BioSystems 97 (2009) 146–153

BioSystems

journal homepage: www.elsevier.com/locate/biosystems

A novel generalized design methodology and realization of Boolean operations using DNA

B.S.E. Zoraida[a,*], Michael Arock[a], B.S.M. Ronald[b], R. Ponalagusamy[c]

[a] Department of Computer Applications, National Institute of Technology, Trichy, Tamilnadu, 620 015, India
[b] Vaccine Research Centre, Bacterial Vaccine, TANUVAS, Chennai, Tamilnadu, 600 051, India
[c] Department of Mathematics, National Institute of Technology, Trichy, Tamilnadu, 620 015, India

ARTICLE INFO

Article history:
Received 5 November 2008
Received in revised form 27 May 2009
Accepted 27 May 2009

Keywords:
DNA computing
Molecular beacon
Adder
Subtractor
Logic gates
Boolean circuit

ABSTRACT

The biological deoxyribonucleic acid (DNA) strand has been increasingly seen as a promising computing unit. A new algorithm is formulated in this paper to design any DNA Boolean operator with molecular beacons (MBs) as its input. Boolean operators realized using the proposed design methodology is presented. The developed operators adopt a uniform representation for logical 0 and 1 for any Boolean operator. The Boolean operators designed in this work employ only a hybridization operation at each stage. Further, this paper for the first time brings out the realization of a binary adder and subtractor using molecular beacons. Simulation results of the DNA-based binary adder and subtractor are given to validate the design.

1. Introduction

Ever since Adleman (1994) has published a paper on molecular computation for solving Hamiltonian path problem (HPP), attempts are being made to utilize DNA manipulations for solving computationally difficult problems. Several models of computation using DNA have been proposed earlier and they are Turing machine (Rothemund, 1996), Sticker model (Roweis et al., 1998), Splicing systems (Erk, 1999), Surface-based computing (Wang et al., 1999; Liu et al., 2000; Su and Smith, 2004) and Boolean circuits (Ogihara and Ray, 1998, 1999).

The inherent parallelism in DNA was utilized before by many researchers in constructing Boolean operators. Amos and Dunne (1997) described the simulation of a bounded fan-in Boolean circuit with NAND gate, which takes time proportional to the depth of the circuit for computation. Ogihara and Ray (1998), proposed a bounded fan-in Boolean circuit functioning in O(1)-time complexity. Ogihara and Ray (1999) also proposed the building of DNA-based Boolean circuits for a semi-unbounded fan-in Boolean circuit. Subsequently, Erk (1999), developed an abstract DNA model for simulating Boolean circuits by finite splicing systems. The main drawback of this model is that the rules need to be altered with the complexity of the Boolean circuit. Mulawka et al. (1999) proposed another simulation of the NAND gate using the Fok I enzymes of nuclease class II. Ahrabian and Nowzari-Dalini (2004) and Ahrabian et al. (2005) proposed a different construction of NAND gate and the same authors presented a DNA algorithm for solving an unbounded fan-in Boolean circuit in O(1)-time complexity. Liu et al. (2005) presented a theoretical model of the NAND gate through the induced hairpin formation. Jianzhong et al. (2006) suggested reusable logic gates for AND and OR functions using MB.

The major demerit of the previous models is that they employ many different types of bio-operations like annealing, ligation, polymerase chain reaction, gel electrophoresis and cleavage. Amos (2005) in his book has stated that it is difficult to ensure 100% ligation and hence strands that should have been restricted escape into the next stage and thereby resulting in errors in computation. Further, the above papers were proposed for simulating few gates only and they do not maintain the uniformity in representing logical 0 and 1, either. The previous authors had also tried simulating stratified Boolean circuits which required the gates to be of the same type at each stage (i.e. either AND or OR gate). Furthermore, in earlier implementations, after obtaining the outputs, the strands employed cannot be reused and they have to be necessarily reconstructed for subsequent use. It should also be noted that a unique generalized algorithm for designing any combinational logic operators has not been formulated in the above articles. In this context, the authors of this work (Zoraida et al., 2008) had earlier proposed a generalized algorithm for constructing logic gates. The proposed realization of a logic operator is attempted with only a

* Corresponding author. Tel.: +91 9443647737; fax: +91 431 2500657.
E-mail addresses: b.s.e.zoraida@gmail.com (B.S.E. Zoraida), michael@nitt.edu (M. Arock), romasa68@yahoo.com (B.S.M. Ronald), rpalagu@nitt.edu (R. Ponalagusamy).

0303-2647/$ – see front matter © 2009 Elsevier Ireland Ltd. All rights reserved.
doi:10.1016/j.biosystems.2009.05.010

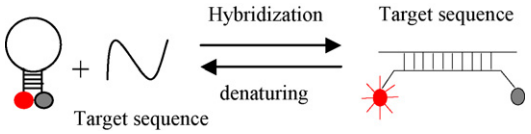

Fig. 1. Hybridization of the MB with the target.

single bio-operation, namely hybridization at each stage. However, the algorithm does not address the realization of combinational logic operators and this paper proposes a generalized algorithm for constructing both logic operators and combinational logic operators such as adders and subtractors using DNA with MB as inputs. The proposed model has uniform representation for logical 0 and 1 throughout. MBs are employed, so that the bio-operations are reliable. MBs also have extremely high selectivity and ability to identify single base-pair mismatch (Fang et al., 1999). The DNA sequence which acts as gate strand is made immobilized onto a surface. In the proposed method, reusability is achieved by a denaturing operation followed by a wash, which leaves back the strands on the surface for subsequent operations.

2. Preliminaries

2.1. Molecular Beacon

Molecular beacons are single-stranded oligonucleotide hybridization probes that possess a stem and a loop structure. The loop has a complementary probe sequence of a target sequence. The stem is formed by annealing with the complementary sequence present in either side of the probe sequence. A fluorophore and a quencher are linked to the two ends of the stem. The two moieties are kept in close proximity to each other by the stem, enabling fluorescence of the fluorophore to be quenched through energy transfer and at this point the MB is "dark". When the MB encounters its target DNA molecule, it undergoes a spontaneous conformational reorganization that forces the stem apart so that the fluorophore and quencher moves away. So there is a transition from "dark" to fluorescence "bright" in MB. This is known as fluorescence resonance energy transfer (FRET). Molecular recognition specificity is one of the major advantages of MB. They are highly target-specific to the extent that they ignore target sequences that differ even by a single nucleotide. Since 4-(dimethylaminoazo) benzene-4-carboxylic acid (DABCYL) can serve as a universal quencher for many fluorophores, a MB is generally synthesized using DABCYL-controlled pore glass (CPG) as the starting material. Different fluorescent dye molecules can be covalently linked to the 5′-end to report fluorescence at different wavelengths (Tyagi and Kramer, 1996). The hybridization of the MB with the target is shown in Fig. 1.

2.2. Blocker

The hybridization of a MB to the target strand is necessary for implementing combinational binary operators. However, in certain location of the target sequence, hybridization should be prevented and hence, a DNA sequence is needed as a blocker. In this paper, the assigned blocker sequence is "GGGGGG". The sequence assigned for each input strand should be different from the blocker sequence. The blocker is denoted by a circle and an "X" through it in the subsequent figures.

3. Proposed Algorithm for Realization of Boolean Operators

It is well known that each binary Boolean variable I can take two values $I = 0$ and $I = 1$. In this work they are denoted as I^0 and I^1. A

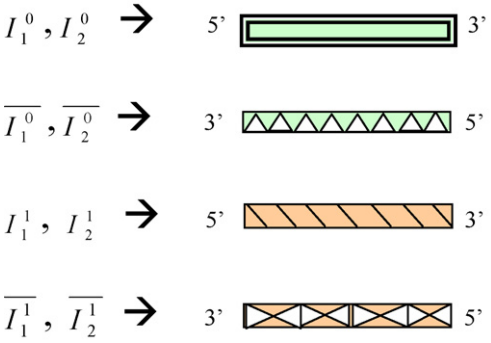

Fig. 2. Colored pattern representation of the strand.

two-input logic gate has their inputs denoted as I_1 and I_2. A unique DNA strand is taken to represent I_1^0 and I_2^0 (logic 0) and another unique strand is chosen to represent I_1^1 and I_2^1 (logic 1). The respective complements of the chosen strands are represented as $\overline{I_1^0}$, $\overline{I_2^0}$, $\overline{I_1^1}$ and $\overline{I_2^1}$. Thus, only four DNA strands are required for a two-input gate. A multiple pattern representation instead of representing the actual strands for $I_1^0, I_2^0, I_1^1, I_2^1, \overline{I_1^0}, \overline{I_2^0}, \overline{I_1^1}$ and $\overline{I_2^1}$ is followed in this paper for visualizing the realization. Fig. 2 shows the patterns and logical inputs and their complements.

A general form of truth table of a two-input gate is shown in Table 1, where I_1 and I_2 are the inputs and R_1 to R_4 are the outputs of the gate.

In the proposed design algorithm, the rows having their output $R_i = 1$ ($i = 1$–4) are considered initially. Among these rows, the first row having $R_i = 1$ is taken and its I_1 and I_2 values are stored in an array. Subsequently, the next row with $R_i = 1$ is considered and its I_1 value is compared with the last value stored in the array. If I_1 value is the same as the last value in the array, I_2 of the row presently considered is added to the array. Otherwise, "*" followed by both the values I_1 and I_2 of the present row is added to the array. The above process is repeated for the remaining rows having their R_i value as 1. After scanning all the rows with $R_i = 1$ in the truth table, the elements in the array are replaced by their corresponding complementary strands which were assigned at the start of the process. The symbol "*" is replaced by the blocker sequence. Consequently, the desired strand which performs the operation of the required Boolean circuit is obtained.

The above process is described as a unique procedure "*Strand_construct*" for designing the required strands. "*Strand_construct*" is the procedure with the following input parameters: (a) truth table of the desired Boolean circuit with the Boolean inputs I_i, $i = 1, 2, \ldots, n$ and R_i, $i = 1$ to 2^n as the Boolean outputs, (b) the number of inputs to the Boolean operator – n, which is assigned to the variable *in_no*, (c) *start* and *finish* variables help to scan the truth table in either of the direction, i.e. top to bottom or bottom to top, by assigning values 1 or 2^n, respectively, (d) variable *step_val* is assigned with -1, if *start* takes 2^n (when scanned from bottom to top), otherwise *step_val* is given a value 1.

Table 1
Truth table for two-input gates.

I_1	I_2	R_i
0	0	R_1
0	1	R_2
1	0	R_3
1	1	R_4

The pseudocode for the procedure is given below:

```
Strand_construct(truth_table,in_no,start,finish,step_val)
{
  is_first_time ← TRUE, index ← 0
  for i ← start to finish step step_val do {
      If (R_i = 1) and (is_first_time = TRUE){
            for j ← 1 to in_no do
            strand_padd[++index] ← I_{i,j};
            is_first_time ← FALSE;
            next_iteration;      // similar to continue in C
           }
      If (R_i =1) {  // for the rest of the time
            check_flag ← TRUE;
            tempindex ← index
          for c ← in_no-1 to 1 step -1 do
          {
          if (strand_padd[tempindex] ≠ I_{i,c})
          check_flag ← FALSE;
          tempindex--;
              }
              If (check_flag = TRUE)
              strand_padd[++index] ← I_{i,in_no};
              else {strand_padd[++index] ← "*";
              for j ← 1 to in_no do
              strand_padd[++index] ← I_{i, in_no};
                 }
        } // for loop completed
  } //End Procedure
```

The final output corresponding to the required DNA strand will be in the array "*strand_padd*". It is evident that the array will have values 0, 1 or "*". The required strand is then synthesized by replacing the values 0s and 1s in "*strand_padd*" with the assigned complementary strand and "*" is replaced with the blocker sequence. Thus, the desired DNA strand for the Boolean circuit is obtained.

The utility of the design algorithm is illustrated with few examples in the following section.

4. Realization of Logic Gates

4.1. Case 1

Consider the truth table of the EX-OR gate in Table 2.

The second row has the R_2 value as 1 and hence (0, 1) is added to the array. The last value in the array is 1 which is compared with the I_1 (=1) value of the third row which has R_3 (=1). As the values are same, I_2 (=0) is added to the array and now the array has (0, 1, 0). In this case, fourth row is not compared because R_4 (=0). After replacing the 0s and 1s of the array with corresponding complementary strands, the gate strand for EX-OR gate is obtained as shown in Fig. 3.

4.2. Case 2

Consider the truth table of the EX-NOR gate in Table 3.

Since the first row has R_1 value as 1, (0,0) is stored in the array. The last value in the array which is 0 is compared with I_1 (=1) value of the fourth row which has $R_4 = 1$. Since the values are not the same, "*" is added and this is followed by I_1 and I_2 values of the fourth row. Now, the array has (0,0,*,1,1). After replacing the 0s and 1s with the corresponding complementary strands and "*" with the blocker sequence, the required EX-NOR gate strand is obtained as in Fig. 4.

The developed procedure "*Strand_construct*" can be used to generate DNA strand for any logic gate. It should be noted that, the input parameters, *start* and *finish* take the value 1 and 4 for all gates except OR gate. In the case of OR gate, the parameter *start* gets 4 and *finish* gets 1.

Table 2
Truth table of the EX-OR gate.

I_1	I_2	R
0	0	0
0	1	1
1	0	1
1	1	0

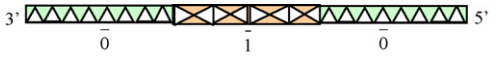

Fig. 3. Gate strand for EX-OR gate.

Table 3
Truth table of the EX-NOR gate.

I_1	I_2	R
0	0	1
0	1	0
1	0	0
1	1	1

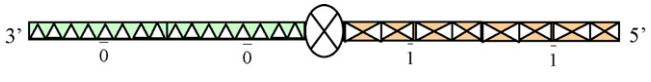

Fig. 4. Gate strand for EX-NOR gate.

Table 4
Gate strands for logic gates.

Gate	Mathematical representation	Sequence of strands which acts as gate strand
OR	$R(I_1, I_2) = \begin{cases} 1, & \text{if } I_1 \neq I_2 \\ I_1, & \text{otherwise} \end{cases}$	
AND	$R(I_1, I_2) = \begin{cases} I_1, & \text{if } I_1 = I_2 \\ 0, & \text{otherwise} \end{cases}$	
NAND	$R(I_1, I_2) = \begin{cases} \neg I_1, & \text{if } I_1 = I_2 \\ 1, & \text{otherwise} \end{cases}$	
NOR	$R(I_1, I_2) = \begin{cases} \neg I_1, & \text{if } I_1 = I_2 \\ 0, & \text{otherwise} \end{cases}$	
EX-OR	$R(I_1, I_2) = \begin{cases} 0, & \text{if } I_1 = I_2 \\ 1, & \text{otherwise} \end{cases}$	
EX-NOR	$R(I_1, I_2) = \begin{cases} 1, & \text{if } I_1 = I_2 \\ 0, & \text{otherwise} \end{cases}$	
NOT	$R(I_1) = \neg I_1$	
BUFFER	$R(I_1) = I_1$	

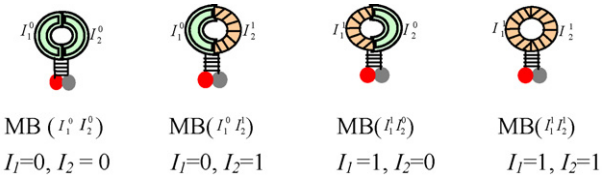

Fig. 5. MB inputs of gates.

Table 5
Truth table of half adder.

I_1	I_2	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

4.3. Inputs of the Logic Gates

Gate strand that has been obtained from executing the above procedure "*Strand_construct*" will be fixed on the surface at the 3′end (Liu et al., 2000). The inputs for these gates are the MBs. The loop portion of the MB has two sequences representing the inputs of the gate I_1 and I_2. The two sequences are the assigned input strands for the input variable of the logic gate. The designated MB inputs of a gate having inputs I_1 and I_2 (shown in Fig. 2) are presented in Fig. 5. Changes that occurred after introducing the MBs will represent the output of the gate. In this simulation, the fluorescence always represents 1 and the dark represents 0.

As a matter of fact, logical operators such as NOT and BUFFER gates have only a single input and a single output. Correspondingly, in DNA realization, the loop portion of the MB has only one sequence, i.e. the input I_1 (where I_1 can be either I_1^0 or I_1^1) of the operator. The DNA strands for various logic gates obtained using the procedure "*Strand_construct*" proposed in Section 3 are given in Table 4. In addition to the construction of logic gates, the proposed procedure "*Strand_construct*" can be employed to construct any combinational logic operators. The design of adder and subtractor using the proposed procedure is presented in the next section.

5. Development of Arithmetic Operators

Arithmetic operators using DNA are constructed by giving the truth table as input to the "*Strand_construct*" procedure. The strand produced by the procedure will be the required arithmetic operator. The inputs for the obtained operator are the MBs, where the loop portions of the MBs have the corresponding DNA sequences. The arithmetic operator strand obtained is then fixed on a surface and the input MBs are given to the surface. If a MB has any complement sequence present in the arithmetic strand it gets hybridized and it gives a fluorescent light. The fluorescent light indicates a logical "1"; otherwise, it is logical "0". The realization and simulation of adders and subtractors are as follows:

5.1. Half Adder Realization

Half adder adds two single-bit binary digits to produce two bits as output, one bit for the Sum and other bit for the Carry. The truth table of the half adder is given in Table 5. This can be realized using the XOR and AND gate strands obtained using the procedure "*Strand_construct*". The half adder strand for an 1-bit adder is given in Fig. 6. The inputs for half adder are two MBs with the same input sequences, one for the sum strand and the other for the carry strand. In this case, the loop portion of the MB has the two sequences I_1 and I_2.

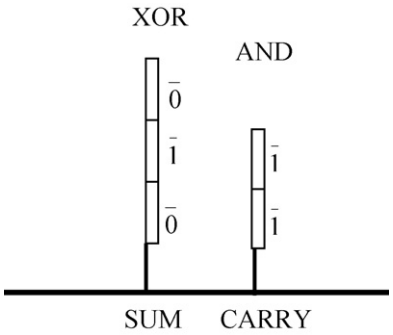

Fig. 6. Half adder strand for one-bit.

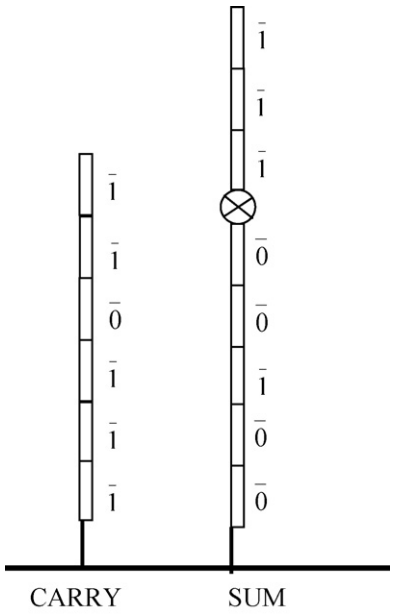

Fig. 7. Strands for one-bit full adder.

5.2. Full Adder Realization

Full adder adds three single-bit binary digits to produce two bits as output, one for the sum and the other for the carry. In the construction of adder and carry strand, the proposed method assigns only one strand to represent logical 0 for I_1^0, I_2^0, and I_3^0 and another strand to represent logical 1 for I_1^1, I_2^1, and I_3^1. The respective complements of the chosen strand are $\overline{I_1^0}$, $\overline{I_2^0}$, $\overline{I_3^0}$, $\overline{I_1^1}$, $\overline{I_2^1}$, and $\overline{I_3^1}$. The sum strand and carry strand are obtained from the "*Strand_construct*" procedure separately. The input parameters for the procedure "*Strand_construct*" (given in Section 3) to generate the sum strand are (a) the truth table of the full adder having the Boolean inputs I_i, $i = 1–3$ and the output sum S_i, $i = 1–8$, (b) 1 is assigned to *start* and (c) 8 is assigned to *finish*. Similarly, to generate the carry strand the input parameters are (a) Boolean inputs I_j, $j = 1–3$ and the output carry C_i, $i = 1–8$, (b) *start* is assigned 8 and (c) *finish* is assigned 1 and 3 is assigned to *in_no* for constructing both the above strands. The sum and carry strands obtained by the procedure "*Strand_construct*" for single-bit binary full adder are shown in Fig. 7. These strands also require a single bio-operation hybridization to get the sum and the carry. The truth table for the full adder is given in Table 6. The inputs for the full adder strands are two MBs representing same input, one for the sum and the other for the carry strand. In this example, the loop portion of the MB has three sequences I_1, I_2, I_3 representing the three-digit inputs, whereas for a two input truth table such as a two-input logic gates, the loop has only two sequences I_1 and I_2.It is evident that *in silico* as shown in Fig. 8, two EX-OR gates are required for producing the Sum. Similarly, an EX-OR gate, two AND gates and one OR gate are necessary for producing the Carry. Thus, C_i output is obtained by the Boolean operation $((A_i \oplus B_i)\cdot C_{i-1}) + (A_i \cdot B_i)$ in digital implementation. On the other hand, a single DNA strand shown in Fig. 7 is sufficient to give the required output C_i. The inherent parallelism exhibited by DNA computing has thus been successfully exploited in the proposed design methodology. It can be seen that in this approach only two DNA strands are required for implementing 1-bit full adder as shown in Fig. 7 as compared to five logic gates (Fig. 8) for the implementation of the same operator *in silico*.

Table 6
Truth table of full adder.

i	I_1	I_2	I_3	Sum (S_i)	Carry (C_i)
1	0	0	0	0	0
2	0	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0
6	1	0	1	0	1
7	1	1	0	0	1
8	1	1	1	1	1

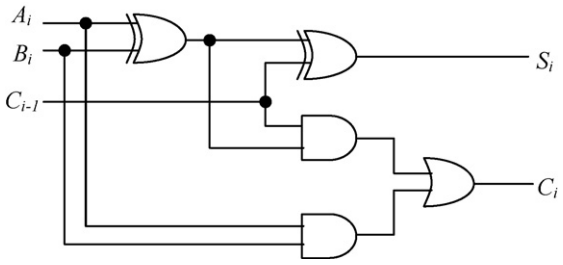

Fig. 8. Full adder circuit employing silicon gates.

5.3. Realization of Four-Bit Carry Ripple Adder

The four-bit carry ripple DNA adder is shown in Fig. 9 with two four-bit binary numbers, A_3, A_2, A_1, A_0 and B_3, B_2, B_1, B_0 as its inputs. Evidently, one half adder for the least significant bit and three full adders for the remaining binary bits are needed to add these two four-bit binary numbers. Here again, the input to the half and full adder are the MBs. At each stage, two sets of inputs are given one for the carry and the other for the sum strand. The half adder input MB has two sequences A_0 and B_0 in the loop, while the full adder input MB has three sequences A_i, B_i, C_{i-1}; $i = 1–3$, where C_{i-1} is the carry from the previous bit. The half adder and full adder strands are fixed on the surface initially. Then, inputs to the least significant bit are given. The sum is recorded and the output of the carry strand is given to the input MBs of the next bit. The fluorescent light represents logical 1 and dark represents logical 0 in this realization also. The process is repeated until the most significant bit is reached.

5.4. Simulation of Four-Bit Carry Ripple Adder

Let the two numbers *A* and *B* take the binary numbers 0110 and 1111, respectively. The half adder and full adder Boolean operator strands are fixed on the surface. The two MBs having the input sequence A_0, B_0 (0,1) in the loop, corresponding to the sum and carry strands are given as inputs to the stage-1. One of the two MBs get attached to the sum strand and gives a fluorescent light representing logical 1 (since the complement sequence ($\bar{0}$, $\bar{1}$) is present in the sum strand). The other MB cannot get attached to the carry strand and does not give a fluorescent light representing logical 0 (since complement sequence ($\bar{0}$, $\bar{1}$) is not present in the carry strand). This logical 0 (C_0) from the carry strand of stage-1 is given as carry to stage-2. Therefore, for stage-2 the inputs are two MBs having the input sequence A_1, B_1, C_0 (1,1,0) in the loop. The process is repeated for the remaining stages. Table 7 gives the four cycles corresponding to the four stages of the adder.

In the proposed design, logical 0 and 1 will be sensed by light sensors based on photoreceptive polypeptides (Pieroni et al., 2001; Ferguson et al., 1996). After sensing the state of the previous stage, appropriate MB will be chosen as input to the next stage by molecular motors. The possibility of manufacturing such molecular motors fueled by light and chemical energy has been attempted earlier (Schalley et al., 2001). A detailed description of the mechanical

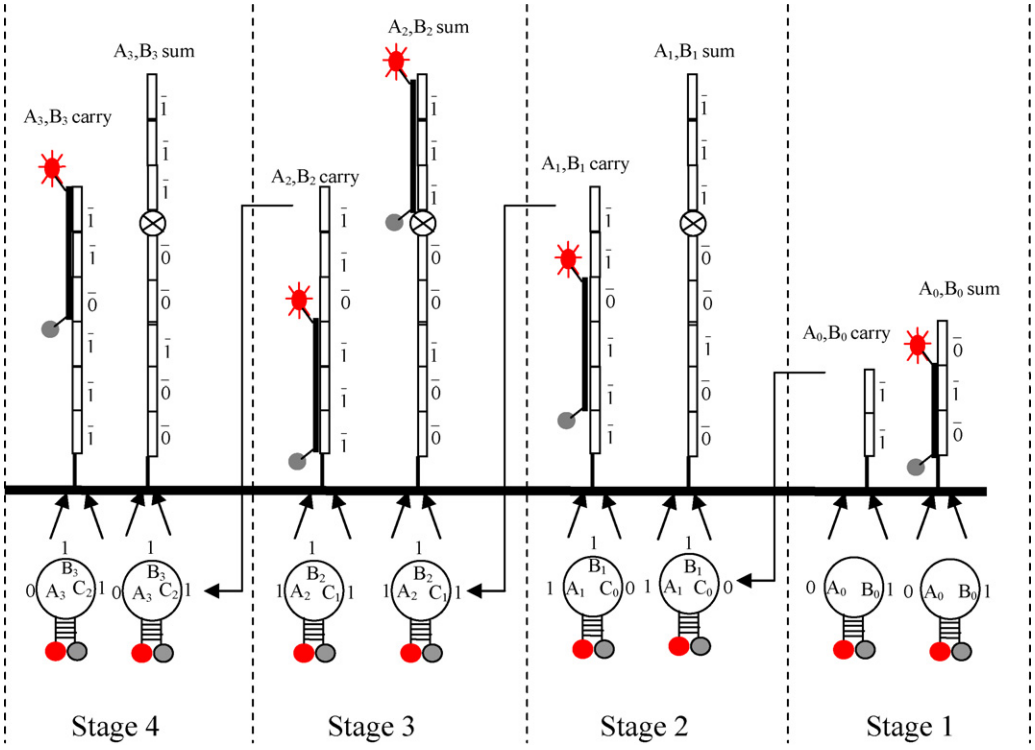

Fig. 9. Four-bit carry ripple DNA adder.

Table 7
Progress cycle for adder.

Cycle number	Input MBs	Carry	Sum
First	Two MB (A_0^0, B_0^1).	0	1
Second	Two MB (A_1^1, B_1^1, C_0^0).	1	0
Third	Two MB (A_2^1, B_2^1, C_1^1).	1	1
Fourth	Two MB (A_3^0, B_3^1, C_2^1)	1	0

arrangement for propagation of logical state from one stage to another is beyond the scope of the present work.

5.5. Half Subtractor Realization

Half subtractor subtracts two binary digits and gives two bits as output, one is the difference bit and the other is the borrow bit. The truth table of the half subtractor is given in Table 8. Following through the design procedure of "*Strand_construct*" described earlier, the XOR and Borrow strands are obtained as shown in Fig. 10. The inputs for half subtractor are two similar MBs, one for the difference and the other for borrow. The loop portion of the MB has the two sequences I_1 and I_2.

5.6. Full Subtractor Realization

Subtraction between two bits is carried out by taking into account the borrow from the previous stage. A full subtractor has three inputs and two outputs. The three inputs, I_1, I_2 and I_3 denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and W, represent the difference and borrow, respectively. In order to generate the difference strand, the "*Strand_construct*" procedure takes the following inputs (a) the truth table of the full subtractor having the Boolean inputs I_j, $j = 1–3$ and the output difference D_i, $i = 1–8$, (b) *start* takes the value 1 and (c) *finish* takes 8. Similarly, for the borrow strand, the truth table of the full subtractor shown in Table 9 has been employed again, with inputs (a) I_j, $j = 1–3$ and the output borrow W_i, $i = 1–8$, (b) *start* gets 1 and (c) *finish* gets 8. In constructing both the strands, *in_no* is

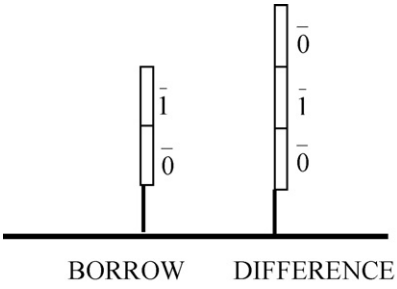

Fig. 10. Strands for one-bit half subtractor.

Table 8
Truth table of half subtractor.

I_1	I_2	Difference (D)	Borrow (W)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Table 9
Truth table of the full subtractor.

i	I_1	I_2	I_3	Difference (D)	Borrow (W)
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	0	1	1
4	0	1	1	0	1
5	1	0	0	1	0
6	1	0	1	0	0
7	1	1	0	0	0
8	1	1	1	1	1

assigned 3 and the full subtractor's difference and borrow strand are obtained as shown in Fig. 11. Thus, the strands obtained by the procedure "*Strand_construct*" for difference and borrow for single-bit binary digit employ single bio-operation, "hybridization". As in the case of full adder, the inherent parallelism of DNA computing is taken advantage of in the design of full subtractor also. Here again, the fluorescent represents logical 1 and dark represents logical 0.

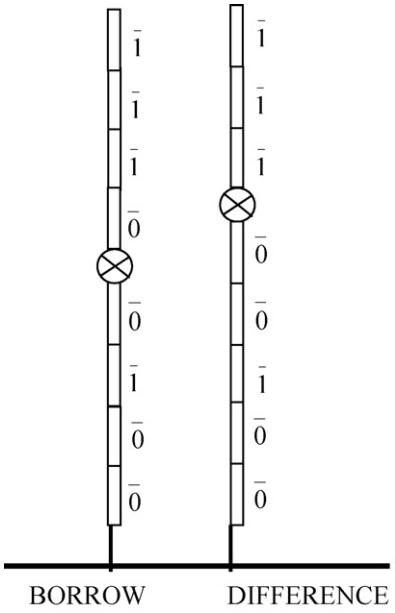

Fig. 11. Strands for one-bit full subtractor.

5.7. Realization of Four-Bit Borrow Ripple Subtractor

Subtraction of 2 four-bit binary numbers A_3, A_2, A_1, A_0 and B_3, B_2, B_1, B_0 requires one half subtractor for the least significant bit and three full subtractors for the remaining binary bits. The input MB of the half subtractor has two sequences A_0 and B_0 in the loop, while the full subtractor input MB has three sequences A_i, B_i, W_{i-1}; $i = 1$–3, in the loop, where W_{i-1} is the borrow from the previous bit. The half subtractor and full subtractor strands are fixed on the surface initially. The difference is recorded and the output of the borrow strand is given to the input MBs of the next bit. The process is repeated until the most significant bit is reached as shown in Fig. 12. The DNA realization of a four-bit borrow ripple DNA subtractor is shown in Fig. 13.

5.8. Simulation of four-Bit Borrow Ripple Subtractor

The operation of the subtractor is illustrated with two numbers $A = 1010$ and $B = 0111$ taken as input. Four cycles are required for its

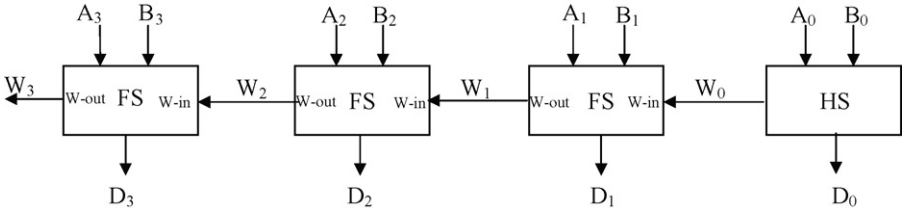

Fig. 12. Four-bit borrow ripple subtractor.

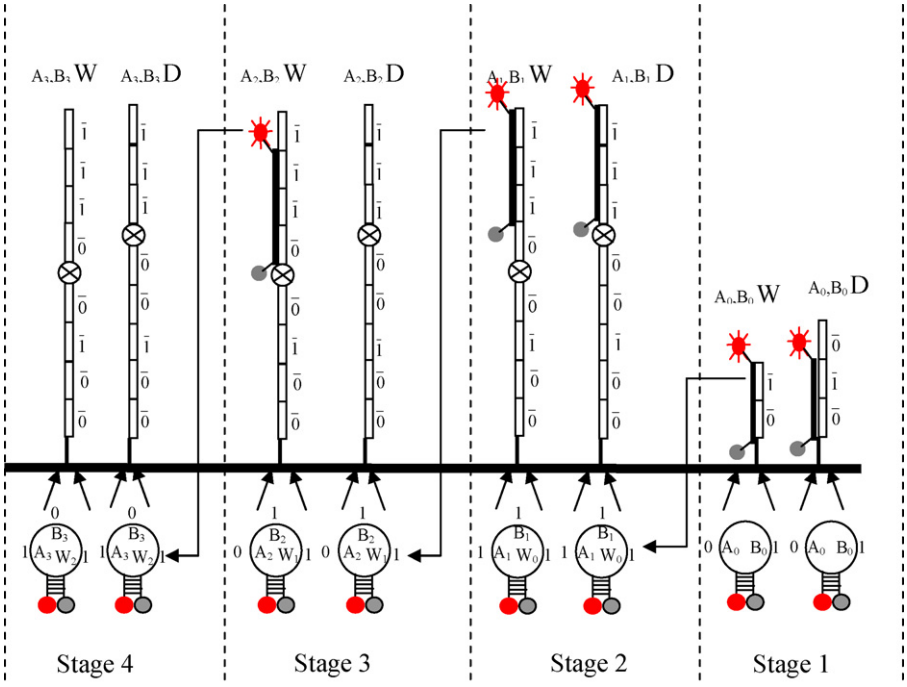

Fig. 13. Four-bit borrow ripple DNA subtractor.

Table 10
Progress cycle for subtractor.

Cycle number	Input MB	Borrow (W)	Difference (D)
First	Two MB (A_0^0, B_0^1)	1	1
Second	Two MB (A_1^1, B_1^1, W_0^1)	1	1
Third	Two MB (A_2^0, B_2^1, W_1^1)	1	0
Fourth	MB (A_3^1, B_3^0, W_2^1)	0	0

operation since there are four bits. As explained in Section 5.4, the process is carried out in each stage. Table 10 gives the four cycles corresponding to the four stages of the subtractor for the above input.

6. Conclusion

A Boolean circuit model using molecular beacons has been established. A unique generalized algorithm for forming any DNA-based combinational Boolean operator is formulated in this paper. The proposed algorithm for the design of a DNA Boolean operator has the same application as that of Karnaugh's map in digital circuit design *in silico*. The developed design algorithm gives DNA strands for Boolean operators that can be easily implemented using molecular beacons. Compared to earlier models, the present work needs only one bio-operation to complete the computation at each stage. Further, Boolean operators obtained using the proposed method is reusable and reliable. Furthermore, the realization of one-bit full adder and one-bit full subtractor circuits *in silico* requires at least five logic gates (Fig. 8). On the other hand, it has been brought out in this paper that only two DNA strands are sufficient for implementing the same. Thus, the inherent parallelism in DNA computing has been effectively exploited in the proposed design algorithm. The proposed design maintains uniformity in representing logical 1 and logical 0 for any Boolean operator. The proposed algorithm can be employed to design arithmetic operators and simple logic operators for traffic control signaling, elevator operation, etc. Nonetheless, the number of inputs to the logic operators is limited by the number of DNA sequence that can be made available in the loop of the MB.

References

Adleman, L., 1994. Molecular computation of solutions to combinatorial problems. Science 266, 1021–1029.

Ahrabian, H., Ganjtabesh, M., Nowzari-Dalini, A., 2005. DNA algorithm for an unbounded fan-in Boolean circuit. Biosystems 82, 52–60.

Ahrabian, H., Nowzari-Dalini, A., 2004. DNA simulation of NAND circuits. AMO-Advanced Modeling and Optimization 6, 2.

Amos M., Dunne, P., 1997. DNA simulation of Boolean circuits. Technical Report CTAC-97009. Department of Computer Science, University of Liverpool.

Amos, M., 2005. Theoretical and experimental DNA computation. In: Natural Computing Series. Springer-Verlag, Berlin, Heidelberg, pp. 85–87.

Erk, K., 1999. Simulating Boolean circuits by finite splicing. In: Proceedings of the Congress on Evolutionary Computation, vol. 2. IEEE Press, pp. 1279–1285.

Fang, X., Liu, X., Schuster, S., Tan, W., 1999. Designing a novel molecular beacon for surface-immobilized DNA hybridization studies. Journal of the American chemical Society 121 (12), 2921–2922.

Ferguson, J.A., Boles, T.C., Adams, C.P., Walt, D.R., 1996. A fiber-optic DNA biosensor microarray for the analysis of gene expression. Nature Biotechnology 14, 1681–1684.

Jianzhong, C., Zhixiang, Y., Wei, W., XiaoHong, S., Linqiang, P., 2006. Towards reliable simulation of bounded fan-in Boolean circuits using molecular beacon. In: Proceedings of the World Congress on Intelligent Control and Automation, IEEE, Dalian, China, pp. 3910–3914.

Liu, Q., Wang, L., Frutos, A.G., Condon, A.E., Corn, R.M., Smith, L.M., 2000. DNA computing on surfaces. Nature 403, 175–179.

Liu, W., Shi, X., Zhang, S., Liu, X., Xu, J., 2005. A new DNA computing model for the NAND gate based on induced hairpin formation. Biosystems 77, 92–97.

Mulawka, J.J., Wasiewicz, P., Plucienniczak, A., 1999. Another logical molecular NAND gate system. In: Proceedings of the Seventh International Conference on microelectronics for Neural, Fuzzy and Bio-Inspired Systems, Granada, Spain, pp. 340–346.

Ogihara, M., Ray, A., 1998. DNA-based self-propagating algorithm for solving bounded-fan-in Boolean circuit. In: Proceedings of the Third Conference on Genetic Programming. Morgan Kaufman Publisher, San Francisco, pp. 725–730.

Ogihara, M., Ray, A., 1999. Simulating Boolean circuits on a DNA computers. Algorithmica 25, 239–250.

Pieroni, O., Fissi, A., Angelini, N., Lenci, F., 2001. Photoresponsive polypeptides. Accounts of Chemical Research 34, 9–17.

Rothemund, P., 1996. A DNA and restriction enzyme implementation of Turing machines. In: DIMACS Series, Proceedings of a DIMAC Workshop, AMS 27, pp. 75–119.

Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N., Goodman, M., Rothemund, P., Adleman, L., 1998. A sticker based model for DNA computation. Journal of Computational Biology 5 (4), 615–629.

Schalley, C.A., Beizai, K., Vogtle, F., 2001. On the way to rotaxane-based molecular motors: studies in molecular mobility and topological chirality. Accounts of Chemical Research 34, 465–476.

Su, X., Smith, L.M., 2004. Demonstration of a universal surface DNA computer. Nucleic Acids Research 32 (10), 3115–3123.

Tyagi, S., Kramer, F.R., 1996. Molecular beacon: probes that fluorescence upon hybridization. Nature Biotechnology 14, 303–308.

Wang, L., Liu, Q., Frutos, A., Gillmor, S., Thiel, A., Strother, T., Condon, A., Corn, R., Lagally, M., Smith, L., 1999. Surface-based DNA computing operations: destroy and readout. Biosystems 52 (1), 189–191.

Zoraida, B.S.E., Arock, M., Ronald, B.S.M., Ponalagusamy, R., 2008. A novel generalized model for constructing reusable and reliable logic gates using DNA. In: Proceedings of the Fourth International Conference on Natural Computing, vol. 7. IEEE Press, China, pp. 353–357.