Exploring the Feasibility of a DNA Computer: Design of an ALU using Sticker Based DNA Model

Mayukh Sarkar †, Prasun Ghosal †, Saraju P. Mohanty ‡
† Indian Institute of Engineering Science and Technology, Shibpur,
Howrah 711103, WB, INDIA
‡ University of North Texas, Denton, TX 76203, USA
email: {mayukh, p_ghosal}@it.iiests.ac.in,
saraju.mohanty@unt.edu

***Abstract*—Since its inception, DNA computing has advanced to offer an extremely powerful, energy-efficient emerging technology for solving hard computational problems with its inherent massive parallelism and extremely high data density. This would be much more powerful and general purpose when combined with other existing well known algorithmic solutions that exist for conventional computing architectures using a suitable ALU. Thus, a specifically designed DNA ALU that can address operations suitable for both domains can mitigate the gap between these two. A complete ALU (Arithmetic and Logic Unit) must be able to perform all possible logic operations including NOT, OR, AND, XOR, NOR, NAND and XNOR; compare, shift etc., integer and floating point arithmetic operations (Addition, Subtraction, Multiplication and Division). In this paper, design of a complete ALU has been proposed using Sticker-based DNA Model with experimental feasibility analysis. Novelties of this work may be in manifold. First, the integer arithmetic operations performed here are 2's complement arithmetic, and the floating point operations follow IEEE 754 floating point format, resembling closely to a conventional ALU. Also, the output of each operation can be reused for any next operation. So any algorithm or program logic that users can think of can be implemented directly on the DNA computer without any modification. Second, once the basic operations of sticker model can be automated, the implementations proposed in this work become highly suitable to design a fully automated complete ALU. Third, proposed approaches are easy to implement. Finally, these approaches can work on sufficiently large binary numbers.**

***Index Terms*—DNA Computing, DNA ALU, Sticker Model.**

I. Introduction

In recent years, computing using DNA molecules, namely, DNA computing, has been proved to be a powerful emerging technology to solve hard computational problem. In 1994, Adleman [1] has shown the use of this powerful tool in solving Travelling Salesman problem, a difficult combinatorial search problem. In 1995, Lipton [2], in a very similar way, solved the SAT problem, by representing all possible combination of values in a graph. Lipton's DNA algorithm for solving the SAT problem was a linear one, which is impossible for a conventional computer at this stage. In the same year, Boneh et.al. [3] broke DES, a famous encryption method, using DNA. In 1997, Ouyang et. al. [4] solved maximal clique problem given a six vertex graph. A huge achievement was made by Adleman and others in 2002, when they solved 20-variable 3-SAT problem by performing an exhaustive search over 1 million possibilities [5]. Along with these experiments, several other NP-complete problems, such as Graph Coloring [6], Bin Packing [7] etc. have been solved using DNA computing. All these experiments have shown that, inherently massive parallelism and high data storage density have made DNA computing a powerful technique to solve such problems over conventional computer. Though these works used massive amounts of data to be encoded on huge number of distinct DNA strands and each bio-molecular operation applied on a DNA solution affects all strands simultaneously leading to possibility of massive parallelism, but the proposed work in this paper looks at a different problem other than searching over a large search space. In fact, the problem targeted in the proposed work is to perform arithmetic and logic operations so that output of one operation can be used in later operations and this requires two numbers to be encoded by two different DNA strands in two separate test tubes. Hence power of massive parallelism is not visible in this work, as the target of this work is not to utilize existing power of DNA computer to solve another computationally hard problem. Rather it is to increase the power of the computing technique by showing that DNA computer can also be used effectively in arithmetic and logic operations and hence solving general problems that do not require search over large search spaces.

A. Basics of DNA

The DNA (Deoxyribonucleic acid), naturally found in living cells, is composed of four bases, namely Adenine(A), Guanine(G), Thymine(T) and Cytosine(C). Each base is attached to its neighbour base in the sequence via phosphate bonding. Base, sugar and the phosphate are together called a nucleotide. Two DNA sequences bond with each other via hydrogen bonding between each Watson-Crick complementary base pairs (A with T, and C with G), forming DNA double helix. Each DNA strand has two ends : 5'-end and 3'-end, that determine the polarity of the DNA strand. During the formation of DNA double strand, two complementary single strands bond with each other in anti-parallel fashion.

B. Recent Trends and Challenges

In 2002, a programmable molecular computing machine composed of enzymes and DNA molecules, have been developed by a group of researchers from Weizmann Institute of Science, Israel. A year later, the team advanced one step further. In the new device, the single DNA molecule, that provides the computer with the input data also provides all the necessary fuel [8]. In 2004, Benenson et. al. [9] described an autonomous bio-molecular computer that logically analyses the levels of messenger RNA species, and in response produces a molecule capable of affecting levels of gene expression. This computer, in theory, would be capable of diagnosing cancer, and producing anti-cancer drug. In 2013, Goldman et. al. [10] encoded computer files totaling 739 kB of hard disk storage and with an estimated Shannon information of 5.2×10^6 bits into a DNA code, synthesized, sequenced and then reconstructed the original data with 100% accuracy. In the same year, bio-engineers at Stanford University created the first biological transistors, named transcriptor, using DNA and RNA [11].

C. Necessity of a DNA ALU

To make DNA computing applicable on wider range of problems, simple logic and arithmetic operations available on a conventional computers, are necessary to be implemented. These operations include logic operations including NOT, OR, AND, XOR, NOR, NAND, and XNOR; compare, shift etc., integer and floating point arithmetic operations (Addition, Subtraction, Multiplication and Division). Proper and efficient implementations of these operations will allow the users to implement their algorithms and program logic directly on a DNA computer without any modification, thus making the applicability of the DNA computing technique in a much wider range.

D. Organization of the paper

Rest of the paper is organized as follows. In section II, advantages of the proposed work are described with an explanation over how the proposed work fits into the automation of a future DNA computer. Section III reports the works that have been done in the field of arithmetic and logic operation implementations using DNA molecules, and the problems they pose. Section IV describes the background information necessary to understand the proposed work viz. the Sticker-based model, biomolecular operations available on the sticker-based DNA model, and their functional formulation. Seven primary logic operations viz. AND, OR, XOR, NAND, NOR, XNOR and NOT, are described in section V with an example demonstrating step-by-step execution of AND operation. Section VI describes the possible integer arithmetic operations viz. comparator, left and right shifters (logical, arithmetic and circular), addition / subtraction, multiplication, and division. Representation of IEEE 754 floating point format and arithmetic operations possible on the floating point numbers viz. addition / subtraction, multiplication, and division are presented in section VII. Following section VIII addresses the possibility of the implementations of the proposed work, complexity analysis, and implementation issues. Finally, section IX concludes the paper with possible future directions.

II. Novel Contributions of this Paper

DNA Computing is already proven powerful for computationally "hard" problems. Inherent parallelism nature of bio-molecular operations gives this power to DNA computing. This would be much more powerful and general purpose when combined with other existing well known algorithmic solutions that exist for conventional computing architectures using a suitable ALU. Thus, a specifically designed DNA ALU that can address operations suitable for both domains can mitigate the gap between these two.

The novelties of the proposed method as well as the logic and arithmetic operations for this DNA ALU proposed in this work may be summarized as follows.

- **Ease of number representation:** Implementations are based on sticker based DNA model, and hence can explore the power of the binary number representation.
- **Ease of implementations:** Operations applicable on sticker-based DNA model are simple to implement, except the $clear$ operation. Proposed operations avoid the $clear$ operation of sticker model, and are using the simple operations, and hence physical implementation is easier.
- **Input and output of the operations are of similar structure:** The input operands are being taken as separate strands in separate test tubes, and output is also provided in separate test tube. Hence, the output of one operation can be used ready-made for any next operation, without any modification. Also, the uniform representation can be used, $i.e.$, the way an user needs to represent the input, the output can be decoded in the user-readable format in the similar manner without any modification.
- **Possibility of automation of DNA computer:** If the basic operations of sticker model, i.e. $combine$, $separate$ and set can be automated, the proposed implementations are highly suitable for the design of a fully automated complete ALU.
- **Design of a complete DNA ALU:** All logic operations, integer, and floating point arithmetic operations have been implemented. Also all representations follow conventional ideology, such as floating point operations follow IEEE 754 floating point format and so on. So any algorithm or program logic that users can perform can be implemented directly on the DNA computer without any modification.

III. Related Prior Research

The earliest attempt to perform arithmetic operations (addition of two binary numbers) using DNA was made by Guarneiri et.al. [12], but the output strands are vastly different from the input ones, and hence, cannot be reused for any next operation.

Later, Gupta et. al. [13] performed logic and arithmetic operations using fixed bit encoding scheme, but requires

manual encoding and addition of intermediate results one by one during processing of arithmetic operations. This is time consuming task, and cannot be used in an automated DNA computer.

Santis et. al. [14] proposed floating-point arithmetic and logic operations using DNA strands, but it requires long DNA strand to represent a single bit, and representation of single binary number requires multiple test tubes. Bits toward LSB in long numbers requires much longer strands to represent.

Several other approaches for implementing arithmetic and logic operations have been proposed, such as Ogihara-Ray method of boolean circuit simulation [15], Amos-Dunne method [16], Barua-Misra method [17], Qiu-Lu method [18] etc.. But none of them are based on sticker based DNA model.

Ignatova et. al. [19] have considered many sticker algorithms, including some simple arithmetic. Guo et. al. [20] proposed an implementation of arithmetic operations using sticker-based DNA model, but they did not propose logic operations and floating point operations. The two numbers under operation, along with the carry are being represented on same memory strand, and output of operation are found as a substring of the output strand. So, the output of one operation can not be reused for next one, until and unless, the remaining parts of the output strand is cleared, and next operand is set at that position. Also, they have used $clear$ operation, but as Roweis et. al. [21] have specified in the base paper of Sticker-based DNA Model , "In terms of physical implementation prospects, *clear* seems to be the most problematic of our operations".

Arnold [22], [23] proposed a more efficient sticker addition algorithm, known as tube-encoded carry; however, this operates in a similar context of [19] and [20] where both operands are represented on the same strand. The major advantage of [22] is that carries are not recorded on DNA strands, rather represented by the tube the strands are placed in. This algorithm is adapted here to make an efficient addition/subtraction algorithm in the different context where two numbers on two distinct strands of DNA produce a sum on a third strand that may be reused in later operations.

Therefore, it remains a challenging task to have all the logic and arithmetic operations implemented with same input and output DNA format, practically possible and easy to implement.

IV. Background of DNA Computer

A. Sticker-based Model

A binary number can be represented in the DNA sticker model by employing two groups of single-stranded DNA molecules. One is *memory strand*, which is a long DNA molecule, subdivided into several non-overlapping region. The other group is a set of *stickers*, which are short DNA molecules, each having length equal to the length of each region of memory strand. Each sticker is complementary to one and only one of the non-overlapping regions. Each non-overlapping region represents a bit. If a sticker is annealed to its matching region on the memory strand, that region then represents 1 bit, otherwise a 0 bit.

For example, to represent a 8-bit number, if we take the memory strand and the corresponding stickers as in Figure 1, where each sticker is complementary to each of the 8 non-overlapping regions respectively. As an example, the numbers 10101110 and 11011001 can be represented using the above sticker model as in Figure 2.

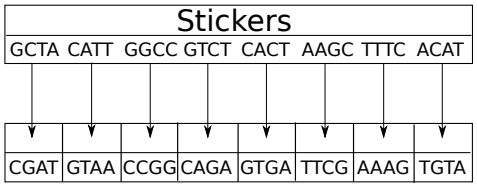


Fig. 1: An example memory strand with corresponding stickers

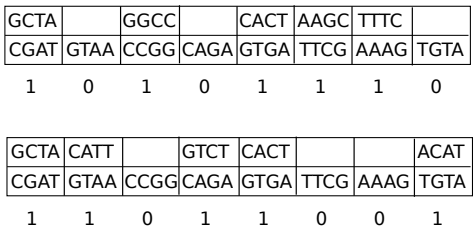


Fig. 2: An example memory strand with corresponding stickers

Any set of bit strings can be represented by identical memory strands, each memory strand having stickers annealed only at the required 1 bit positions.

B. Operations on Sticker based DNA

The operations available on sticker-based DNA strands are as follows.

1) ***Combine*** : In this operation, two sets of bit strings in two test tubes in combined in one test tube. This corresponds to obtaining a different tube containing all the memory complexes from both input tubes. The combine operation is not the same as a conventional assignment statement: combine moves DNA strands to a different physical location whereas conventional assignment makes a copy.
2) ***Separate*** : In this operation, a set of strings is separated into two sets based on a particular bit. This creates two different tubes, where one tube contains strings having that particular bit *on*, and the other tube containing the strings having the bit *off*.
3) ***Set*** : In this operation, a particular bit in every string of the DNA solution is set (turned on). The sticker

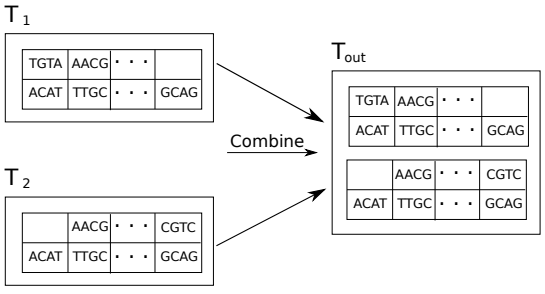


Fig. 3: Combine operation

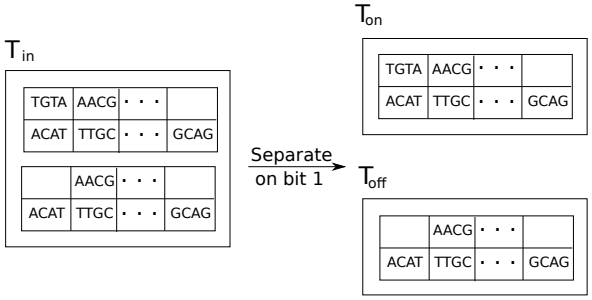


Fig. 4: Separate operation

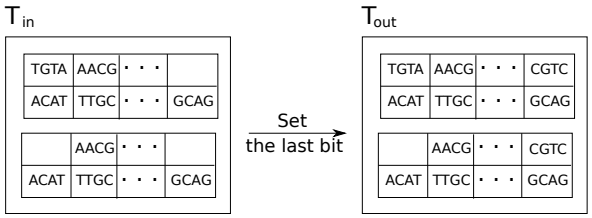


Fig. 5: Set operation

for that bit is annealed to the appropriate region on every complex in the set's tube. Setting is performed by annealing the particular sticker to the bit that needs to be set in the memory strand. Setting multiple bits can be done in parallel by pouring all the stickers corresponding to the targeted bits, and annealing. All the poured stickers will anneal with the memory strand at the same time. Excess stickers need to be filtered out of the tube before proceeding to the next operation.

4) ***Clear*** : In this operation, a particular bit in every string of a DNA solution, is removed by removing the sticker (if present) for that bit from every memory complexes in the test tube. Implementation of this operation is difficult, and hence avoided in the proposed algorithms.
5) ***Discard*** : In this operation, contents of a non-empty test tube is discarded and the test tube becomes empty after this operation.

C. Functional formulation of the operations available on Sticker-based DNA model

For the ease of representation, functional formulations of the operations described in the previous subsection IV-B will be used as follows.

1) $Combine(T_{dest}; T^1_{src}, T^2_{src}, \cdots, T^n_{src})$ - Pour the contents of T_{src}s in T_{dest}. After this operation, T_{dest} will contain the union of the contents of T_{dest} and all the T_{src}s, and T_{src}s will become empty.
2) $Separate(T_1, i, b_{on}, b_{off})$ - Separate the contents of T_1 based on the value of i^{th} bit in two test tubes b_{on}, containing DNA strands having i^{th} bit on, and b_{off}, containing DNA strands having i^{th} bit off.
3) $Set(T_1, i)$ - Set the i^{th} bit b_i of all DNA strand in test tube T_1.
4) $Discard(T)$ - Discard the contents of test tube T.

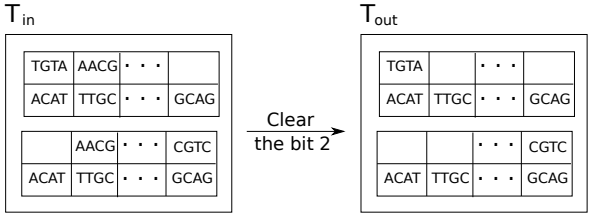


Fig. 6: Clear operation

V. Proposed Approaches for Logic Operations using DNA

All possible logic operations, $viz.$, AND, OR, XOR, NAND, NOR, XNOR, NOT are being proposed in this section. But before describing the operations to be performed in the proposed ALU, a helper operation $Substring$ needs to be discussed, that will help in making backup of input test tubes and will allow the input test tubes to stay intact, so that they can be used in other operations without recreating the numbers.

A. Substring

Substring operation takes a n-bit input string in a test tube T_{in}, a start index $start$ and an end index end, such that $0 \leq start \leq end < n$. The operation will output the required DNA strand in test tube T_{out} that will represent the substring of the input string from start index to end index, taking end index inclusive. This operation itself may not be used as a part of main ALU operations, but will be proved helpful in several of next operations. The value of n will be specified in later algorithms that use substring and is global for all routines that use Algorithm 1. Typical choices for n are 16, 32 and 64 for integers and 22, 46 and 108 for floating point (twice the mantissa size).

Algorithm 1 Sub string

```
1: procedure SUBSTRING(in : T_in, start, end; out : T_out)
2:     length ← end − start + 1;
3:     Pour blank memory strands of length bits, 00...00 (length bits), in T_out;
4:     for i ← 0 to length − 1 do
5:         Separate(T_in, start + i, b_on, b_off);
6:         if b_on is not empty then
7:             Set(T_out, i);
8:         end if
9:         Combine(T_in; b_on, b_off);
10:    end for
11: end procedure
```

Now, the logic operations can be discussed. Let two test tubes T_1 and T_2 be containing the DNA strands corresponding to the two n-bit binary numbers under operation, respectively. The test tube T_{out} will contain the output after the corresponding operation. At the beginning of each operation, T_{out} is considered to contain blank memory strands, $i.e.$, the string $\underbrace{00\ldots00}_{\text{n bits}}$.

The implementations of the logic operations as follows,

GCTA			GTCT
CGAT	GTAA	CCGG	CAGA
1	0	0	1

GCTA	CATT		
CGAT	GTAA	CCGG	CAGA
1	1	0	0

Fig. 7: DNA molecules for the example of AND operation

B. AND Operation

Algorithm 2 performs AND operation between two n-bit numbers represented using sticker model, and kept in two test tubes T_1 and T_2 respectively. The output of the operation is available in test tube T_{out}, also in sticker model representation. This algorithm checks whether both numbers have value of 1 at each bit, and runs in $O(n)$ time.

Algorithm 2 AND Logic

```
1: procedure AND(in : T_1, T_2, n; out : T_out)
2:     T_1^C ← Substring(T_1, 0, n − 1);
3:     T_2^C ← Substring(T_2, 0, n − 1);
4:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
5:     Combine(T_C; T_1^C, T_2^C);
6:     for all bit i do
7:         Separate(T_C, i, b_on, b_off);
8:         if b_off is empty then
                                   ▷ i^th bits of both numbers are 1
9:             Set(T_out, i);
10:        end if
11:        Combine(T_C; b_on, b_off);
12:    end for
13: end procedure
```

As an example, assume two 4-bit numbers, 1001 and 1100 be kept in two test tubes T_1 and T_2 respectively. Let the memory strand be $CGAT|GTAA|CCGG|CAGA$. Hence, the two numbers will be represented as in Figure 7.

Now, the algorithm runs as follows.

1) The contents of T_1 and T_2 are first taken as a backup into T_1^C and T_2^C respectively and combined into T_C. Hence, T_C now contains $\{1001, 1100\}$. Blank memory strands are taken into test tube T_{out}.
2) T_C is separated based on bit 1, b_{on} contains both 1001 and 1100, and b_{off} is empty. So, the sticker for bit 1, $GTTA$ is taken into test tube T_{out}, and annealed to set bit 1 of output.
3) T_C is separated based on bit 2, b_{on} contains 1100, and b_{off} contains 1001. As b_{off} is not empty, nothing to do. Combine b_{on} and b_{off} back into T_C.
4) T_C is separated based on bit 3, b_{on} is empty, and b_{off} contains both 1001 and 1100. As b_{off} is not empty, nothing to do. Combine b_{on} and b_{off} back into T_C.
5) T_C is separated based on bit 4, b_{on} contains 1001, and b_{off} contains 1100. As b_{off} is not empty, nothing to do. Combine b_{on} and b_{off} back into T_C.
6) T_{out} now contains the DNA strand as in Figure 8, representing the value 1000.

GCTA			
CGAT	GTAA	CCGG	CAGA
1	0	0	0

Fig. 8: DNA molecule representing output of AND operation

C. OR Operation

Algorithm 3 performs OR operation between two n-bit numbers kept in two test tubes T_1 and T_2 respectively. The output of the operation is available in test tube T_{out}. Following the similar logic of AND operation, as it checks for each bit whether atleast one of them is 1, this algorithm also takes $O(n)$ number of steps to perform.

Algorithm 3 OR Logic

```
1: procedure OR(in : T_1, T_2, n; out : T_out)
2:     T_1^C ← Substring(T_1, 0, n − 1);
3:     T_2^C ← Substring(T_2, 0, n − 1);
4:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
5:     Combine(T_C; T_1^C, T_2^C);
6:     for all bit i do
7:         Separate(T_C, i, b_on, b_off);
8:         if b_on is not empty then
                                   ▷ i^th bit of atleast one number is 1
9:             Set(T_out, i);
10:        end if
11:        Combine(T_C; b_on, b_off);
12:    end for
13: end procedure
```

D. XOR Operation

Algorithm 4 performs XOR operation between two n-bit numbers kept in two test tubes T_1 and T_2 respectively. The output of the operation is available in test tube T_{out}. This algorithm is also $O(n)$, as it is checking whether the bits of the two input numbers are different for each bit.

Algorithm 4 XOR Logic

```
1: procedure XOR(in : T_1, T_2, n; out : T_out)
2:     T_1^C ← Substring(T_1, 0, n − 1);
3:     T_2^C ← Substring(T_2, 0, n − 1);
4:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
5:     Combine(T_C; T_1^C, T_2^C);
6:     for all bit i do
7:         Separate(T_C, i, b_on, b_off);
8:         if neither b_on nor b_off is empty then
                                   ▷ i^th bits of two numbers are different
9:             Set(T_out, i);
10:        end if
11:        Combine(T_C; b_on, b_off);
12:    end for
13: end procedure
```

E. NAND Operation

Algorithm 5 performs NAND operation between two n-bit numbers kept in two test tubes T_1 and T_2 respectively. The output of the operation is available in test tube T_{out}. This algorithm runs in $O(n)$ number of steps.

Algorithm 5 NAND Logic

procedure NAND($in : T_1, T_2, n; out : T_{out}$)
  $T_1^C \leftarrow Substring(T_1, 0, n-1)$;
  $T_2^C \leftarrow Substring(T_2, 0, n-1)$;
  Pour blank memory strands of *n* bits, $\underbrace{00\ldots00}_{n \text{ bits}}$, in T_{out};
  Combine($T_C; T_1^C, T_2^C$);
  for all bit i **do**
    Separate(T_C, i, b_{on}, b_{off});
    if b_{off} is not empty **then**
      ▷ i^{th} bit of atleast one number is 0
      Set(T_{out}, i);
    end if
    Combine($T_C; b_{on}, b_{off}$);
  end for
end procedure

F. NOR Operation

Algorithm 6 performs NOR operation between two n-bit numbers kept in two test tubes T_1 and T_2 respectively. The output of the operation is available in test tube T_{out}. Similar to other logic operations, this is $O(n)$ time.

Algorithm 6 NOR Logic

procedure NOR($in : T_1, T_2, n; out : T_{out}$)
  $T_1^C \leftarrow Substring(T_1, 0, n-1)$;
  $T_2^C \leftarrow Substring(T_2, 0, n-1)$;
  Pour blank memory strands of *n* bits, $\underbrace{00\ldots00}_{n \text{ bits}}$, in T_{out};
  Combine($T_C; T_1^C, T_2^C$);
  for all bit i **do**
    Separate(T_C, i, b_{on}, b_{off});
    if b_{on} is empty **then**
      ▷ i^{th} bits of both numbers are 0
      Set(T_{out}, i);
    end if
    Combine($T_C; b_{on}, b_{off}$);
  end for
end procedure

G. XNOR Operation

Algorithm 7 performs XNOR operation between two n-bit numbers kept in two test tubes T_1 and T_2 respectively. The output of the operation is available in test tube T_{out}. This algorithm also has $O(n)$ number of steps.

Algorithm 7 XNOR Logic

procedure XNOR($in : T_1, T_2, n; out : T_{out}$)
  $T_1^C \leftarrow Substring(T_1, 0, n-1)$;
  $T_2^C \leftarrow Substring(T_2, 0, n-1)$;
  Pour blank memory strands of *n* bits, $\underbrace{00\ldots00}_{n \text{ bits}}$, in T_{out};
  Combine($T_C; T_1^C, T_2^C$);
  for all bit i **do**
    Separate(T_C, i, b_{on}, b_{off});
    if either b_{on} or b_{off} is empty **then**
      ▷ i^{th} bits of two numbers are same
      Set(T_{out}, i);
    end if
    Combine($T_C; b_{on}, b_{off}$);
  end for
end procedure

H. NOT Operation

Algorithm 8 performs NOT of a n-bit number kept in the test tube T_1. The output of the operation is available in test tube T_{out}. This algorithm checks for the value of each bit, ine by one, of the input number too and hence has $O(n)$ number of steps.

Algorithm 8 NOT Logic

procedure NOT($in : T_{in}, n; out : T_{out}$)
  Pour blank memory strands of *n* bits, $\underbrace{00\ldots00}_{n \text{ bits}}$, in T_{out};
  for all bit i **do**
    Separate($T_{in}, i, b_{on}, b_{off}$);
    if b_{on} is empty **then**
      ▷ i^{th} bit is 0
      Set(T_{out}, i);
    end if
    Combine($T_{in}; b_{on}, b_{off}$);
  end for
end procedure

VI. Proposed Approaches for Integer Arithmetic Operations using DNA

In the following operations, two numbers under operation will be considered of *n* bits each, and will be kept in two separate test tubes initially. Assume, test tube T_1 containing the number $a_0a_1 \cdots a_{n-2}a_{n-1}$, and test tube T_2 containing the number $c_0c_1 \cdots c_{n-2}c_{n-1}$. If the two numbers are of unequal bit count, smaller number will be appended with 0 bits on the left side to make them equal length. For both of the numbers, index 0 will represent the MSB, and index $n-1$ will represent the LSB. This is often called Big-Endian Order.

A. Comparator

The comparator operation 9 will compare two n-bit numbers in T_1 and T_2, and store the result in three test tubes T_g, T_l and T_e, initially empty. If unequal, T_g will contain the greater number, T_l will contain the smaller number, and T_e will remain empty. If the two numbers are equal, T_e will contain both of the numbers, and the rest of the two test tubes will remain empty. This algorithm runs in $O(n)$ number of steps.

Algorithm 9 Comparator

procedure Comparator($in : T_1, T_2, n; out : T_g, T_l, T_e$)
  $T_1^C \leftarrow Substring(T_1, 0, n-1)$;
  $T_2^C \leftarrow Substring(T_2, 0, n-1)$;
  Combine($T_C; T_1^C, T_2^C$);
  for all bit i from MSB to LSB **do**
    Separate(T_C, i, b_{on}, b_{off});
    if b_{on} or b_{off} is empty **then**
      Combine($T_C; b_{on}, b_{off}$);
    else
      Combine($T_g; b_{on}$);
      Combine($T_l; b_{off}$);
      break ▷ Exit from for loop
    end if
  end for
  if both T_g and T_l are empty **then**
    ▷ T_C contains both strands
    Combine($T_e; T_C$);
  end if
end procedure

B. Left Shifter and Right Shifter

The Left Shifter (Right Shifter) operations will take a single n-bit number in T_{in} test tube as input, left shift (right shift) the number by one bit, and output the shifted number in test tube T_{out}. T_{out} initially contains blank memory strands, i.e., the number 0000...0, no stickers are annealed to the memory strands. All types of shifters (Logical, Arithmetic and Circular) in both directions, except arithmetic left shift, are presented below, as arithmetic left shift and logical left shift are similar in nature. All of these shifters will complete in *O(n)* time.

1) Logical Left Shift: In logical left shift, all the bits except the left most bit (MSB) is shifted to the left, and the MSB is discarded. Algorithm 10 performs the logical left shift of n-bit number kept in test tube T_{in} and outputs the result in test tube T_{out}.

Algorithm 10 Logical Left Shifter

```
1: procedure LogicalLeftShift(in : T_in, n; out : T_out)
2:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
3:     for all bit i except MSB do                          ▷ Leave 0^th bit
4:         Separate(T_in, i, b_on, b_off);
5:         if b_on is not empty then                        ▷ i^th bit of the number is 1
6:             Set(T_out, i − 1);
7:         end if
8:         Combine(T_in; b_on, b_off);
9:     end for
10: end procedure
```

2) Logical Right Shift: In logical right shift, all the bits except the right most bit (LSB) is shifted to the right, and the LSB is discarded. Algorithm 11 performs the logical right shift of n-bit number kept in test tube T_{in} and outputs the result in test tube T_{out}.

Algorithm 11 Logical Right Shifter

```
1: procedure LogicalRightShift(in : T_in, n; out : T_out)
2:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
3:     for all bit i except LSB do                          ▷ Leave n − 1^th bit
4:         Separate(T_in, i, b_on, b_off);
5:         if b_on is not empty then                        ▷ i^th bit of the number is 1
6:             Set(T_out, i + 1);
7:         end if
8:         Combine(T_in; b_on, b_off);
9:     end for
10: end procedure
```

3) Arithmetic Right Shift: In arithmetic right shift, all the bits except the right most bit (LSB) is shifted to the right, the LSB is discarded, and the sign bit (MSB) is preserved. Algorithm 12 performs the arithmetic right shift of n-bit number kept in test tube T_{in} and outputs the result in test tube T_{out}.

Algorithm 12 Arithmetic Right Shifter

```
1: procedure ArithmeticRightShift(in : T_in, n; out : T_out)
2:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
3:     for all bit i except LSB do                          ▷ Leave n − 1^th bit
4:         Separate(T_in, i, b_on, b_off);
5:         if b_on is not empty then                        ▷ i^th of the number is 1
6:             if i = 0 then                                ▷ MSB is 1, preserve the MSB
7:                 Set(T_out, 0);
8:             end if
9:             Set(T_out, i + 1);
10:        end if
11:        Combine(T_in; b_on, b_off);
12:    end for
13: end procedure
```

4) Circular Left Shift: In circular left shift, all the bits except the left most bit (MSB) is shifted to the left, and the MSB is circulated to LSB. Algorithm 13 performs the circular left shift of n-bit number kept in test tube T_{in} and outputs the result in test tube T_{out}.

Algorithm 13 Circular Left Shifter

```
1: procedure CircularLeftShift(in : T_in, n; out : T_out)
2:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
3:     for all bit i do
4:         Separate(T_in, i, b_on, b_off);
5:         if b_on is not empty then                        ▷ i^th bit of the number is 1
6:             if i = 0 then                                ▷ MSB is 1, circulate to LSB
7:                 Set(T_out, n − 1);
8:             else
9:                 Set(T_out, i − 1);
10:            end if
11:        end if
12:        Combine(T_in; b_on, b_off);
13:    end for
14: end procedure
```

5) Circular Right Shift: In circular right shift, all the bits except the right most bit (LSB) is shifted to the right, and the LSB is circulated to MSB. Algorithm 14 performs the circular right shift of n-bit number kept in test tube T_{in} and outputs the result in test tube T_{out}.

Algorithm 14 Circular Right Shifter

```
1: procedure CircularRightShift(in : T_in, n; out : T_out)
2:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
3:     for all bit i do
4:         Separate(T_in, i, b_on, b_off);
5:         if b_on is not empty then                        ▷ i^th bit of the number is 1
6:             if i = n − 1 then                            ▷ LSB is 1, circulate to MSB
7:                 Set(T_out, 0);
8:             else
9:                 Set(T_out, i + 1);
10:            end if
11:        end if
12:        Combine(T_in; b_on, b_off);
13:    end for
14: end procedure
```

C. Adder / Subtracter

Addition and subtraction are two basic operations of an ALU and have been performed by several authors till date [19], [22], [23] running in $O(n)$ time. The most interesting among

them is the approach proposed in [22]. As has been specified in section III, addition/subtraction algorithm proposed here is being adapted from this work [22] and modified to include it into current context, such as, different numbers being implemented on different strands in two different tubes *etc.*. In this algorithm, two n-bit numbers are being represented by two different DNA strands in test tubes T_1 and T_2 and result of addition will be available in test tube T_{out} in the same format. The overflow of addition will be detected by the presence of a particular DNA strand in a test tube. Let us represent the overflow detecting DNA strand as some single strand $< DNA >_{Overflow}$. If after addition, overflow test tube is empty, no overflow has happened, and if that particular DNA strand is present, it signals that, an overflow has happened.

In this algorithm, two globally available test tubes T_c and T_{nc} are used. If the numbers are present in T_c, it represents there is a carry, otherwise it represents absence of carry. Along with input numbers T_1 and T_2, another test tube T_{init} is being taken as input, which acts as the starting test tube containing the numbers. In practice, the formal T_{init} will always be either the global T_c or the global T_{nc}. Choosing T_{init} as T_{nc} means there is no carry input; choosing T_{init} as T_c means there is a carry input.

Algorithm 15 Adder with carry in

```
1: procedure ADDER(in : T_1, T_2, n, T_init; out : T_out, T_overflow)
2:     Pour blank memory strands of n bits, 00...00 (n bits), in T_out;
3:     Combine(T_init; T_1, T_2);
4:     for all bit i from LSB to MSB do
5:         if T_nc is not empty then
6:             Separate(T_nc, i, b_on, b_off);
7:             if neither b_on nor b_off is empty then
                                        ▷ i^th bits of two numbers are different
8:                 Set(T_out, i);
9:                 Combine(T_nc; b_on, b_off);
10:            else if b_off is empty then
                                        ▷ i^th bits of two numbers are 1
11:                Combine(T_c; b_on, b_off);
12:            else
                                        ▷ i^th bits of two numbers are 0
13:                Combine(T_nc; b_on, b_off);
14:            end if
15:        else
16:            Separate(T_c, i, b_on, b_off);
17:            if neither b_on nor b_off is empty then
                                        ▷ i^th bits of two numbers are different
18:                Combine(T_c; b_on, b_off);
19:            else if b_off is empty then
                                        ▷ i^th bits of two numbers are 1
20:                Set(T_out, i);
21:                Combine(T_c; b_on, b_off);
22:            else
                                        ▷ i^th bits of two numbers are 0
23:                Set(T_out, i);
24:                Combine(T_nc; b_on, b_off);
25:            end if
26:        end if
27:    end for
28:    if T_c is not empty then
           ▷ There is a carry after end of addition. An overflow occurred
29:        Pour < DNA >_Overflow in T_overflow;
30:    end if
31: end procedure
```

Now, to perform addition, as the initial carry bit needs to be 0, the two inputs need to be placed in T_{nc}. The addition of two n bit numbers residing in T_1 and T_2 can now be performed as follows.

Algorithm 16 Addition

```
1: procedure ADD(in : T_1, T_2, n; out : T_out, T_overflow)
2:     T_1^C ← Substring(T_1, 0, n − 1);
3:     T_2^C ← Substring(T_2, 0, n − 1);
4:     {T_out, T_overflow} ← Adder(T_1^C, T_2^C, n, T_nc);
5: end procedure
```

Similarly $2's$ Complement subtraction can be performed by first performing NOT of second number and placing them in T_c, as follows.

Algorithm 17 Subtraction

```
1: procedure SUBTRACT(in : T_1, T_2, n; out : T_out, T_overflow)
2:     T_1^C ← Substring(T_1, 0, n − 1);
3:     T_2^C ← NOT(T_2, n);
4:     {T_out, T_overflow} ← Adder(T_1^C, T_2^C, n, T_c);
5: end procedure
```

As an example, let us consider the addition of two 4-bit numbers 1101 and 1001. Now the addition will be performed as follows.

1) Take 0000 in T_{out}.
2) Take backup of both test tubes T_1 and T_2 in T_1^C and T_2^C respectively.
3) Pour both numbers from T_1^C and T_2^C into T_{nc}.
4) Now for bit 3, as T_{nc} is not empty, perform following steps as follows.
   a) Separate T_{nc} in two test tubes based on bit 3 in b_{on} and b_{off} respectively, thus giving $b_{on} = \{1101, 1001\}$ and b_{off} is empty.
   b) So according to line 11 of algorithm 15, pour b_{on} back in T_c.
5) For bit 2, as T_{nc} is empty,
   a) Separate T_c in two test tubes based on bit 2 in b_{on} and b_{off} respectively, thus giving $b_{off} = \{1101, 1001\}$ and b_{on} is empty.
   b) According to lines 23 and 24 of the algorithm respectively, set bit 2 in T_{out} and pour b_{off} back in T_{nc}.
6) For bit 1, as T_{nc} is not empty,
   a) Separate T_{nc} in two test tubes based on bit 1 in b_{on} and b_{off} respectively, thus giving $b_{on} = \{1101\}$ and $b_{off} = \{1001\}$.
   b) So according to lines 8 and 9 of the algorithm respectively, set bit 1 in T_{out} and pour b_{on} and b_{off} back in T_{nc}.
7) For MSB (b_0), as T_{nc} is not empty,
   a) Separate T_{nc} in two test tubes based on bit 0 in b_{on} and b_{off} respectively, thus giving $b_{on} = \{1101, 1001\}$ and b_{off} is empty.
   b) So according to line 11 of the algorithm, pour b_{on} back in T_c.
8) As T_c is not empty, it detects overflow and pour $< DNA >_{Overflow}$ in $T_{overflow}$.

So, the result of the addition is 0110 with an overflow.

As it can be observed, to add two n bit numbers, for each bit, emptiness of either T_c or T_{nc} is being checked and depending on that, particular constant time operations are being performed. As in the given example, to add 1101 and 1001, total number of bio-molecular operations being performed after taking backup of the two test tubes are 12 (1 as pouring both test tubes in T_{nc}, 2 for bit 3, 3 for bit 2 and 1, and 2 operation for MSB, and finally pouring $< DNA >_{Overflow}$ in $T_{overflow}$ to signal overflow). So, in worst condition, after combining both in T_C, 3 bio-molecular operations will be performed for each bit and overflow will be signalled, giving $3n+2$ operations. So, the addition/subtraction algorithm executes in $O(n)$, since substring and NOT also execute in $O(n)$ time.

D. Multiplication

Booth's algorithm is an well known algorithm to perform 2's complement multiplication on conventional circuits. This algorithm works as follows.

1) Take the two binary numbers *x* and *y* to be multiplied. If negative, take the number in 2's complement format.
2) Set two registers *u* and *v* initially to 0. These two registers will hold the multiplication result after completion.
3) Set a single-bit register *x'* initially to 0. This register will hold the LSB of *x*.
4) Continue the following operations for the number of bits present in *x*,
   a) Look at the LSB of *x*, and the value of *x'*. Let us represent it as $x_{LSB}[x-1]$.
   b) If $x_{LSB}[x-1] = 10$, subtract *y* from *u*. Discard the overflow.
   c) If $x_{LSB}[x-1] = 01$, add *y* to *u*. Discard the overflow.
   d) Perform Arithmetic Right Shift on *u* and *v*.
   e) Copy LSB of *x* in *x'*.
   f) Perform Circular Right Shift on *x*.

To perform multiplication operation, the output will be provided as two separate strands, corresponding to *u* and *v*, in two separate test tubes T_U and T_V respectively. The two n-bit operands *x* and *y* are being taken as inputs in two separate test tubes T_X and T_Y respectively. Another test tube T_{X1} will contain single strand DNA $< DNA >_{X1}$, if *x'* is 1, otherwise it will be empty. The algorithm will be as follows.

Algorithm 18 Multiplier

procedure MULTIPLY($in : T_X, T_Y, n; out : T_U, T_V$)
  Discard(T_{X1}), if not already empty;
  Pour blank memory strands of *n* bits, $\underbrace{00\ldots00}_{n \text{ bits}}$, in T_U;
  Pour blank memory strands of *n* bits, $\underbrace{00\ldots00}_{n \text{ bits}}$, in T_V;
  for number of bits in *x* **do**
    Separate($T_X, n-1, b_{on}, b_{off}$); ▷ Identify x_{LSB}
    if b_{on} is not empty and T_{X1} is empty **then**
      ▷ $x_{LSB}[x-1] = 10$. Perform Subtraction $(u-y)$
      $\{T_{out}, T_{overflow}\} \leftarrow Subtract(T_U, T_Y, n)$;
      Discard($T_{overflow}$);
      Discard(T_U);
      Combine($T_U; T_{out}$);
    else
      if b_{off} is not empty and T_{X1} is not empty **then**
        ▷ $x_{LSB}[x-1] = 01$. Perform Addition $(u+y)$
        $\{T_{out}, T_{overflow}\} \leftarrow Add(T_U, T_Y, n)$;
        Discard($T_{overflow}$);
        Discard(T_U);
        Combine($T_U; T_{out}$);
      end if
    end if
    Combine($T_X; b_{on}, b_{off}$);
    ▷ Arithmetic Right Shift of *u* and *v*
    $T_V^1 \leftarrow LogicalRightShift(T_V, n)$;
    Discard(T_V);
    Combine($T_V; T_V^1$);
    Separate($T_U, n-1, b_{on}, b_{off}$);
    if b_{on} is not empty **then**
      Set($T_V, 0$); ▷ Set MSB of *v* = LSB of *u*
    end if
    Combine($T_U; b_{on}, b_{off}$);
    $T_U^1 \leftarrow ArithmeticRightShift(T_U, n)$;
    Discard(T_U);
    Combine($T_U; T_U^1$);
    Separate($T_X, n-1, b_{on}, b_{off}$); ▷ Set $(x-1) = x_{LSB}$
    if b_{on} is not empty **then**
      Pour DNA strand $< DNA >_{X1}$ in T_{X1};
    else
      Discard(T_{X1});
    end if
    Combine($T_X; b_{on}, b_{off}$);
    $T_X^{temp} \leftarrow CircularRightShift(T_X, n)$; ▷ Circular Right Shift of *x*
    Discard(T_X);
    Combine($T_X; T_X^{temp}$);
  end for
end procedure

E. Division

In an integer division operation, a $2n$ bit number is divided by an n bit number, and the result is obtained as two n bit numbers, one is quotient and the other one is remainder. Let us assume, the $2n$ bit dividend is being represented by two test tubes T_{DD}^1 and T_{DD}^2, containing the upper half and lower half respectively, and the divisor is being represented by a single test tube T_{DR}. As an example, if the dividend is 9421, divisor is 117 and 8 bit calculation is considered, the dividend will be represented by 16 bit binary number (00100100 11001101), the upper half 00100100 will be stored in T_{DD}^1, the lower half 11001101 will be stored in T_{DD}^2, and the 8 bit binary representation of the divisor (01110101) will be stored in T_{DR}.

The general division algorithm works as follows.

1) Store upper half of dividend in R and lower half in Q.
2) For number of bits (which in this case is 8), repeat the following steps.
3) Left Shift RQ pair.
4) Subtract the divisor from R, and keep the result in R.
5) If R is positive, set right most bit in Q to 1.
6) Else Restore R by adding back the divisor.

At the end of the above procedure, Q will contain the quotient and R will contain the remainder.

Algorithm 19 Divider

procedure DIVISION($in : T^1_{DD}, T^2_{DD}, T_{DR}, n; out : T_Q, T_R$)
  $T_R \leftarrow Substring(T^1_{DD}, 0, n-1)$;
  $T_Q \leftarrow Substring(T^2_{DD}, 0, n-1)$;
  for number of bits n **do**
    ▷ Left Shift RQ pair
    $T_{temp} \leftarrow LogicalLeftShift(T_R, n)$;
    Discard(T_R);
    Combine($T_R; T_{temp}$);
    Separate($T_Q, 0, b_{on}, b_{off}$);
    if b_{on} is not empty **then**
      Set($T_R, n-1$); ▷ Set $R_{LSB} = Q_{MSB}$
    end if
    Combine($T_Q; b_{on}, b_{off}$);
    $T_{temp} \leftarrow LogicalLeftShift(T_Q, n)$;
    Discard(T_Q);
    Combine($T_Q; T_{temp}$);
    $\{T_{out}, T_{overflow}\} \leftarrow Subtract(T_R, T_{DR}, n)$; ▷ Subtract divisor from R
    Discard $T_{overflow}$;
    Discard(T_R);
    Combine($T_R; T_{out}$);
    Separate($T_R, 0, b_{on}, b_{off}$); ▷ Check MSB of R
    if b_{on} is empty **then** ▷ R is positive, set Q[LSB] = 1
      Set($T_Q, n-1$);
      Combine($T_R; b_{on}, b_{off}$);
    else ▷ R is negative, add divisor back to restore
      Combine($T_R; b_{on}, b_{off}$);
      $\{T_{out}, T_{overflow}\} \leftarrow Add(T_R, T_{DR}, n)$;
      Discard($T_{overflow}$);
      Discard(T_R);
      Combine($T_R; T_{out}$);
    end if
  end for
end procedure

VII. Proposed Approaches for Floating Point Arithmetic Operations using DNA

In IEEE 754 format [24], half-precision floating point numbers are represented by 16 bits, single-precision floating point numbers are represented by 32 bits, and double-precision floating point numbers are represented by 64 bits. The operations performed in this work are for double-precision floating point numbers, but they can be modified to work for half or single precision too, by just changing the exponent field length (n_{exp}) and mantissa field length (n_{mant}), and the bias.

In IEEE 754 floating point format, the MSB (leftmost bit) represents the sign of the number, 0 for negative number and 1 for the positive number. Next is the exponent (11 bits for double-precision, 8 bits for single-precision or 5 bits for half-precision). Finally, is the mantissa (52 bits for double-precision, 23 bits for single-precision or 10 bits for half-precision). The mantissa is normalized to have a 1 before the decimal point, and is excluded in the representation. The actual exponent is added with the bias (1023 for double-precision, 127 for single-precision or 15 for half-precision) to get the stored exponent. Full compliance with the IEEE-754 standard requires adherence to rules that deal with exceptional cases outside the range of normalized values: subnormals (including zero) when underflow occurs, signed infinity when overflow occurs, NaN (Not a Number) when a result cannot be represented. The proposed algorithms ignore these issues. Full compliance also requires implementing several rounding modes, of which only truncation is considered here. The standard describes both binary and decimal floating-point arithmetic, of which only binary is considered here.

As an example of 32-bit single precision format, the binary number $11000000101100110000000000000000$ can be separated into 3 parts, first bit 1 specifies that the number is negative. Next 8 bits 10000001 represents the biased exponent with decimal value 129. The next 23 bits 01100110000000000000000 represents the normalized mantissa, with original mantissa being $1.01100110000000000000000$. So the decimal equivalent is thus as follows.

$$\begin{aligned}&(11000000101100110000000000000000)_2\\ &= (-1) \times 2^{129-127} \times \{1 + 2^{-2} + 2^{-3} + 2^{-6} + 2^{-7}\}\\ &= -(1 + 0.25 + 0.125 + 0.015625 + 0.0078125) \times 4\\ &= -(1.3984375 \times 4) = (-5.59375)_{10}\end{aligned}$$

Similarly to get the binary equivalent of a decimal real number, let the number to be converted is 0.765. As the number is positive, the sign bit should be 0. To get the exponent, the number needs to be multiplied by 2 until the number gets represented in the form $(1 + fraction)$. We get, $0.765 \times 2 = 1.53$, and hence, the number can be represented as 1.53×2^{-1}. So, the power used is -1 and as the bias for single precision format is 127, the exponent becomes $-1 + 127 = 126 = (01111110)_2$. To get the mantissa, the 1 needs to be removed from the front. Then 0.53 is converted into equivalent binary by repetitively multiplying by 2 until 23-bit binary equivalent found. The mantissa thus obtained is 10000111101011100001010. So the binary equivalent of 0.765 is $00111111010000111101011100001010$.

A floating point number will be represented as $b_0 b_1 \dots b_{n-1}$, where n is the length of the bit string (16 bits for half-precision, 32 bits for single-precision, and 64 bits for double-precision format). So, for all formats bit b_0 will be the sign bit; along with this, for half-precision format b_1 to b_5 will be the exponent, and b_6 to b_{15} will be the mantissa; for single-precision format b_1 to b_8 will be the exponent, and b_9 to b_{31} will be the mantissa; and for double-precision format b_1 to b_{11} will be the exponent, and b_{12} to b_{63} will be the mantissa.

Before starting the arithmetic operations, we need some more operations to implement.

The first operation among them is $FPComparator$, that will compare the exponents of two numbers $num1$ and $num2$, kept in two test tubes $T1_{exp}$ and $T2_{exp}$ respectively, and will output two test tubes $T1^{hi}_{exp}$ and T^{eq}_{exp}. The presence of some particular single DNA strand $< DNA >_E$ will be used as event. After the operation, the comparison result can be determined from the emptiness of the two output test tubes as follows.

- If $T1^{hi}_{exp}$ is non-empty and T^{eq}_{exp} is empty, exponent of $num1$ is higher.
- If both test tubes are empty, exponent of $num2$ is higher.
- If $T1^{hi}_{exp}$ is empty and T^{eq}_{exp} is non-empty, both exponents are equal.
- Both test tubes cannot be non-empty simultaneously.

Algorithm 20 Floating Point Exponent Comparator

procedure FPCOMPARATOR($in : T1_{exp}, T2_{exp}; out : T1^{hi}_{exp}, T^{eq}_{exp}$)
  Pour $< DNA >_E$ in T^{eq}_{exp};
  for all bit i from MSB to LSB in exponents **do**
    Separate($T1_{exp}, i, b^{T1}_{on}, b^{T1}_{off}$);
    Separate($T2_{exp}, i, b^{T2}_{on}, b^{T2}_{off}$);
    if both of b^{T1}_{on} and b^{T2}_{on} are empty **then**
      Combine($T1_{exp}; b^{T1}_{on}, b^{T1}_{off}$);
      Combine($T2_{exp}; b^{T2}_{on}, b^{T2}_{off}$);
    end if
    if neither b^{T1}_{on} nor b^{T2}_{on} is empty **then**
      Combine($T1_{exp}; b^{T1}_{on}, b^{T1}_{off}$);
      Combine($T2_{exp}; b^{T2}_{on}, b^{T2}_{off}$);
    end if
    if b^{T1}_{on} is not empty and b^{T2}_{on} is empty **then** ▷ $T1_{exp}$ larger
      Combine($T1_{exp}; b^{T1}_{on}, b^{T1}_{off}$);
      Combine($T2_{exp}; b^{T2}_{on}, b^{T2}_{off}$);
      Pour $< DNA >_E$ in $T1^{hi}_{exp}$;
      Discard(T^{eq}_{exp});
      break;
    end if
    if b^{T1}_{on} is empty and b^{T2}_{on} is not empty **then** ▷ $T2_{exp}$ larger
      Combine($T1_{exp}; b^{T1}_{on}, b^{T1}_{off}$);
      Combine($T2_{exp}; b^{T2}_{on}, b^{T2}_{off}$);
      Discard(T^{hi}_{exp});
      Discard(T^{eq}_{exp});
      break;
    end if
  end for
end procedure

The next operation that will be needed is $Append\_Bit\_At\_Head$, that will take an input string of n bits in a test tube T_{in}, and a bit $head$ to set at the head, and will output a test tube T_{out}. T_{out} will contain the string that is longer than the string in T_{in} by 1 bit.

Algorithm 21 Append Bit At Head

procedure $Append\_Bit\_At\_Head(in : T_{in}, n, head; out : T_{out})$
  Pour blank memory strands of $(n+1)$ bits, $\underbrace{00\ldots00}_{(n+1)\text{ bits}}$, in T_{out};
  if $head$ is 1 **then**
    Set($T_{out}, 0$);
  end if
  for all bit i **do**
    Separate($T_{in}, i, b_{on}, b_{off}$);
    if b_{on} is not empty **then**
      Set($T_{out}, i+1$);
    end if
    Combine($T_{in}; b_{on}, b_{off}$);
  end for
end procedure

Another operation that will be required is $Setbits$. This algorithm takes the input string of n bits in test tube T_{in}, two indices of source number $start$ and end and an $offset$ index of the output number. It copies the bits between indices $start$ and end of the source number in T_{in} to the indices starting at $offset$ of n-bit output number already kept in T_{out}.

Algorithm 22 Set Bits

procedure SETBITS($in : T_{in}, start, end, offset; out : T_{out}$)
  $length \leftarrow end - start + 1$;
  for $i \leftarrow 0$ to $length$ **do**
    Separate($T_{in}, start + i, b_{on}, b_{off}$);
    if b_{on} is not empty **then**
      Set($T_{out}, offset + i$);
    end if
    Combine($T_{in}; b_{on}, b_{off}$);
  end for
end procedure

A. Add / Subtract

Addition of two floating point numbers is performed in the following steps.

1) Add 1 bit at the head of the two mantissas. This will give us the real mantissas, as during storing, the 1 before the decimal point is excluded.
2) Equalize two exponents by continually adding 1 to the smaller number exponent and shifting the corresponding mantissa to the right, until the two exponents are equal.
3) If the sign bits of the two numbers are equal, add the mantissas. Set the sign bit of the result according to the sign bits of the operands. If overflow occurs, right shift the result by one bit, and increment the exponent by 1 to make the result normalized.
4) If the sign bits of the two numbers are different, append 0 bit at the head of the mantissas. Perform 2's complement subtraction on the two mantissas.
   a) If the result of the subtraction is negative, take the 2's complement of the mantissa, and set the sign bit of the output as negative.
   b) Continually left shift the mantissa, and subtract 1 from the exponent, until the output is normalized.

To perform subtraction of two numbers, just invert the sign bit of the number to be subtracted, and perform the above operation.

The DNA implementation of the floating point adder/subtracter is as follows. This implementation takes two floating point numbers num_1 and num_2 in two test tubes T_1 and T_2 respectively, and outputs the result in test tube T_{out}.

Algorithm 23 Floating Point Adder

procedure FPADD($in : T_1, T_2, n_{exp}, n_{mant}; out : T_{out}$)
  Pour blank memory strands of $(n_{exp} + n_{mant} + 1)$ bits in T_{out};
    ▷ Get exponents and mantissas separated from the two numbers
  $T1_{exp} \leftarrow Substring(T_1, 1, n_{exp})$;
  $T1_{mant} \leftarrow Substring(T_1, n_{exp}+1, n_{exp}+n_{mant})$;
  $T2_{exp} \leftarrow Substring(T_2, 1, n_{exp})$;
  $T2_{mant} \leftarrow Substring(T_2, n_{exp}+1, n_{exp}+n_{mant})$;
    ▷ Make an increment / decrement operand ready.
  Pour $\underbrace{00\ldots01}_{n_{exp}\text{bits}}$ in T^{incr}_{exp};
  $T1^{original}_{mant} \leftarrow Append\_Bit\_At\_Head(T1_{mant}, n_{mant}, 1)$;
  $T2^{original}_{mant} \leftarrow Append\_Bit\_At\_Head(T2_{mant}, n_{mant}, 1)$;
    ▷ Append 1 bit at the head of the mantissas
  $\{T1^{higher}_{exp}, T^{equal}_{exp}\} \leftarrow FPComparator(T1_{exp}, T2_{exp})$;
  if T^{equal}_{exp} is empty **then** ▷ Exponents are not equal
    if $T1^{higher}_{exp}$ is not empty **then** ▷ $T1_{exp}$ higher

while $T1_{exp}$ and $T2_{exp}$ are not equal **do**
▷ Equality can be checked by calling $Comparator(T1_{exp}, T2_{exp})$ and checking emptiness of T_{equal}
▷ Right shift $T2_{mant}^{original}$ and increment $T2_{exp}$
$T2_{mant}^{shifted} \leftarrow LogicalRightShift(T2_{mant}^{original}, n_{mant}+1)$;
Discard($T2_{mant}^{original}$);
Combine($T2_{mant}^{original}; T2_{mant}^{shifted}$);
$\{T_{ADD}, T_{Overflow}\} \leftarrow Add(T2_{exp}, T_{exp}^{incr}, n_{exp})$;
Discard($T_{Overflow}$);
Discard($T2_{exp}$);
Combine ($T2_{exp}; T_{ADD}$);
end while
else ▷ $T2_{exp}$ higher
while $T1_{exp}$ and $T2_{exp}$ are not equal **do**
▷ Right shift $T1_{mant}^{original}$ and increment $T1_{exp}$
$T1_{mant}^{shifted} \leftarrow LogicalRightShift(T1_{mant}^{original}, n_{mant}+1)$;
Discard($T1_{mant}^{original}$);
Combine($T1_{mant}^{original}; T1_{mant}^{shifted}$);
$\{T_{ADD}, T_{Overflow}\} \leftarrow Add(T1_{exp}, T_{exp}^{incr}, n_{exp})$;
Discard($T_{Overflow}$);
Discard($T1_{exp}$);
Combine ($T1_{exp}; T_{ADD}$);
end while
end if
end if
Separate($T_1, 0, b_{on}^{T1}, b_{off}^{T1}$); ▷ Separate on sign bit
Separate($T_2, 0, b_{on}^{T2}, b_{off}^{T2}$);
Take a blank test tube T_{op} and a DNA Strand $< DNA >_A$;
▷ Empty T_{op} determines subtract operation, else addition
if both b_{on}^{T1} and b_{on}^{T2} are empty **then** ▷ Both numbers positive
Pour $< DNA >_A$ in T_{op};
$\{T_{add}, T_{overflow}\} \leftarrow Add(T1_{mant}^{original}, T2_{mant}^{original}, n_{mant}+1)$;
end if
if neither b_{on}^{T1} nor b_{on}^{T2} are empty **then** ▷ Both numbers negative
Pour $< DNA > A$ in T_{op};
Set($T_{out}, 0$); ▷ Result negative. Set MSB
$\{T_{add}, T_{overflow}\} \leftarrow Add(T1_{mant}^{original}, T2_{mant}^{original}, n_{mant}+1)$;
end if
if b_{on}^{T1} is empty and b_{on}^{T2} is not empty **then**
▷ T_1 positive, T_2 negative
$T1_{mant}^{subtract} \leftarrow Append\_Bit\_At\_Head(T1_{mant}^{original}, n_{mant}+1, 0)$;
$T2_{mant}^{subtract} \leftarrow Append\_Bit\_At\_Head(T2_{mant}^{original}, n_{mant}+1, 0)$;
$\{T_{add}, T_{overflow}\} \leftarrow Subtract(T1_{mant}^{subtract}, T2_{mant}^{subtract}, n_{mant}+2)$;
Discard($T_{overflow}$);
end if
if b_{on}^{T1} is not empty and b_{on}^{T2} is empty **then**
▷ T_1 negative, T_2 positive
$T1_{mant}^{subtract} \leftarrow Append\_Bit\_At\_Head(T1_{mant}^{original}, n_{mant}+1, 0)$;
$T2_{mant}^{subtract} \leftarrow Append\_Bit\_At\_Head(T2_{mant}^{original}, n_{mant}+1, 0)$;
$\{T_{add}, T_{overflow}\} \leftarrow Subtract(T2_{mant}^{subtract}, T1_{mant}^{subtract}, n_{mant}+2)$;
Discard($T_{overflow}$);
end if
if T_{op} is not empty **then** ▷ Addition performed
if $T_{overflow}$ is not empty **then**
▷ Overflow occurred. Normalize mantissa by right shift
$T_{matissa}^{normalized} \leftarrow LogicalRightShift(T_{add}, n_{mant}+1)$;
$\{T_{exp}^{normalized}, T_{overflow}\} \leftarrow Add(T1_{exp}, T_{exp}^{incr}, n_{exp})$;
Discard($T_{overflow}$);
$T_{out} \leftarrow Setbits(T_{exp}^{normalized}, 0, n_{exp}-1, 1)$;
$T_{out} \leftarrow Setbits(T_{mantissa}^{normalized}, 1, n_{mant}, n_{exp}+1)$;
else
$T_{out} \leftarrow Setbits(T1_{exp}, 0, n_{exp}-1, 1)$;
$T_{out} \leftarrow Setbits(T_{add}, 1, n_{mant}, n_{exp}+1)$;
end if
else ▷ Subtraction performed
Separate($T_{add}, 0, b_{on}, b_{off}$);
if b_{on} is not empty **then** ▷ negative result of subtraction
Set($T_{out}, 0$); ▷ Set MSB of output
Combine($T_{add}; b_{on}, b_{off}$);
▷ Perform 2's complement of the result
$T_{add}^{negated} \leftarrow NOT(T_{add}, n_{mant}+2)$;
Pour $\underbrace{00\ldots01}_{(n_{mant}+2)\text{ bits}}$ in T_{mant}^{incr};
$\{T_{2c}, T_{temp}\} \leftarrow Add(T_{add}^{negated}, T_{mant}^{incr}, n_{mant}+2)$;
Discard(T_{temp});
Discard(T_{add});
Combine($T_{add}; T_{2c}$);
end if
Combine($T_{add}; b_{on}, b_{off}$);
▷ Normalize result, if necessary, by left shifting mantissa. b_1 is the digit before decimal point
Separate($T_{add}, 1, b_{on}, b_{off}$);
while b_{on} is empty **do** ▷ while 1^{st} index is not 1
Combine($T_{add}; b_{on}, b_{off}$);
$T_{mantissa}^{normalized} \leftarrow LogicalLeftShift(T_{add}, n_{mant}+2)$;
Discard(T_{add});
Combine($T_{add}; T_{mantissa}^{normalized}$);
$T_{exp}^{normalized} \leftarrow Subtract(T1_{exp}, T_{exp}^{incr}, n_{mant}+2)$;
Discard($T1_{exp}$);
Combine($T1_{exp}; T_{exp}^{normalized}$);
Separate($T_{add}, 1, b_{on}, b_{off}$);
end while
Combine($T_{add}; b_{on}, b_{off}$);
$T_{out} \leftarrow Setbits(T1_{exp}, 0, n_{exp}-1, 1)$;
$T_{out} \leftarrow Setbits(T_{add}, 2, n_{mant}+1, n_{exp}+1)$;
end if
Combine($T_1; b_{on}^{T1}, b_{off}^{T1}$);
Combine($T_2; b_{on}^{T2}, b_{off}^{T2}$);
end procedure

Next, subtraction algorithm for two floating-point numbers is as follows.

Algorithm 24 Floating Point Subtracter

procedure FPSUBTRACT($in: T_1, T_2, n_{exp}, n_{mant}; out: T_{out}$)
▷ Invert sign bit of T_2
Pour blank memory strands of $(n_{exp}+n_{mant}+1)$ bits in $T_2^{negated}$;
Separate($T_2, 0, b_{on}, b_{off}$);
if b_{on} is empty **then** ▷ sign bit of T_2 is 0
Set($T_2^{negated}, 0$);
end if
Combine b_{on} and b_{off} in T_2;
▷ Copy rest of the bits of T_2
$T_2^{negated} \leftarrow Setbits(T_2, 1, n_{exp}+n_{mant}, 1)$;
$T_{out} \leftarrow FPAdd(T_1, T_2^{negated})$;
end procedure

B. Multiplication

Multiplication of two floating point numbers can be performed following the steps as.

1) Add two exponents. This addition must be performed on real exponents. So this step is performed by first subtracting bias from the two exponents, then adding the real exponents, and finally adding bias to the addition result.
2) Multiply the two mantissas.
3) Normalize mantissa by right shifting and incrementing exponent, if necessary.

The multiplication can be implemented using Sticker based DNA as follows.

Algorithm 25 Floating Point Multiplication

procedure FPMULTIPLY($in : T_1, T_2, n_{exp}, n_{mant}; out : T_{out}$)
  Pour blank memory strands of $(n_{exp} + n_{mant} + 1)$ bits in T_{out};
  ▷ Get exponents and mantissas separated from the two numbers
  $T1_{exp} \leftarrow Substring(T_1, 1, n_{exp})$;
  $T1_{mant} \leftarrow Substring(T_1, n_{exp} + 1, n_{exp} + n_{mant})$;
  $T2_{exp} \leftarrow Substring(T_2, 1, n_{exp})$;
  $T2_{mant} \leftarrow Substring(T_2, n_{exp} + 1, n_{exp} + n_{mant})$;
  ▷ Make bias ready in test tube T_{bias}
  Pour n_{exp} bit binary equivalent of $2^{n_{exp}-1} - 1$ in T_{bias};
  $\{T1_{exp}^{unbiased}, T_{overflow}\} \leftarrow Subtract(T1_{exp}, T_{bias}, n_{exp})$;
  Discard($T_{overflow}$);
  $\{T2_{exp}^{unbiased}, T_{overflow}\} \leftarrow Subtract(T2_{exp}, T_{bias}, n_{exp})$;
  Discard($T_{overflow}$);
  $\{T_{exp}^{unbiased}, T_{overflow}\} \leftarrow Add(T1_{exp}^{unbiased}, T2_{exp}^{unbiased}, n_{exp})$;
  Discard($T_{overflow}$);
  $\{T_{exp}, T_{overflow}\} \leftarrow Add(T_{exp}^{unbiased}, T_{bias}, n_{exp})$;
  Discard($T_{overflow}$); ▷ We have Booth's algorithm ready at hand. To make use of that, append 01 before mantissas
  Append "01" at the head of $T1_{mant}$ by calling *Append_Bit_At_Head* twice; Let the output be $T1_{mant}^{multiply}$;
  Append "01" at the head of $T2_{mant}$ by calling *Append_Bit_At_Head* twice; Let the output be $T2_{mant}^{multiply}$;
  $\{T_1^{multiply}, T_2^{multiply}\} \leftarrow Multiply(T1_{mant}^{multiply}, T2_{mant}^{multiply}, n_{mant} + 2)$;
  ▷ Get multiplication result in single DNA strand
  Pour blank memory strands of $2 * (n_{mant} + 2)$ bits in $T_{multiply}$;
  $T_{multiply} \leftarrow Setbits(T_1^{multiply}, 0, n_{mant} + 1, 0)$;
  $T_{multiply} \leftarrow Setbits(T_2^{multiply}, 0, n_{mant} + 1, n_{mant} + 2)$;
  Separate($T_{multiply}, 3, b_{on}, b_{off}$);
  if b_{on} is empty **then** ▷ Not normalized. Right shift to normalize
    Combine($T_{multiply}; b_{on}, b_{off}$);
    $T_{multiply}^{shifted} \leftarrow LogicalRightShift(T_{multiply}, 2 * (n_{mant} + 2))$;
    Discard($T_{multiply}$);
    Combine($T_{multiply}; T_{multiply}^{shifted}$);
    Pour $\underbrace{00\ldots01}_{11\text{ bits}}$ in T_{exp}^{incr};
    $\{T_{ADD}, T_{Overflow}\} \leftarrow Add(T_{exp}, T_{exp}^{incr}, n_{exp})$;
    Discard($T_{Overflow}$);
    Discard(T_{exp});
    Combine($T_{exp}; T_{ADD}$);
  end if
  Combine($T_{multiply}; b_{on}, b_{off}$);
  Separate($T_1, 0, b_{on}^{T1}, b_{off}^{T1}$);
  Separate($T_2, 0, b_{on}^{T2}, b_{off}^{T2}$);
  if One of b_{on}^{T1} and b_{on}^{T2} is non-empty, and the other one is empty **then**
    ▷ T_1 and T_2 have different sign bits
    Set($T_{out}, 0$);
  end if
  Combine($T_1; b_{on}^{T1}, b_{off}^{T1}$);
  Combine($T_2; b_{on}^{T2}, b_{off}^{T2}$);
  $T_{out} \leftarrow Setbits(T_{exp}, 0, n_{exp} - 1, 1)$;
  $T_{out} \leftarrow Setbits(T_{multiply}, 4, n_{mant} + 3, n_{exp} + 1)$;
end procedure

C. Division

The division of two floating point numbers is implemented by convergence technique. Before implementing the division operation, let us implement reciprocal operation. In the convergence technique, the numerator and denominator is multiplied by same number $R_0R_1 \cdots R_{m-1}$, so that denominator converges to 1.

$$\frac{N}{D} = \frac{NR_0R_1 \cdots R_{m-1}}{DR_0R_1 \cdots R_{m-1}} = \frac{NR_0R_1 \cdots R_{m-1}}{1} = NR_0R_1 \cdots R_{m-1} \quad (1)$$

The convergence technique is implemented as follows.

1) Normalize D to be within 0.5 and 1. To do that, divide D by 2^{e+1}, where e is the exponent of D.
2) Consider $D_0 \leftarrow D$ and $N_0 \leftarrow N$. To find reciprocal, N is 1.
3) For number of steps m, do the following,
   a) Select $R_{i-1} \leftarrow 2 - D_{i-1}$.
   b) Select $N_i \leftarrow N_{i-1}R_{i-1}$.
   c) Select $D_i \leftarrow D_{i-1}R_{i-1}$

   where i starts from 1.
4) Output $\frac{N_m}{2^{e+1}}$.

The number of steps m for an n bit number is $log_2 n$. For 64 bit double precision floating point division, m is 6.

To perform division, we need another algorithm $Absolute\_Divide\_By\_Power\_Of\_2$, which inputs a floating point number in test tube T_{in} and a biased integer exp in test tube T_{exp}, and will output a test tube T_{out}, that will contain the floating point result of dividing the content of T_{in} by 2^{exp+1}. This algorithm will return absolute value of result of division, sign to be assigned by the next $Reciprocal$ algorithm itself. The division by 2 can be performed by simply decrementing the exponent. So, to divide by 2^{exp+1}, the exponent needs to be decremented $(exp + 1)$ times. Note that, the biased exponent exp must be compared relative to the bias (1023 for double-precision, 127 for single-precision or 15 for half-precision).

Algorithm 26 Absolute Divide By Power of 2

procedure ABSOLUTE_DIVIDE_BY_POWER_OF_2($in : T_{in}, T_{exp}, n_{exp}, n_{mant}; out : T_{out}$)
  Pour blank memory strands of $(n_{exp} + n_{mant} + 1)$ bits in T_{out};
  ▷ Get exponent and mantissa separated from the number
  $T1_{exp} \leftarrow Substring(T_{in}, 1, n_{exp})$;
  $T1_{mantissa} \leftarrow Substring(T_{in}, n_{exp} + 1, n_{exp} + n_{mant})$;
  ▷ Make bias ready in test tube T_{bias}
  Pour n_{exp} bit binary equivalent of $2^{n_{exp}-1} - 1$ in T_{bias};
  Pour $\underbrace{00\ldots01}_{n_{exp}\text{ bits}}$ in T_{exp}^{decr};
  ▷ Compare biased exp with bias
  $\{T_{greater}, T_{lesser}, T_{equal}\} \leftarrow Comparator(T_{exp}, T_{bias}, n_{exp})$;
  if T_{equal} is empty **then** ▷ exp is different than bias
    $\{T_{g1}, T_{l1}, T_{e1}\} \leftarrow Comparator(T_{exp}, T_{greater}, n_{exp})$;
    if T_{e1} is not empty **then** ▷ exp is greater than bias
      while contents of T_{exp} and T_{bias} are not equal **do**
        ▷ Decrement both T_{exp} and $T1_{exp}$
        $\{T_{ADD}, T_{Overflow}\} \leftarrow Subtract(T1_{exp}, T_{exp}^{decr}, n_{exp})$;
        Discard($T_{Overflow}$);
        Discard($T1_{exp}$);
        Combine($T1_{exp}; T_{ADD}$);

16: $\{T_{ADD}, T_{Overflow}\} \leftarrow Subtract(T_{exp}, T_{exp}^{decr}, n_{exp})$;
17: Discard($T_{Overflow}$);
18: Discard(T_{exp});
19: Combine($T1_{exp}; T_{ADD}$);
20: **end while**
▷ Exponent of T_{in} decremented $(exp - bias)$ times. Decrement once more
21: $\{T_{ADD}, T_{Overflow}\} \leftarrow Subtract(T1_{exp}, T_{exp}^{decr}, n_{exp})$;
22: Discard($T_{Overflow}$);
23: Discard($T1_{exp}$);
24: Combine($T1_{exp}; T_{ADD}$);
25: **end if**
26: **end if**
27: $T_{out} \leftarrow Setbits(T1_{exp}, 0, n_{exp} - 1, 1)$;
28: $T_{out} \leftarrow Setbits(T1_{mantissa}, 0, n_{mant} - 1, n_{exp} + 1)$;
29: **end procedure**

Now the reciprocal operation may be written as follows.

Algorithm 27 Reciprocal

1: **procedure** RECIPROCAL($in : T_{in}, n_{exp}, n_{mant}; out : T_{out}$)
2: $T_{exp} \leftarrow Substring(T_{in}, 1, n_{exp})$;
3: $T_{exp2} \leftarrow Substring(T_{in}, 1, n_{exp})$;
4: Pour floating point representation of 2.0 in T_{num2};
▷ Normalize D to be within 0.5 and 1
5: $T_{D0} \leftarrow Absolute\_Divide\_By\_Power\_Of\_2(T_{in}, T_{exp}, n_{exp}, n_{mant})$;
▷ $R_0 = 2 - D_0, D_1 = R_0 * D_0, N_1 = R_0$
6: $T_{R0} \leftarrow FPSubtract(T_{num2}, T_{D0}, n_{exp}, n_{mant})$;
7: $T_{D1} \leftarrow FPMultiply(T_{R0}, T_{D0}, n_{exp}, n_{mant})$;
▷ $R_1 = 2 - D_1, D_2 = R_1 * D_1, N_2 = R_0 * R_1$
8: $T_{R1} \leftarrow FPSubtract(T_{num2}, T_{D1}, n_{exp}, n_{mant})$;
9: $T_{D2} \leftarrow FPMultiply(T_{R1}, T_{D1}, n_{exp}, n_{mant})$;
10: $T_{N2} \leftarrow FPMultiply(T_{R0}, T_{R1}, n_{exp}, n_{mant})$;
▷ $R_2 = 2 - D_2, D_3 = R_2 * D_2, N_3 = N_2 * R_2$
11: $T_{R2} \leftarrow FPSubtract(T_{num2}, T_{D2}, n_{exp}, n_{mant})$;
12: $T_{D3} \leftarrow FPMultiply(T_{R2}, T_{D2}, n_{exp}, n_{mant})$;
13: $T_{N3} \leftarrow FPMultiply(T_{N2}, T_{R2}, n_{exp}, n_{mant})$;
▷ $R_3 = 2 - D_3, D_4 = R_3 * D_3, N_4 = N_3 * R_3$
14: $T_{R3} \leftarrow FPSubtract(T_{num2}, T_{D3}, n_{exp}, n_{mant})$;
15: $T_{D4} \leftarrow FPMultiply(T_{R3}, T_{D3}, n_{exp}, n_{mant})$;
16: $T_{N4} \leftarrow FPMultiply(T_{N3}, T_{R3}, n_{exp}, n_{mant})$;
▷ $R_4 = 2 - D_4, D_5 = R_4 * D_4, N_5 = N_4 * R_4$
17: $T_{R4} \leftarrow FPSubtract(T_{num2}, T_{D4}, n_{exp}, n_{mant})$;
18: $T_{D5} \leftarrow FPMultiply(T_{R4}, T_{D4}, n_{exp}, n_{mant})$;
19: $T_{N5} \leftarrow FPMultiply(T_{N4}, T_{R4}, n_{exp}, n_{mant})$;
▷ $R_5 = 2 - D_5, N_6 = N_5 * R_5$, No need to calculate last D
20: $T_{R5} \leftarrow FPSubtract(T_{num2}, T_{D5}, n_{exp}, n_{mant})$;
21: $T_{N6} \leftarrow FPMultiply(T_{N5}, T_{R5}, n_{exp}, n_{mant})$;
22: $T_{out} \leftarrow Absolute\_Divide\_By\_Power\_Of\_2(T_{N6}, T_{exp2}, n_{exp}, n_{mant})$;
23: $T_{out} \leftarrow Setbits(T_{in}, 0, 0, 0)$; ▷ Set sign bit as same as input
24: **end procedure**

Finally the division operation can be written as follows.

Algorithm 28 Floating Point Division

procedure FPDIVISION($in : T_1, T_2, n_{exp}, n_{mant}; out : T_{out}$)
$T_{reciprocal} \leftarrow Reciprocal(T_2, n_{exp}, n_{mant})$;
$T_{out} \leftarrow FPMultiply(T_1, T_{reciprocal}, n_{exp}, n_{mant})$;
end procedure

For brevity, this paper did not consider IEEE-754 sub-normals, infinities, NaN and rounding modes. The general-purpose routines provided here make implementing these details straightforward (although too cumbersome for presentation in this paper).

VIII. Experimental Validation

A. Feasibility Analysis

As already has been specified, physical implementations of the proposed algorithms are not difficult, as the basic operations used are easily implementable. To implement $separate$ operation suitably, it is better to design the memory strand as a DNA molecule, but the stickers need to be designed of some alternate backbone material, such as PNA (Peptide Nucleic Acid).

$Combine$ is the easiest of all operations, and can be implemented by simply pouring all the contents of source test tube in destination test tube.

$Separate$ on a particular bit b can be implemented by designing probe out of DNA molecule for that particular bit, and affixing it to a solid support. The nucleotide sequence of the probe will be same as that of the corresponding PNA sticker. During separation, strands having b off, will be captured on the probes, while the strands having b on, will remain in solution, as that region is already covered by corresponding PNA sticker. The solid support can then be washed in zero salt solution, which will detach the DNA probe, but will keep the PNA stickers intact, as decrease in salt concentration strengthens PNA-DNA binding, but weakens DNA-DNA binding.

Set operation can also be implemented easily by taking corresponding PNA sticker of the bit needed to be set in the memory strand solution, and annealing.

Details of the physical implementations of these basic operations can be found in [21].

B. Complexity Analysis

As has already been specified in corresponding sections, logic operations, comparator and all shifters execute in $O(n)$ time. Addition/Subtraction operation also executes in $O(n)$. Multiplication and division algorithm executes in $O(n^2)$ time. As for example, to perform multiplication, for each bit, 4 $O(n)$ operations (one of Add or $Subtract$, one $LogicalRightShift$, one $ArithmeticRightShift$ and one $CircularRightShift$) and several constant time operations need to be perfomed. Similarly floating point operations follow similar execution complexity as on a conventional computer. As an example, at first, both of the comparison of exponents takes linear time in the size of exponent and appending bit at head of mantissa takes linear time in the size of mantissa. In addition algorithm, after appending 1 at head of mantissas and comparing exponents, if the two exponents are found out unequal, needs two more linear integer operations, $viz.$, $LogicalRightShift$ and Add to make the two exponents equal. After that, depending on the sign bit of two numbers, appropriate linear operations are performed on the mantissas and at the end, resultant mantissa is normalized. So, overall the floating point addition is also linear as well as floating point subtraction. Similar logic goes for floating point multiplication, reciprocal and division, which execute in $O(n^2)$ time, but with much higher constant factor than integer operations. So, even though proposed algorithms are no way better than conventional algorithms, but this work have been designed

keeping in mind that, to make DNA Computer generally applicable to solve general problems, where it falls short of conventional computer, a complete ALU is necessary, so that any problem can be solved by not altering the main algorithm much. Where DNA computing is already proven powerful to solve "hard" computational problems requiring searching over large search-space, giving this technology the power of solving general problems will make this computing technique almost complete.

C. Implementation Issues

How much complete a proposed model of ALU is, needs to be measured on the basis of availability of arithmetic and logic operations, the ease of their physical implementations and the range of binary number representable without breaking the stability of DNA molecules. The work presented here has implemented all possible logic, integer and floating point operations. Moreover, the integer operations follow 2's complement arithmetic, and floating point operations follow IEEE 754 floating point format, thus resembling closely to a complete conventional ALU. Also, the physical implementation of the operations are easy.

A strand with 2000 oligonucleotides are considered stable [1] [18]. Each memory segment on the memory strand needs to be unique. Even if we assume, each bit will need 6 bp long sequence, it will allow more than 300 bit long binary numbers to be represented without violating the constraint of memory segment uniqueness, as there can be $4^6 = 4096$ possible different 6 bp long nucleotide sequences.

IX. Conclusions and Future Research

Novel approaches to implement all possible logic, integer and floating point operations have been proposed in this work, resembling closely to a complete conventional ALU. This approach can handle sufficiently large binary numbers. All the biomolecular operations used here are easy to implement.

Although there has been a diminished interest [25] in sticker method due to unreliable nature of DNA operations, [21] and [26] describe methods to work with redundant DNA strands to cope with this. Proposed algorithms can be highly suitable in this context and can be adapted in future research to deal with such redundant representation, as algorithms need to check for whether test tubes are empty or not to take most of the major decisions, and it can easily be performed by checking the concentration of DNA strands in the corresponding test tube.

Proposed techniques are highly suitable for design of automated DNA machine, once the basic operations of sticker DNA model, except $clear$, are automated. As to become a proper substitute of a conventional computer, all these operations are necessary to implement in easily implementable manner, this work is expected to play an important role in the future design of automated DNA machine.

References

[1] L. M. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, vol. 266, pp. 1021–1024, Nov. 1994.

[2] R. J. Lipton, "DNA Solution of Hard Computational Problems," *Science*, vol. 268, pp. 542–545, Apr. 1995.

[3] D. Boneh, C. Dunworth, and R. J. Lipton, "Breaking DES Using a Molecular Computer." DIMACS workshop on DNA computing, 1995.

[4] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber, "DNA Solution of the Maximal Clique Problem," *Science*, vol. 278, pp. 446–449, Oct. 1997.

[5] R. S. Braich, N. Chelyapov, C. Johnson, P. W. K. Rothemund, and L. M. Adleman, "Solution of a 20-Variable 3-SAT Problem on a DNA Computer," *Science*, vol. 296, pp. 499–503, Apr. 2002.

[6] Y. Liu, J. Xu, L. Pan, and S. Wang, "DNA Solution of a Graph Coloring Problem," *J Chem Inf Comput Sci.*, vol. 42, pp. 524–528, May-Jun 2002.

[7] C. A. A. Sanches and N. Y. Soma, "A Polynomial-Time DNA Computing Solution for the Bin-Packing Problem," *Applied Mathematics and Computation*, vol. 215, pp. 2055–2062, 2009.

[8] "Computer Made from DNA and Enzymes," http://news.nationalgeographic.com/news/2003/02/0224_030224_DNAcomputer.html, accessed: 2016-01-25.

[9] Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro, "An Autonomous Molecular Computer for Logical Control of Gene Expression," *Nature*, vol. 429, pp. 423–429, May 2004.

[10] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. LeProust, B. Sipos, and E. Birney, "Towards Practical, High-Capacity, Low-Maintenance Information Storage in Synthesized DNA," *Nature*, vol. 494, pp. 77–80, Feb. 2013.

[11] "Stanford creates biological transistors, the final step towards computers inside living cells," http://www.extremetech.com/extreme/152074-stanford-creates-biological-transistors-the-final-step-towards-computers-inside-living-cells, accessed: 2016-01-25.

[12] F. Guarnieri, M. Fliss, and C. Bancroft, "Making DNA Add," *Science*, vol. 273, pp. 220–223, 1996.

[13] V. Gupta, S. Parthasarathy, and M. J. Zaki, "Arithmetic and Logic Operations with DNA." 3rd DIMACS workshop on DNA computing, 1997.

[14] F. de Santis and G. Iaccarino, "A DNA Arithmetic Logic Unit," *WSEAS Trans. Biomed.*, vol. 1, pp. 436–440, 2004.

[15] M. Ogihara and A. Ray, "Simulating Boolean Circuits on a DNA Computer," in *Proceedings of the First Annual International Conference on Computational Molecular Biology*, ser. RECOMB '97, 1997, pp. 226–231.

[16] M. Amos and P. E. Dunne, "DNA Simulation of Boolean Circuits," Proceedings of 3rd Annual Genetic Programming Conference, Tech. Rep., 1997.

[17] R. Barua and J. Misra, "Binary Arithmetic for DNA Computers," in *Revised Papers from the 8th International Workshop on DNA Based Computers: DNA Computing*, ser. DNA8, 2002, pp. 124–132.

[18] Z. F. Qiu and M. Lu, "Arithmetic and Logic Operations for DNA Computers," in *Proceedings of the Second IASTED International conference on Parallel and Distributed Computing and Networks*, 1998, pp. 481–486.

[19] Z. Ignatova, I. Martínez-Pérez, and K.-H. Zimmermann, *DNA Computing Models*. Springer Science & Business Media, New York, 2008.

[20] P. Guo and H. Zhang, "DNA Implementation of Arithmetic Operations," in *Natural Computation, 2009. ICNC '09. Fifth International Conference on*, vol. 6, Aug 2009, pp. 153–159.

[21] S. Roweis, E. Winfree, R. Burgoyne, N. V. Chelyapov, M. F. Goodman, P. W. K. Rothemund, and L. M. Adleman, "A Sticker Based Model for DNA Computation," *Journal of Computational Biology*, vol. 5, pp. 615–629, 1996.

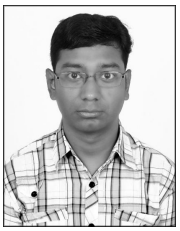
[22] M. G. Arnold, "An improved DNA-sticker Addition Algorithm and its Application to Logarithmic Arithmetic," in *International Workshop on DNA-Based Computers*. Springer, 2011, pp. 34–48.

[23] M. G. Arnold, "Improved DNA-sticker Arithmetic: Tube-Encoded-Carry, Logarithmic Number System and Monte-Carlo Methods," *Natural Computing*, vol. 12, no. 2, pp. 235–246, 2013.

[24] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[25] I. Martínez-Pérez, W. Brandt, M. Wild, and K.-H. Zimmermann, "Bioinspired Parallel Algorithms for Maximum Clique Problem on FPGA Architectures," *Journal of Signal Processing Systems*, vol. 58, no. 2, pp. 117–124, 2010.

[26] M. G. Arnold, "Extending DNA-Sticker Arithmetic to Arbitrary Size Using Staples," in *International Workshop on DNA-Based Computers*. Springer, 2013, pp. 1–15.



Mayukh Sarkar Mayukh Sarkar is currently associated with Indian Institute of Engineering Science and Technology, Shibpur, India as a CSIR (Council of Scientific and Industrial Research, Govt of India) Senior Research Fellow and working on "Optimized Design and Realization of Computing Algorithms on a DNA Computer". He has received his Masters Degree from BESU, Shibpur, India and Bachelors Degree from WBUT, India. His other research interests include Quantum computing, Reversible logic and circuits, Programming techniques, Algorithm analysis and design.

Prasun Ghosal Prasun Ghosal is currently an Assistant Professor in Indian Institute of Engineering Science and Technology, Shibpur, India. He is a Raman Post Doctoral Fellow (Indo-US) and Heidelberg Laureate Post Doctoral Fellow (Germany). His research is in "Performance Centric, Power Aware Nanoscale Electronic System Design and Computing". He has contributed more than 95 research articles, and 14 Book Chapters. He is Young Scientist Research Awardee from ISCA, Best Paper Awardee in IEEE iNIS - 2016, ICAEE - 2014, ADCONS - 2011 etc. He serves as VC, Executive Committee, IEEE CS, TCVLSI, VC, Steering Committee IEEE iNIS etc. More information about him is available at his website: http://www.iiests.ac.in/aboutprasun-ghosal-itmenuitem.

Saraju P. Mohanty Saraju P. Mohanty (SM08) is a Professor at the University of North Texas. Prof. Mohanty's research is in Energy-Efficient Secure Electronic Systems. Prof. Mohanty is an author of 220 research articles, and 3 books. His Google Scholar h-index is 27 and i10-index is 82. He received Society for Technical Communication 2017 Award of Merit. He received 2016 PROSE Award for best Textbook in Physical Sciences & Mathematics. He is the Editor-in-Chief of the IEEE Consumer Electronics Magazine. He serves as the Chair of Technical Committee on VLSI, IEEE Computer Society. More about him is available at: http://www.smohanty.org.