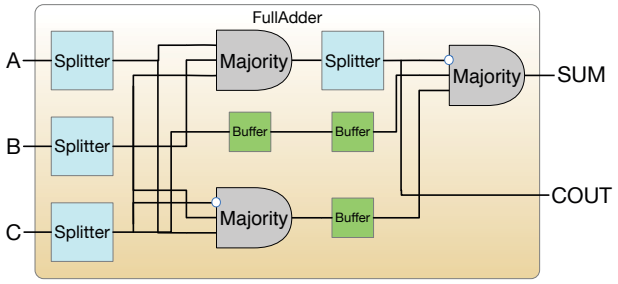

Fig. 4. Full adder implementation in AQFP

An AQFP circuit may contain a huge portion of splitters and buffers. This is partially due to the fact that the fan-out capability of splitters is limited by the standard library, which is usually 3 or 4. Therefore, a fan-out with many outputs can results in a splitter tree, which may also introduce extra delay. As shown in Fig. 4, a full adder implementation in AQFP has nearly half of its resource occupied by buffers and splitters, which is also the optimum amount inserted for this simple logic. As the complexity of circuit increases, buffer and splitter insertion can be very challenging.

C. Logic Synthesis for AQFP

An AQFP design flow has been proposed in [13]. This design flow uses the open-source yosys [19] for logic synthesis with AQFP standard cell library. For buffer and splitter insertion, it uses a post-synthesis script to add buffers and splitters on top of the netlist derived by yosys. However, the buffer and splitter insertion method implemented is a straightforward method with no optimization, which could lead to unnecessary introduced extra space and delay.

In this paper, we would like to propose a new buffer and splitter method with optimization. Similar to the previously proposed method, the method proposed in this paper is to be executed on top of the netlist synthesised with AQFP standard cell library using the open-source yosys[19] logic synthesis tool.

III. Buffer and Splitter Insertion

A. Overview

As discussed in section II-B, all inputs connected to a given gate must have the same data arrival time, therefore, all fan-outs must be connected from physical splitters for each branch to propagate individually. The key tasks for this framework are inserting splitters for fan-outs and adding buffers for data synchronization, which can be implemented in an intuitive approach by firstly inserting splitters/splitter trees at all fan-out locations then adding buffers to level all uneven data arrivals. However, this does not guarantee good solution. As the maximum fan-out capability of the splitters is limited by the library, large fan-outs can not be directly replaced with splitters. Simply breaking down large fan-outs into splitter-trees can easily introduce extra delay, which could require more buffers to be inserted at later stage. Therefore, it is important to implement a more effective method to break-down large fan-outs to introduce the minimum extra delay before adding buffers to fill the void of other uneven paths.

The proposed method is capable of achieving balanced delay by inserting the minimum amount of buffers and splitters to any given gate-level netlist. It is based on three key observations:

- The system delay can be determined once all splitters are fixed. While buffers are only used to fill the void for uneven paths, splitters are inserted to handle fan-outs and consequently introduce new unbalanced path on top of the raw netlist. Therefore, after splitters insertion, the netlist timing structure is locked, awaiting buffers to be filled to the unbalanced paths. As mentioned above, an effective large fan-outs breakdown strategy is essential to guarantee minimum extra delay to be introduced, and consequently good solution.
- As discussed in section II-B, a splitter, regardless of its fan-out capability, shares the same physical resources as a splitter. Therefore, nested splitters should be always preferred over a single splitter parenting buffers. For example, as illustrated in Fig. 5, a four-output splitter with two of its outputs each connecting to a buffer should be

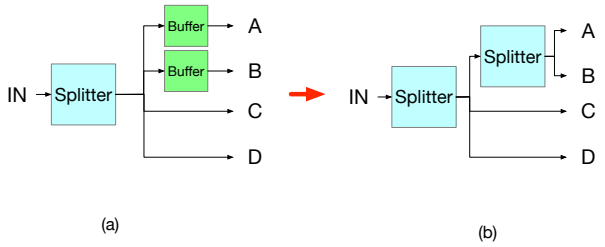

Fig. 5. (a) A four-output splitter with two of its output connecting to buffer. (b) The same structure can be implemented using a three-output splitter chaining a two-output splitter, which can reduce total resource consumption by 33%.

optimized as a three-output splitter nesting a two-output splitter, which saves the resource consumption by 33%.

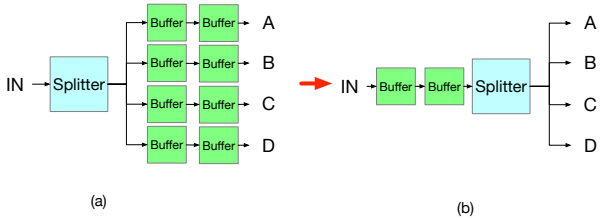

Fig. 6. (a) A four-output splitter with all of its output connecting to two buffers. (b) The same structure can be implemented using a four-output splitter with two buffers connected after its input, which can reduce total resource consumption by 66%.

- If all outputs of a splitter require further delay to be introduced, the shared amount of delay should be introduced before the splitter for less overall hardware footprint. As shown in Fig. 6, a four-output splitter is connected with two buffers after each of its inputs, an alternative structure that can achieve the same delay model while utilizing less hardware resources would be two buffers connecting to the input of the four-output splitter. The latter implementation can reduce the total gate count by 66%.

Based on the above three observations, this paper proposes an effective method for buffer and splitter insertion for AQFO circuits. For each net connecting fan-out, it contains three main steps: Firstly, a virtual splitter with arbitrary fan-out capabilities are inserted to the netlist to replace the fan-out; Each virtual splitter are then separated to smaller virtual splitters based on delay required at each outputs, forming a virtual splitter chain; Next, all virtual splitters on the virtual splitter chain are mapped to splitters. Finally, after all fan-outs are fixed with splitters, buffers are added to the remaining uneven paths.

B. Virtual Splitter Insertion

In the first step, a virtual splitter is inserted to cover the fan-out net. A virtual splitter is a splitter with arbitrary output capability, and its delay is assumed to be the same as a buffer. With all fan-outs are replaced with virtual splitters, one can estimate the maxim delay and hardware resource cost under ideal condition, in where all virtual splitters can be separated to splitters without introducing extra delay. After a virtual splitter is placed, it is possible to calculate buffers needed at each of its fan-outs, so that all fan-outs can reach to corresponding following logic gates synchronized.

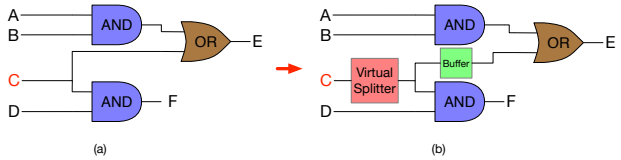

Fig. 7. (a) Sample logic before virtual buffer insertion. (b) Virtual splitter is inserted at C for fan-out.

Please note that this step does not determine the final amount of buffers to be added after each fan-outs, but provides an easier structure to be optimized. As shown in Fig. 7, net C is connected with virtual splitters connected. Based on the second observation mentioned above, it is possible to breakdown the root virtual splitter by introducing minimum amount of extra delay.

Each virtual splitter holds the following information: the fan-out nets it's hosting, amounts of buffers needed after each of its fan-outs and its input, and its input net. For better algorithm illustration, $VS.fanouts$ is used when referring to the fan-out nets hosted by virtual splitter connected VS, the input net to the fanout is denoted as $VS.in$, the amount of buffers to be inserted after net n_i is denoted as $n\_buf(n_i)$, and the amount of buffers needed and $n\_buf_{min}(Nets)$ is used to refer to the minimum value of the amounts of buffers to be connected after nets in $Nets$.

C. Virtual Splitter Separation and Generation

This step transforms a single virtual splitter to a virtual splitter chain. As mentioned in the second observation, nested splitters should be preferred over splitters with buffers connected after, therefore, when more than one outputs are connected to buffers of which the amount is greater than the rest, these outputs with more buffers connected should be grouped under a new virtual splitter nested under the parent virtual splitter. In other words, the target virtual splitter keeps outputs that have the minimum amount of buffers connected, and generates a new virtual splitter to host the rest outputs with each of their connecting buffers reduced by one. This step does not introduce extra delay as only new virtual splitters are generated to replace vectors of buffers. Instead, it alters the distributivity of fan-outs to minimize buffer utilization. Same step is applied on the newly generated virtual splitters until all fan-outs needs the same delay or only one of the branch requires more buffers than the rest.

As illustrated in algorithm 1, the same function is applied recursively to the newly generated virtual splitters to create a virtual splitter chain structure. Fan-out nets of each virtual splitter in the chain is either connected to the virtual splitter in the next level or buffers. As presented in Fig. 8, the root virtual splitter has 9 outputs, with 3 of them requiring 2 buffers to be

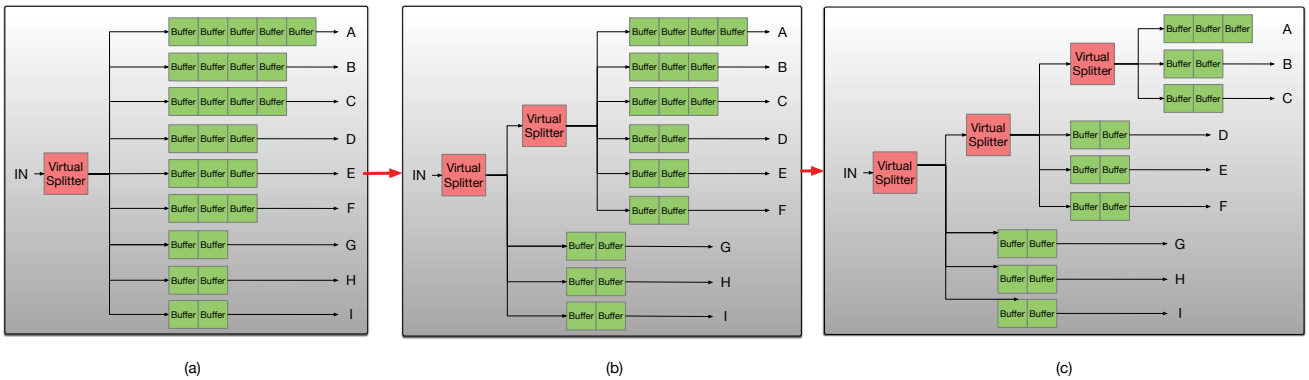

Fig. 8. Example of virtual splitter separation and generation. (a) Initial virtual splitter with 9 outputs with each having buffers connected. (b) Separation and generation performed at the root virtual splitter, resulting the 6 longest paths sharing a new virtual splitter. (c) The virtual splitter at the second level is used to generate the leaf virtual splitter, with the top 3 longest paths to host.

Algorithm 1: Virtual Splitter Generation Algorithm

input : target virtual splitter VS
output: virtual splitter VS_{leaf}
$VS.fanouts$ is the set of fan-outs nodes hosted by VS;
G_{next} is an empty set;
$n\_buf_{min}(Nets)$ is the minimum value of the amounts of buffers to be connected after nets in $Nets$;
foreach net n_i in $VS.fanouts$ **do**
  if $n\_buf(n_i) > n\_buf_{min}(VS.fanouts)$ **then**
    add n_i to G_{next};
  end
end
if $lenth(G_{next}) > 1$ *and* $length(VS.fanouts) - length(G_{next}) > 1$ **then**
  Create new net n_{new} and virtual splitter VS_{new};
  $VS_{new}.in = n_{new}$;
  Add n_{new} to $VS.fanouts$;
  foreach net n_i in G_next **do**
    Remove n_i from $VS.fanouts$;
    Add n_i to $VS_{new}.fanouts$;
    $n\_buf(n_i) = n\_buf(n_i) - 1$
  end
  return VirtualSplitterGeneration(VS_{new})
end
return VS

connected after, 3 of them requiring 3 buffers to be connected after, 2 of them requiring 4 buffers to be connected after and the remaining one requiring 5 buffers to be connected after. The longest six fan-out nets are grouped and formed into a new virtual splitter under the root virtual splitter. Each of its relative delay required is reduced by one. And finally, the longest three fan-outs nets are picked into the last newly generated virtual splitter. After this step, the virtual splitter created in the first step are now converted to a virtual splitter chain structure, and for each virtual splitters, all or all but one hosted fan-out nets that are not connected to a virtual splitter have the same amount of buffers connected after. This structure can be easily optimized further based on the third observation.

D. Virtual Splitter Mapping

After a virtual splitter is turned into a virtual splitter chain based on the numbers of buffers to be inserted after each its fan-out nets, it can be mapped to splitters with fan-out capability restriction given by the library. In addition, based on the third observation, any virtual splitters can be further optimized by removing the common amount of buffers at the fan-out side and adding them after the input-net to the splitters. The buffers relocated to the input side of a virtual splitter in the middle of a virtual splitter chain can then be considered as the buffers after one of the fan-out net of the parent virtual splitter, allowing further optimization to be applied on the parent virtual splitter. Therefore, this step applies the same operations on all virtual splitters on a virtual splitter chain, from the leaf one to the root one.

As presented in algorithm 2, the virtual splitter mapping algorithm is recursively applied on the virtual splitters on the virtual splitter chain created in the previous step. It starts from the leaf virtual splitter of the virtual splitter chain. First, if the virtual splitter can be directly mapped to a library splitter, it can be directly mapped to a library splitter. If the target virtual splitter has more outputs than the library limitation, it is firstly separated to the minimum amount of splitters given by the library. For a mapped splitter, if all of its outputs are connected with buffers, the common minimum amount of buffers is factored to the input side. Inputs of these splitters are then added to the parent virtual splitter. For the root virtual splitter, a new root virtual splitter is generated to host the mapped splitters if it cannot be directly mapped. If its input is connected with buffers, the amount of buffers associated is reduced by one. This step will only introduce one unit of extra delay if the root virtual splitter cannot be mapped directly and its input does not have any buffers connected, initially or factored from its fan-outs. Consequently, this method can alway guarantee minimum extra delay to be introduced to the given virtual splitter chain.

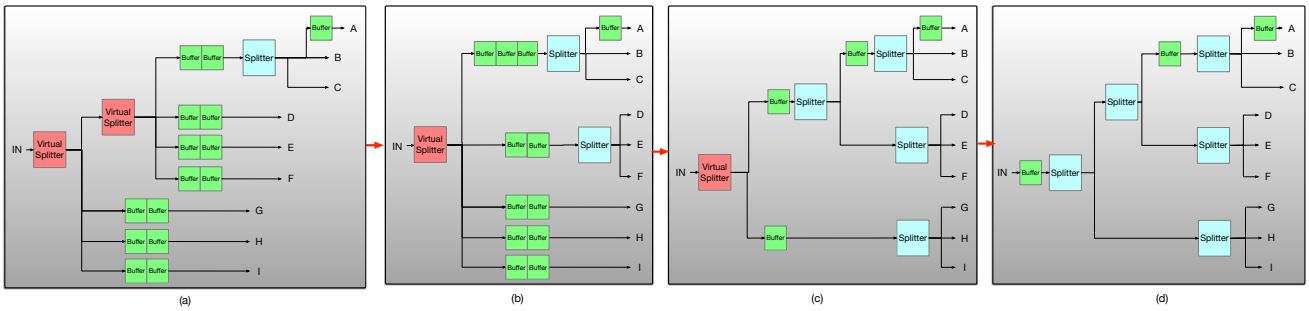

Fig. 9. Example of virtual splitter mapping. (a) Virtual splitter mapping performed at the leaf virtual splitter, which is directly mapped to a three-output splitter. Mutual amount of buffers (2) is lifted to the input side of the splitter. (b) The virtual splitter at the second level is mapped to a three-output splitter, and a buffer, mutual amount of buffers (2) among its fan-outs is added to the input side. (c). The root virtual splitter is mapped to a three-output splitter and two-output splitter. Since it has no parent virtual splitter, a new virtual splitter is created, and the buffer to be added is reduced by one. (d) The lastly created virtual splitter is mapped to a two-output virtual splitter. Mutual amount of buffer (1) is factored to the inputs side.

Fig. 9 presents an example of the virtual splitter mapping algorithm operating on a three level virtual splitter chain created in the previous step shown in Fig. 8 (c). The maximum fan-out capability given by the library is three. The leaf virtual splitter with three fan-outs can be directly mapped to library splitter. As three fan-outs are connected to 2, 2, 3 buffers respectively, 2 buffers can be factored to the input side and propagated to the parent level. The virtual splitter in the middle, which now have four fan-outs, is mapped to a three fan-out splitter and a buffer. The amount of buffers to be mapped to the parent level is four. Finally, the root virtual splitter is mapped to a three fan-out and a two fan-out splitter, and a new two fan-out splitter is created as the root splitter.

E. Summary

The first three steps determines the splitter tree layout for each fan-out nets as well as the amount of buffers to be inserted after each fan-out nets. Buffers can then be added accordingly. The primary outputs of a module can be considered as inputs to a virtual gate to be processed in the same manner as the rest of the wires. Register inputs and outputs are treated as primary outputs and primary inputs, respectively.

The above steps seek a splitter tree structure for a given fan-out net for minimum extra delay introduction. However, it still may affect buffers needed at other paths. Therefore, virtual splitter insertion, virtual splitter generation and mapping are applied to fan-out nets in the delay ascending order dynamically. In each iteration, the fan-out net closest to the primary inputs is picked to be processed, so that any extra delay it introduces can be reflected accordingly at paths that are affected.

The overall algorithm is presented in algorithm 3. In each step, it picks the fan-out net closest to the primary inputs to insert virtual splitter. The newly inserted virtual splitter is then processed using the separation and generation algorithm to form a virtual splitter chain, before being mapped to library splitters. After all fan-out nets are processed, buffers can be inserted for the other uneven paths and primary outputs.

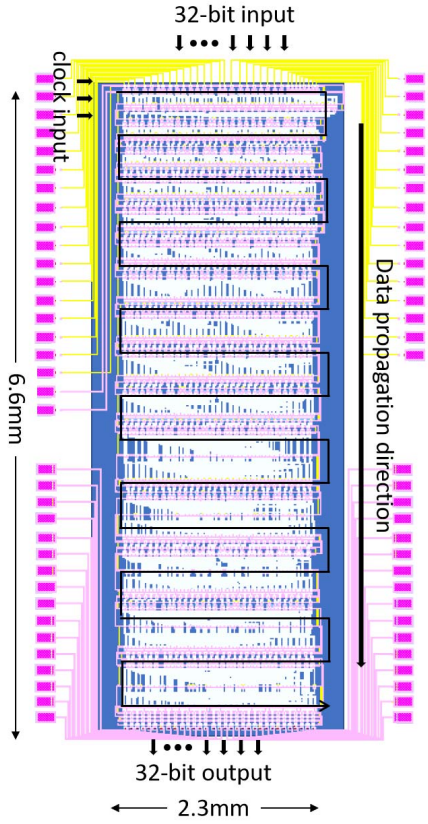

Fig. 10. AQFP 32-bit binary sorter layout, designed with the proposed buffer and splitter insertion framework.

IV. Experimental Results and Analysis

A. Experiment Setup

To test the efficiency of the proposed buffer and splitter insertion framework, several circuits including ISCAS85 benchmark circuits [20], adders, multipliers and other computational logic are tested. We used the open-source synthesis tool yosys

Algorithm 2: Virtual Splitter Mapping Algorithm

input: target virtual splitter VS
SPO_{max} is the maximum fan-out capability of splitters given by the library; $VS.fanouts$ is the set of fan-outs nodes hosted by VS;
VS_p is the parent virtual splitter of VS if exists, otherwise None;
$n\_buf_{min}(Nets)$ is the minimum value of the amounts of buffers to be connected after nets in Net;
if $length(VS.fanouts) \leq SPO_{max}$ **then**
  Map $VS.fanouts$ and $VS.in$ to a splitter;
  $n\_buf(VS.in) = n\_buf_{min}(VS.fanouts)$;
  foreach Net n_v in $VS.fanouts$ **do**
    $n\_buf(n_v) = n\_buf(n_v) - n\_buf_{min}(VS.fanouts)$;
  end
else
  if VS_p is None **then**
    Create new virtual splitter VS_p;
    $VS_p.in = VS.in$;
    if $n\_buf_{min}(VS.fanouts) \neq 0$ **then**
      foreach Net n_f in $VS.fanouts$ **do**
        $n\_buf(n_f) = n\_buf(n_f) - 1$;
      end
    end
  end
  while $VS.fanouts$ *is not empty* **do**
    Create new net $n_{internal}$;
    Pop up-to SPO_{max} amount of fan-out nets from $VS.fanouts$ with most delay, denoted as $fanouts_i$;
    Map $fanouts_i$ and n_i to splitter;
    Add n_i to $VS_p.fanouts$;
    $n\_buf(n_i) = n\_buf_{min}(fanouts_i)$;
    foreach Net n_j in $fanouts_i$ **do**
      $n\_buf(n_j) = n\_buf(n_j) - n\_buf_{min}(fanouts_i)$;
    end
  end
  remove $VS.in$ from $VS_p.fanouts$;
end
if VS_p is not None **then**
  VirtualSplitterMapping(VS_p);
end

Algorithm 3: Buffer and Splitter Insertion

input: target module netlist Mod
foreach fan-out net n closest to primary inputs in Mod **do**
  Create Virtual Splitter VS;
  foreach fan-out n_f from net n **do**
    Add n_f to $VS.fanouts$;
    Calculate and initialize $n\_buf(n_f)$;
  end
  VS_{leaf} = VirtualSplitterGeneration(VS);
  VirtualSplitterMapping(VS_{leaf});
end
Add buffers to paths and outputs;
return Mod

[19] to synthesis above designs with AQFP standard library. Netlists of the benchmark designs are then processed with the proposed method for buffer and splitter insertion. For comparison, netlists are also processed with the method proposed in [13] for buffer and splitter insertion. The maximum output capability of splitters is limited to four.

In addition to circuit benchmark tests, the proposed automatic framework is also used in AQFP circuit design for further fabrication and testing. As the layout shown in Fig. 10, the proposed method is used in the automation design of a 32-bit binary sorter in AQPF.

B. Results and Analysis

The absolute and relative comparison results of multiple circuits are shown in Table I, the proposed method is compared with the raw netlist synthesised using yosys and the balanced netlist produced via the existing method [13]. The proposed method shows reduction in resource consumption for buffers and splitters insertion. In some cases, such as simple logic or bitonic sorter with no big fan-out nets, results produced by the existing method [13] needs no further optimization. Therefore, the proposed method performs on par with the existing method on such cases. However, as the complexity of circuits grow, the proposed method can deliver up-to 40.84% reduction in JJ count for design such as a 32-bit ALU. Overall, the proposed method shows very promising results. Compare to the existing method, the proposed method can reduce JJ count by up-to 40.84% and delay by up-to 3.13%.

V. Conclusion

In this paper, we propose a complete automatic framework that is capable of inserting buffers and splitters to a given gate-level logic design to make it compatible with the AQFP technology. The proposed framework can find the solution with an optimized amount of buffers and splitters inserted to achieve equal delay for all paths in the given design with any given splitter size restriction. Experimental results showing up-to 40.84% less in size and 3.13% less in delay compared to the existing insertion method, suggests that the proposed can deliver very promising results.

Acknowledgements

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via the U.S. Army Research Office grant W911NF-17-1-0120.

TABLE I
JOSEPHSON JUNCTION COUNT AND DELAY COMPARISON BETWEEN UNPROCESSED CIRCUITS, EXISTING METHOD AND THE PROPOSED METHOD.

Device	w/o buffer splitter		Existing method[13]		Proposed method		Reduction in percentage	
	JJ count	JJ level	JJ count	JJ level	JJ count	JJ level	JJ count	JJ level
1-bit adder	42	4	82	8	74	8	9.76%	0.00%
8-bit adder	462	17	1378	33	1204	33	12.63%	0.00%
8-bit multiplier	2634	35	7610	71	6414	71	15.72%	0.00%
16-bit parallel counter	174	9	394	17	338	17	14.21%	0.00%
32-bit parallel counter	534	13	1088	23	912	23	16.18 %	0.00%
64-bit parallel counter	1296	17	2554	30	2134	30	16.44%	0.00%
128-bit parallel counter	2862	22	5564	38	4652	38	16.39%	0.00%
c17	36	3	60	5	60	5	0.00%	0.00%
c432	732	26	3078	39	2538	39	17.54%	0.00%
c499	2356	18	7748	32	5034	31	35.03%	3.13%
c880	1850	27	6890	41	5442	41	21.02%	0.00%
c1355	2352	18	7396	31	4908	30	33.64%	3.23%
c1908	1800	21	7184	37	5116	37	28.79%	0.00%
c2670	3282	20	9924	29	8204	29	17.33%	0.00%
c3540	4820	32	13332	56	10450	56	21.62%	0.00%
c5315	7960	26	31408	42	21878	42	30.34%	0.00%
c6288	11220	89	78246	180	50356	180	35.64%	0.00%
c7552	8456	33	32946	66	27180	66	17.50%	0.00%
32-bit binary sorter	2880	15	3840	30	3840	30	0.00%	0.00%
48-bit binary sorter	5280	20	7456	35	7040	35	5.58%	0.00%
32-bit RISCV ALU	9218	100	75458	172	44638	170	40.84%	1.16%

REFERENCES

[1] 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*, vol. 1, no. 1, pp. 3–28, March 1991.

[2] H. Hayakawa, N. Yoshikawa, S. Yorozu, and A. Fujimaki, "Superconducting digital electronics," *Proceedings of the IEEE*, vol. 92, no. 10, pp. 1549–1563, 2004.

[3] M. Tanaka, F. Matsuzaki, T. Kondo, N. Nakajima, Y. Yamanashi, A. Fujimaki, H. Hayakawa, N. Yoshikawa, H. Terai, and S. Yorozu, "A single-flux-quantum logic prototype microprocessor," in *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*. IEEE, 2004, pp. 298–529.

[4] T. V. Filippov, A. Sahu, A. F. Kirichenko, I. V. Vernik, M. Dorojevets, C. L. Ayala, and O. A. Mukhanov, "20 ghz operation of an asynchronous wave-pipelined rsfq arithmetic-logic unit," *Physics Procedia*, vol. 36, pp. 59–65, 2012.

[5] 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," *Japanese Journal of Applied Physics*, vol. 51, no. 5R, p. 053102, 2012. [Online]. Available: http://stacks.iop.org/1347-4065/51/i=5R/a=053102

[6] Q. P. Herr, A. Y. Herr, O. T. Oberg, and A. G. Ioannidis, "Ultra-low-power superconductor logic," *Journal of applied physics*, vol. 109, no. 10, p. 103903, 2011.

[7] N. Yoshikawa and Y. Kato, "Reduction of power consumption of rsfq circuits by inductance-load biasing," *Superconductor Science and Technology*, vol. 12, no. 11, p. 918, 1999.

[8] O. A. Mukhanov, "Energy-efficient single flux quantum technology," *IEEE Transactions on Applied Superconductivity*, vol. 21, no. 3, pp. 760–769, 2011.

[9] N. Takeuchi, D. Ozawa, Y. Yamanashi, and N. Yoshikawa, "An adiabatic quantum flux parametron as an ultra-low-power logic device," *Superconductor Science and Technology*, vol. 26, no. 3, p. 035010, 2013.

[10] S. Nagasawa, Y. Hashimoto, H. Numata, and S. Tahara, "A 380 ps, 9.5 mw josephson 4-kbit ram operated at a high bit yield," *IEEE Transactions on Applied Superconductivity*, vol. 5, no. 2, pp. 2447–2452, 1995.

[11] S. K. Tolpygo, V. Bolkhovsky, T. J. Weir, A. Wynn, D. E. Oates, L. M. Johnson, and M. A. Gouker, "Advanced fabrication processes for superconducting very large-scale integrated circuits," *IEEE Transactions on Applied Superconductivity*, vol. 26, no. 3, pp. 1–10, 2016.

[12] N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, "Adiabatic quantum-flux-parametron cell library adopting minimalist design," *Journal of Applied Physics*, vol. 117, no. 17, p. 173912, 2015.

[13] Q. Xu, C. L. Ayala, N. Takeuchi, Y. Murai, Y. Yamanashi, and N. Yoshikawa, "Synthesis flow for cell-based adiabatic quantum-flux-parametron structural circuit generation with hdl back-end verification," *IEEE Transactions on Applied Superconductivity*, vol. 27, no. 4, pp. 1–5, June 2017.

[14] A. Almatrood and H. Singh, "A comparative study of majority/minority logic circuit synthesis methods for post-cmos nanotechnologies," *Engineering*, vol. 9, no. 10, p. 890, 2017.

[15] K. Kong, Y. Shang, and R. Lu, "An optimized majority logic synthesis methodology for quantum-dot cellular automata," *IEEE Transactions on Nanotechnology*, vol. 9, no. 2, pp. 170–183, 2010.

[16] P. Wang, M. Y. Niamat, S. R. Vemuru, M. Alam, and T. Killian, "Synthesis of majority/minority logic networks," *IEEE Transactions on Nanotechnology*, vol. 14, no. 3, pp. 473–483, 2015.

[17] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1842–1855, 2017.

[18] J. Clarke and A. I. Braginski, *The SQUID handbook: Applications of SQUIDs and SQUID systems*. John Wiley & Sons, 2006.

[19] C. Wolf, "yosys," http://www.clifford.at/yosys/.

[20] http://www.pld.ttu.ee/~maksim/benchmarks/iscas85/verilog/.