Parallel Pairwise Operations on Data Stored in DNA: Sorting, XOR, Shifting, and Searching

Arnav Solanki[1], Tonglin Chen[1] and Marc Riedel[1*]

[1]Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, 55454, Minnesota, United States.

*Corresponding author(s). E-mail(s): mriedel@umn.edu; Contributing authors: solan053@umn.edu; chen5202@umn.edu;

Abstract

Prior research has introduced the Single-Instruction-Multiple-Data paradigm for DNA computing (SIMD DNA). It offers the potential for storing information and performing in-memory computations on DNA, with massive parallelism. This paper introduces three new SIMD DNA operations: sorting, shifting, and searching. Each is a fundamental operation in computer science. Our implementations demonstrate the effectiveness of parallel pairwise operations with this new paradigm.

Keywords: Molecular Computing, DNA Computing, DNA Storage, Parallel Computing, Strand Displacement

1 Introduction

Beginning with the seminal work of Adelman a quarter-century ago [1], DNA computing has promised the benefits of massive parallelism in operations. More recently, there has been considerable interest in DNA storage [2, 3]. A particularly promising approach is to encode data by “nicking” DNA with editing enzymes such as PfAgo and CRISPR-Cas9 [4, 5]. A novel paradigm that combines this form of data storage with computation, dubbed “SIMD DNA”, was introduced in 2019 [6]. Data is stored on potentially long DNA strands, divided into “cells”, each storing a single bit. Nicks and denaturing create open toeholds in each cell. Toehold-mediated strand displacement [7, 8] is used to implement computation on the stored values.

This paper proposes an encoding system for SIMD DNA computation, suitable for general pairwise operations. We had previously presented 3 novel applications using this encoding system [9]. The first was a binary bubble sorting algorithm (equivalent to rule 184 with elementary cellular automata [10, 11]). We showed that sorting could be performed in only N parallel steps, where N was the number of bits to be sorted. The second application was a left-shifting operation (equivalent to rule 170 with elementary cellular automata), performed in a single parallel step. The third application was a parallel search algorithm that checked if a query substring was present in a target string. In principle, the algorithm could return an answer in $\log(n)$ steps, but our implementation required between $\log(n)$ and n steps to complete, where n was the length of the query string. This paper expands upon this encoding system with a new application, a parallel Exclusive OR calculation. This XOR operation requires at most N steps to compute the XOR of N bits. All 4 of these applications are of immediate practical interest, as many forms of computation on stored data entail some form of sorting, XOR, shifting, and searching.

2 Background

2.1 Parallel computation using SIMD

SIMD is a computer engineering acronym for Single Instruction, Multiple Data [12], a form of computation in which multiple processing elements perform the same operation on multiple data points simultaneously. It contrasts with the more general class of parallel computation called MIMD (Multiple Instructions, Multiple Data), where multiple processing elements can perform completely different operations on multiple data points simultaneously. While general MIMD parallelism might be desirable, it is often not practical. Much of the modern progress in electronic computing power has come by scaling up SIMD computation with platforms such as graphical processing units (GPUs).

2.2 SIMD DNA structure

SIMD implemented on DNA is intriguing. It provides a means to transform stored data, perhaps large amounts of it, with a single parallel instruction. We will review the paradigm as we introduce our new encoding scheme and our new applications; of course, we do not claim credit for the original concepts. The reader is referred to [6].

SIMD DNA computation is predicated on the encoding scheme for data. Conceptually, we divide stretches of double-stranded DNA into "domains", where each domain is a contiguous sequence of nucleotides of some small specified length (typically 5 to 20). A sequence of several (typically 5 to 7) domains maps to a "cell" storing one binary bit. Whether a cell stores a 0

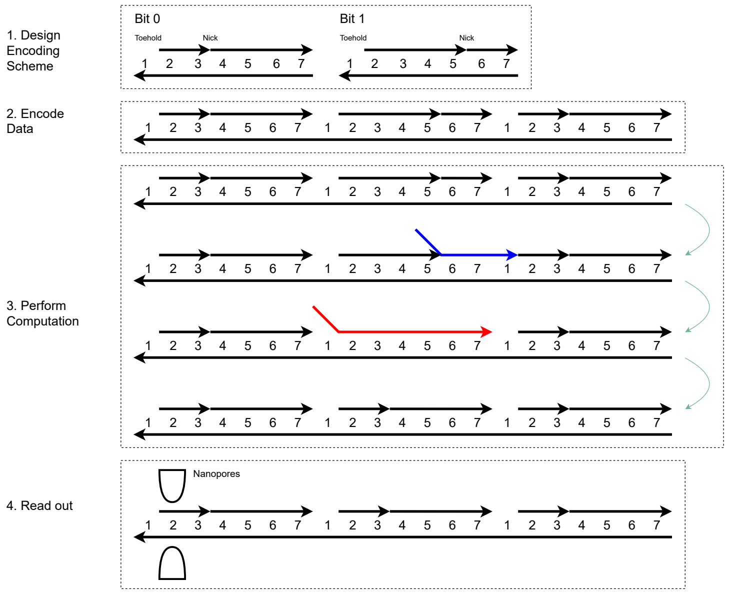


Fig. 1: General Outline of SIMD DNA Computations. Stage 1 shows the encoding of binary bits 0 and 1 across the 7 domains per bit. Stage 2 shows an example of encoding the bits 010. Stage 3 illustrates the step in which computation is performed with strand displacement, in a general sense. Details of this step will be provided for specific algorithms in later sections. Note that, in this generic example, the location of nick in the second cell has changed at the end of stage 3. Stage 4 illustrates how nanopore sequencing could be used to perform readout.

or a 1 depends upon topological variations, specifically the location of nicks, i.e., breaks in the DNA backbone. The nicks always occur on one strand of a double-stranded complex (generally the top strand in our examples); the other remains untouched.

The computation is carried out by a sequence of "instructions", where each instruction implements DNA strand displacement reactions on cells. Instructions are initiated by single-stranded "instruction strands" added to the solution. After the strand displacement cascades complete, all freely floating fragments in the solution are washed away; the original strand is kept and separated via a magnetic bead. After a sequence of instructions, the data is transformed to its final state. The readout can be performed via fluorescence or with Oxford nanopore devices [13], [4].

Our approach to computation is summarized as follows and illustrated in Figure 1.

1. Design an encoding structure that best suits the algorithm.

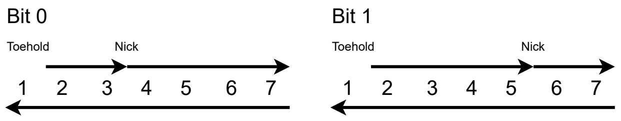


Fig. 2: Bit representation in the encoding scheme. Horizontal lines represent DNA strands. Integers represent "domains": specific sequences of nucleotides. Arrow heads represent nicked positions: places where the phosphodiester bond in the backbone of the DNA strand has been broken, via gene-editing techniques. Cells store binary values. Each cell consists of 7 domains. Domain 1 is always exposed, forming a toehold.

2. Encode the data at specific locations, using enzymes to nick corresponding targets.
3. Gently denature the DNA, allowing segments between adjacent nicks to detach, exposing toeholds.
4. Execute instructions, implemented as strand-displacement operations.
5. Finally, read out data using fluorescence or with nanopores.

3 Design of Encoding System

Several schemes for encoding binary data were proposed in prior work [6], each chosen to minimize the number of operations for a specific algorithm. Here we propose a new encoding scheme that works well for the broad class of algorithms that consist of parallel operations on pairs of bits. A requirement for running these algorithms is that the encoding scheme should allow the algorithm to read bits adjacent to each other. This specification comes at the expense of more complexity for some algorithms, i.e., more operations per step than possible with a customized encoding.

The encoding scheme is shown in Figure 2. Each cell stores a single binary value (a "bit"). Each cell consists of 7 domains. We do not specify the actual nucleotide sequence of the domains here for simplicity. While preparing this cell, the top DNA strand must be nicked before and after domain 1. This strand can then be displaced by denaturing, creating an exposed toehold. Domain 1 is always exposed as a toehold in this representation. Domains 2 through 7 are covered. When storing a bit 0, we will nick the top strand between domains 3 and 4; when storing a bit 1, we will nick between domains 5 and 6. There are four possible pairings for two adjacent cells. Each will be detected using different domain combinations: for $(0, 0)$, domains 1, 2 and 3; for $(0, 1)$, domain 1 only; for $(1, 0)$, domains 6 through 3 with wrapping at domain 7 and 1; and for $(1, 1)$, domains 6, 7 and 1.

Before describing the implementation of specific algorithms for sorting, shifting, and searching, we will present some general algorithmic steps useful in implementing all of these.

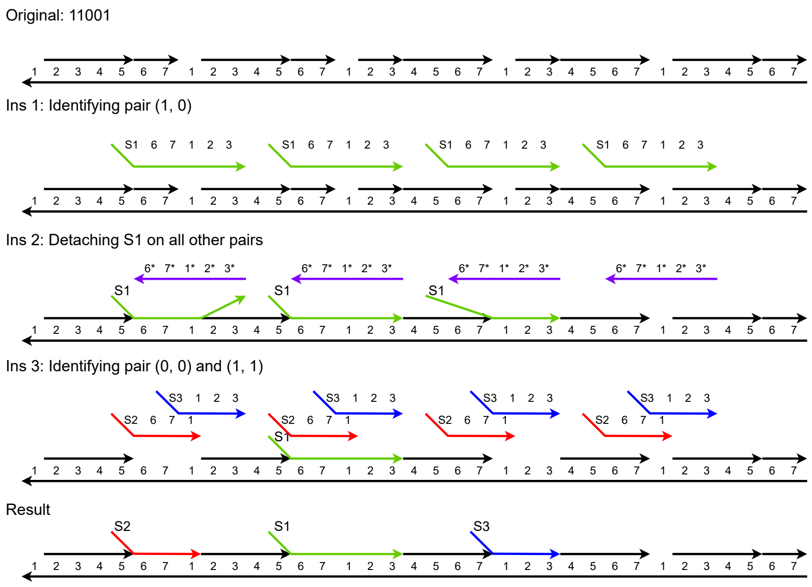

Original: 11001
Ins 1: Identifying pair (1, 0)
Ins 2: Detaching S1 on all other pairs
Ins 3: Identifying pair (0, 0) and (1, 1)
Result
S1 6 7 1 2 3
6* 7* 1* 2* 3*
S1
S2 6 7 1
S3 1 2 3
S2
S3
1 2 3 4 5 6 7

Fig. 3: Example of Identifying Different Pairs of Adjacent Bits.

3.1 Identifying Bit Pairs

A common task in our algorithms is "identifying" pairs of adjacent bits, i.e., recognizing the specific pair of cells at a location of interest. We will exploit the fact that domain 1 is always exposed to identify these specific pairs. Figure 3 illustrates our approach on the string 11001, which contains all 4 possible adjacent pairs: $00, 01, 10$ and 11.

Identification is performed with three instructions. In instruction 1, the strands (S_1 6 7 1 2 3) are issued to all pairs of bits. S_1 first binds at the toehold of domain 1, between each pair. If this preceding bit has a value of 1, there will be a nick between domains 5 and 6. Through branch migration, the left side of S_1 (i.e., the (S_1 6 7) part) will displace the original strand covering domains 6 and 7 of the preceding bit. This is shown in Figure 3, instructions 1 and 2. If the value of the following bit is 0, there will be a nick between domains 3 and 4. Through branch migration, the right side of S_1 (i.e., the (S_1 2 3) part) will displace the original strand covering domains 2 and 3. This is shown in Figure 3, instructions 1 and 2. Only if the preceding bit is 1 and the following bit is 0 will S_1 displace *both* these strands. For the pair $(1, 1)$, domains 2 and 3 of S_1 are left overhanging. For the pair $(0, 0)$, domains 6 and 7 of S_1 are left overhanging. For the pair $(0, 1)$ S_1 will not bind at all, since the only exposed toehold is domain 1. This is how the algorithm *identifies* the pair $(1, 0)$.

In instruction 2, using the complementary strands (6* 7* 1* 2* 3*), the strand S_1 that attaches to the pairs $(0, 0)$ and $(1, 1)$ is pulled out. This is done through the open domains 2 and 3 in the pair $(0, 0)$ and the open domains 6

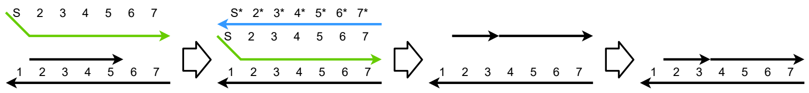


Fig. 4: Example of Rewriting in Three Steps

and 7 in the pair $(1, 1)$ on strand S_1. After this instruction, strand S_1 remains only for the pair $(1, 0)$.

In instruction 3, two instruction strands are issued at the same time: (S_2 6 7 1) and (S_3 1 2 3). Here (S_2 6 7 1) will bind to the pair $(1, 1)$ and (S_3 1 2 3) will bind to the pair $(0, 0)$. They will not bind with any other pairs since the only exposed toehold for binding would be domain 1; they will prefer the locations with more exposed domains.

The result is that the adjacent bit pairs $(1, 1)$, $(1, 0)$ and $(0, 0)$ are each *labeled* with strands S_2, S_1 and S_3 respectively. Pairs $(0, 1)$ are labeled with an exposed toehold at domain 1. This toehold could be replaced by a strand (S_x 4 5 6 7 1) or a strand (S_x 1 2 3 4 5); the choice would be made depending on the use case.

3.2 Rewriting a cell

By exposing toeholds across domains 2 through 7 in a cell, we can rewrite the content of that cell – so change a 1 to 0 or a 0 to 1 – with three instructions. The idea is that, since there are exposed domains, we can displace the content of the cell with a single strand covering all these domains. Then we can remove the covering strand through the exposed "tag" domain (S in Figure 4) using a complementary strand. The cell is now completely exposed. We can write a new bit to it by hybridizing the strands according to our encoding scheme, leaving domain 1 as a toehold and placing the nick at the desired location.

4 Parallel Binary Bubble Sorting

Sorting is a simple yet fundamental operation in computer science. Here we consider sorting binary values.[1] Sorting can be used to determine the "weight" of a vector of 0's and 1's: the count of the number of 1's relative to the length of the vector. It can also be used to compute the majority function: whether there are more 1's than 0's or not in the input set. Majority is a fundamental operation for many machine-learning algorithms.

Our SIMD DNA implementation performs parallel bubble sorting on binary bits [14]. It can be expressed as a pairwise operation in the form of $f(a, b) = (c, d)$, where (a, b) is the value of the input bit pair, and (c, d), the output pair. f represents the action we take on a given bit pair – whether to rewrite or to leave it as it is. The individual bit outputs can be 0 or 1, which means that we

[1]Perhaps counter-intuitively, sorting binary values in hardware is as difficult algorithmically as sorting arbitrary values such as integers or real numbers [14]

can arbitrarily change the value of the cell. We discuss what kind of pairwise operations can be performed on our encoding in Section 8.2.

The sorting operation can be expressed in the following pairwise operation,

$$f(0,0) = (0,0), \quad f(0,1) = (0,1), \quad f(1,0) = (0,1), \quad f(1,1) = (1,1).$$

Note that only the third input pair $(1,0)$ needs to be rewritten to $(0,1)$. The rest of the bit pairs do not need to be changed.

Algorithmically, the following "bit swapping" is performed:

- If the current bit is 1, it changes it to 0 if and only if its right neighbor is 0.
- If the current bit is 0, it changes it to 1 if and only if its left neighbor is 1.

We argue that repeatedly performing such bit-swapping will sort the entire sequence of binary values. An example is provided in Appendix F.

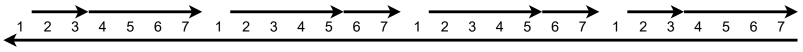
Proposition 1 *The $f(1,0) = (0,1)$ pairwise operation can only occur once in any sequence of three bits.*

Proof It is impossible to have two consecutive, overlapping pairs $(1,0)$ spanning three bits. Therefore, the $f(1,0) = (0,1)$ pairwise operation (i.e., the bit-swap step) can only occur once in any sequence of three bits. Consequently, the bubble sort algorithm only performs non-conflicting pairwise operations. (Please see Section 8.2 for more details.) □

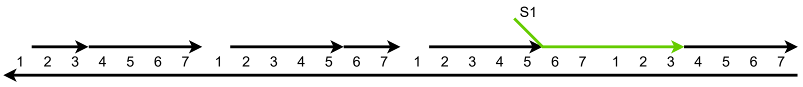
Accordingly, bubble sorting binary values in parallel does not require an odd and even index addressing scheme, as does bubble sorting arbitrary values.

Proposition 2 *Sorting completes in at most $N-1$ parallel steps where N is the total number of bits.*

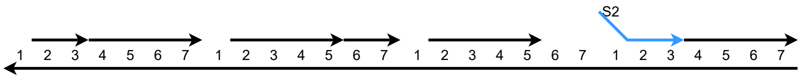
Proof Suppose we have a sequence of binary bits of length N, in which all bits except the first are 0. When applying the algorithm, the 1 located at the start will be pushed back one position at a time with the $f(1,0) = (0,1)$ bit swap operation. Fully sorting the sequence, i.e., moving the 1 to the last position, requires $N-1$ total swaps. Now suppose we are sorting an arbitrary bit sequence. We argue that, after $N-1$ swaps, all the 1's will be at the end of the sequence. To see why, note that an $f(1,0) = (0,1)$ operation moves a 1 forward, while an $f(1,1) = (1,1)$ operation does not affect adjacent 1's. Thus, in $N-1$ steps, all 1's will have moved to the end of the sequence. □



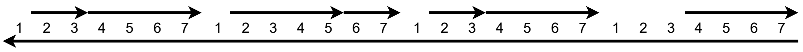
(a) Initial Sequence 0110



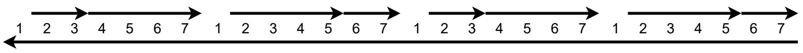
(b) After Recognizing $(1, 0)$



(c) Protection on Bit 0



(d) Flipped third bit to 0, Protection Removed



(e) Flipped fourth bit to 1, Result 0101

Fig. 5: Outline of the SIMD DNA parallel binary sorting algorithm.

4.1 Implementation

Here we give an instruction set for performing parallel binary bubble sort with SIMD DNA, using the encoding in Figure 2. It consists of 12 individual instructions. These are summarized as follows.

1. Label pairs $(1, 0)$.
2. Uncover these, leaving domains 6 and 7 for the bits 1 and domains 2 and 3 for the bits 0 open in these pairs.
3. Protect the bits 0 of these pairs by covering the corresponding toehold at domains 2 and 3.
4. Flip the bits 1 to 0 in these pairs.
5. Release the protective covers; flip the bits 0 to 1 in these pairs.

For the initialization, we can use the first two instructions described in Section 3.1, with an additional instruction to fix open domains for bits that do not change. We can use the rewriting method described in Section 3.2 to flip the bits. A full description of the implementation of sorting is provided in Appendix B.

5 Parallel Exclusive OR

The Exclusive OR operation, shortened to XOR, is a useful bit operation with many applications, including in error correction. Simply put, a multi-input XOR operation checks if there are an odd number of 1's in the input bits. With our implementation, it is possible to compute the XOR of all bits on a strand. This can be achieved with the following pairwise operations per step of the algorithm:

$$f(0,0) = (0,0), \quad f(0,1) = (0,1), \quad f(1,0) = (0,1), \quad f(1,1) = (0,0).$$

These pairwise operations are mostly similar to the ones for the parallel bubble sorting, in particular with the $f(1,0) = (0,1)$ that sorts all 1's and pushes them to the right. However, the $f(1,1) = (0,0)$ modification ensures that any contiguous pair of cells both containing 1 are overwritten to 0. This means that after every individual step of the XOR algorithm, the parity of 1's is preserved, and all 1's are shifted one bit towards the right. After a certain number of steps, the last bit on the strand will store the XOR output. One issue that must be addressed with this algorithm is how cells are paired per step. For example, if the triplet of cells (1,1,1) are recognized as two pairs of $(1,1)$, then the overwriting step will change all three 1's to 0 and break the parity of the sequence. To avoid this, all DNA strand instructions identifying (1,1) pairs must be run on non-overlapping pairs of cells at each step. For example, the first step should operate on pairs of cells i and $i+1$ where i is an even number, and then the next step should operate on pairs i and $i+1$ where i is an odd number. However, the $f(1,0) = (0,1)$ operation can be run on overlapping pairs.

To perform such operations, i.e., one iteration on only even-to-odd pairings and the next iteration on only odd-to-even pairings, the cells must have unique sequences. The even and odd cells must have different DNA base sequences – all even cells based on one sequence, and all odd cells are based on another, distinct sequence. The nicking architecture, shown in Figure 2, applies despite the different base sequences. Changing the sequences ensures that instruction strands can be synthesized to bind to the appropriate i and $i+1$ cells discussed above.

Each iteration of the XOR algorithm is thus as follows:

- Determine a pairing that is offset by 1 cell compared to the previous iteration's pairing.
- Detect all non-overlapping pairs of (1,1) and convert them to (0,0).
- Detect all pairs of (1,0) and convert them to (0,1). For this writing process, pairs can be overlapping.

Each of these iterations pushes all 1's to the end of the strand while also overwriting any adjacent (non-overlapping) pairs of 1's. Therefore, after a sufficient number of iterations, all bits in the strand will be 0's except for the last bit – that bit will only be 1 if there were an odd number of 1's to begin with. Therefore, the algorithm computes the XOR function.

Proposition 3 *The parallel exclusive OR completes in at most N parallel steps where N is the total number of bits.*

Proof Consider the worst case of an array of N bits with two 1's, one at the start and one at the end. For the correct XOR computation, the first 1 must be moved to the end of the array, which requires at most $N-1$ steps. Then $f(1,1) = (0,0)$ requires one final step for the proper XOR output. Now any extra 1's added to the array occurr between the start and end. These will be moved to an adjacent position in those $N-1$ steps. The $f(1,1) = (0,0)$ operation for these two 1's in the middle of the array does not impede sorting the first 1 in the array. Thus computing the XOR of N bits requires at most 1 step more than the worst-case parallel sorting time. Therefore, any arbitrary array of N bits requires $(N-1)+1 = N$ steps for a correct XOR computation. □

5.1 Implementation

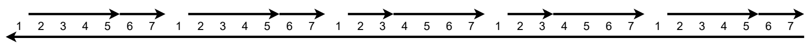
Here we give an instruction set for performing one step of the parallel exclusive OR with SIMD DNA, using the encoding in Figure 2. It consists of 20 individual instructions. A few of these instructions can be merged, but for the sake of completeness, we describe them separately. These instructions are summarized as follows.

1. Label non-overlapping pairs $(1,1)$.
2. Cover all other pairs.
3. Uncover the identified $(1,1)$ pairs and expose both bits in this pair.
4. Rewrite all uncovered bits to 0.
5. Now label all pairs of $(1,0)$.
6. Uncover the $(1,0)$ pairs.
7. Protect the bits 0 of these pairs by covering the corresponding toehold at domains 2 and 3.
8. Flip the bits 1 to 0 in these pairs.
9. Release the protective covers; flip the bits 0 to 1 in these pairs.

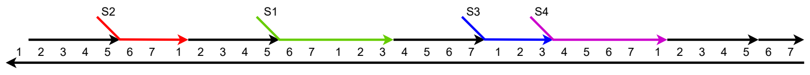
Please note that in each step, the non-overlapping pairing is offset by one cell compared to the preceding step to ensure all (1,1) pairs are overwritten. A full description of the implementation of the XOR is provided in Appendix C.

6 Parallel Left Shifting

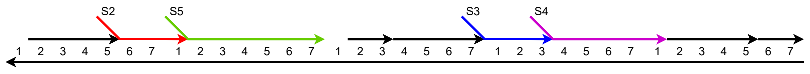
We propose a SIMD DNA implementation of shifting, another fundamental operation in computer science. Shifting left corresponds to multiplying a binary number by 2; shifting right corresponds to dividing it by 2. It is a useful operation in general for aligning data in a variety of algorithms [14]. We present



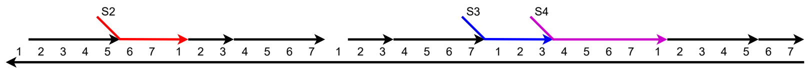
(a) Initial Sequence 11001



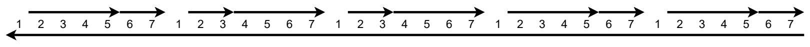
(b) After identifying all pairs



(c) Release S_1 from Pair $(1, 0)$



(d) Rewrite bit 1 in the previous pair with 0



(e) Release S_2, S_3 and S_4 then write 1, Result 10011

Fig. 6: Outline of the SIMD DNA parallel left shift operations. The initial sequence S is 11001 and the result sequence T is 10011. The operation shifts each bit to the left one position (T[5:1]=S[4:0]), while keeping the Least Significant Bit unchanged.

a left shift algorithm, one that shifts all N binary bits one position to the left, with the Least Significant Bit (LSB) remaining unchanged. This operation is, of course, a parallel left shift, moving all bits simultaneously in lockstep. Our implementation requires 11 instructions per shift. Note that unlike usual arithmetic or a logical left shift that inserts a bit 0 to the LSB, the left shift operation described here keeps the original LSB, thereby duplicating the LSB. The usual left shift could be implemented by adding instructions rewriting the LSB to 0 after the instructions we provide here.

We describe the shift operation using the following pairwise operation as follows:

$$f(0,0) = (0, X), \quad f(0,1) = (1, X), \quad f(1,0) = (0, X), \quad f(1,1) = (1, X).$$

Here X is a "don't care" bit value (to use the parlance of digital design). When computing on a specific input bit pair, the output for the X bit is not impacted by that input pair (for example, in left shifting, X is actually calculated from the bit pair to the right since that bit will be shifted to the left). For each bit pair, the operation writes the value of the right bit to the left bit. Since only the value of the left bit is changed in each bit pair, the operation is non-overlapping and can be implemented using the encoding scheme we propose. We illustrate with the example of shifting 11001 to 10011, shown in Figure 6.

1. Label all the bit pairs. Cover the toeholds for the pairs $(0, 0)$ and $(1, 1)$.
2. For the pairs $(1, 0)$, flip the bits 1 to 0.
3. For the pairs $(0, 1)$, flip the bits 0 to 1.
4. Finally, uncover all the toeholds for the pairs $(0, 0)$ and $(1, 1)$.

A full description of the implementation of shifting is given in Appendix D.

7 Parallel Search Algorithm

Searching is fundamental to all branches of computer science that involve data storage and retrieval. We consider the problem of deciding whether a given substring exists in a stored string of bits. We first discuss a general algorithm that returns an answer to such a question in $\log(n)$ parallel steps, where n is the substring length. We then propose an implementation in SIMD DNA. Due to practical constraints, the time complexity of the implementation is not $O(\log(n))$; it is closer to $O(n)$, depending on the problem size and implementation details. We note that a requirement of our algorithm is that the length of the query string is a power of 2. We discuss the time complexity and constraints in detail in Section 8.4.

7.1 Algorithm

Suppose we have a *query* substring Q of a length n, and we would like to search whether it appears in a much longer *target* string A. Pseudocode for our approach is given as Listing 1. We will elucidate the pseudocode by stepping through examples.

7.1.1 Parallel search procedure

We illustrate searching for a query string $Q = 1101$ in the following target string A:

$$\begin{aligned} A_0 &= 10101010\textcolor{red}{1101}10100011\textcolor{red}{1101}01000100 \\ A_1 &= a_2a_2a_2a_2a_3a_1a_2a_2a_0a_3a_3a_1a_1a_0a_1a_0 \\ A_2 &= b_0b_0\textcolor{red}{b_1}b_0b_2\textcolor{red}{b_1}b_3b_3 \end{aligned} \tag{1}$$

The original string is A_0. In each step, two consecutive symbols are read and replaced with a single symbol. Here $a_0 = 00, a_1 = 01, a_2 = 10, a_3 = 11, b_0 = a_2a_2, b_1 = a_3a_1, b_2 = a_0a_3, b_3 = a_1a_0$. Note that $Q = 1101 = a_3a_1 = b_1$. After three steps, we conclude that the query string exists in the target string since there are two matches in the string A_2.

Listing 1: Pseudo-code for Parallel Search Algorithm. Note that the operations inside the two `while` loops can be performed in parallel since they are independent. The `pair` operation here is to find a corresponding symbol that replaces the two symbols in the lookup table, and the `identity` operation is to look up the symbol that represents the query string.

```
Q = Query String
A = Target String
n = length of Q
for i in range(0,n-1):
    A_i = A
    truncate first i characters of A_i
    p = 1
    while p <= n:
        j = 0
        while j < (length(A_i)-1):
            x = A_i[j]
            y = A_i[j+1]
            z = pair(x,y) # Pair 2 consecutive cells
            if z.identity(Q): # Check if it is query
                return True
            replace x,y in A_i with z
            j += 1
        p = 2*p
return False
```

7.1.2 Search procedure with offset

It is possible that the query string does not align with divisions of length n in the target string. Thus, we need to repeat the operation with offsets. The following example illustrates the operation with an offset of 2 bits.

$$\begin{aligned} A_0 &= \cancel{10}10101101011000000111100010001 00 \\ A_1 &= \cancel{10}a_2a_2a_3a_1a_1a_2a_0a_0a_3a_3a_0a_1a_0a_1a_0 \\ A_2 &= \cancel{10}b_0b_1b_2b_3b_4b_5b_5\cancel{a_0} \end{aligned} \tag{2}$$

Here, the replacement is given by the aggregated pairs $a_0 = 00, a_1 = 01, a_2 = 10, a_3 = 11, b_0 = a_2a_2, b_1 = a_3a_1, b_2 = a_1a_2, b_3 = a_0a_0, b_4 = a_3a_3, b_5 = a_0a_1$. Again, an instance of the query string is found in the target string.

Searching for a query string with a given offset requires at most $\log(n)$ steps. In general, for an arbitrary query string of a length n (a power of 2), the search must be performed n times with offsets ranging from 0 to $n-1$. In principle, all of these searches could be performed in parallel, as none would interfere with any other. Accordingly, our parallel implementation of searching completes in $\log(n)$ steps.

Note that the number of aggregated pair identifiers needed – the a's and b's in the example above – grows exponentially with the length of the target string. For example, to search for all possible queries of length 2, 4 identifiers are needed. For all queries of length 4, $16 + 4 = 20$ identifiers are needed. The total number of identifiers needed for queries of length n can be formulated as:

$$\sum_{i=1}^{\log(n)} 2^{2^i}.$$

This number grows very quickly as n increases (so for longer query strings). This would seem to be a serious limitation of our algorithm. However, this calculation assumes that we are searching for *all* possible query strings. If the search is for a *specific* query string, then the number of identifiers required drops considerably. This is because the search only needs to identify pairs in this specific string. The maximum number of identifiers needed is:

$$\sum_{i=1}^{\log(n)} 2^i = n - 1,$$

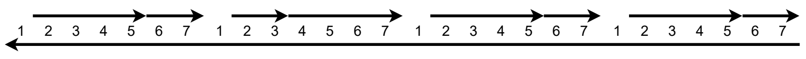
a much more manageable number.

7.2 Implementation

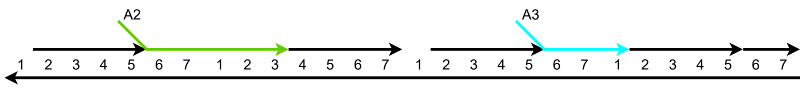
To implement the algorithm in SIMD DNA, we do not issue instruction strands to each pair of overlapping bits. Instead, we consider the non-overlapping bit pairs. In the example shown in Figure 7, for the bit sequence 1011, we would consider operations on bit pair 10 and 11, but not on bit pair 01.

Figure 7 shows the critical steps when searching a target sequence 1011. It provides an example of a successful search and also the potential outcome of two failed searches. To implement the search operation with an offset, we can simply skip the number of bits according to the offset. We use the word *symbol* to represent the consecutive cells that we search for on a certain level. For example, in the first level, the symbols are 10 and 11. We can use the bit-identifying steps described in Section 3.1 to recognize these symbols. We use identifiers $A_0 = 00, A_1 = 01, A_2 = 10, A_3 = 11$ to represent symbols in this level. We then move on to the next level, searching for consecutive symbols A_2A_3, which corresponds to the target string 1011.

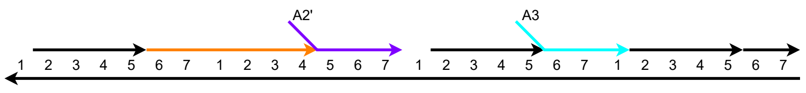
In the first step of the second level, we first rewrite the topological structure at symbols that appear to be a query result. In this example, A_2 should be found as the left symbol, and A_3 should be found as the second symbol. We pull identifier A_2 out from every *odd* symbol (we only look at the first, third, fifth, etc.) and rewrite the entire symbol with the technique described in Section 3.2. After rewriting, we have the identifier A_2' that covers domains (5 6 7) in the *right most* cell, as seen in Figure 7c. For the second symbol A_3, we repeat the step described, except we pull the identifier out from every *even* symbol and



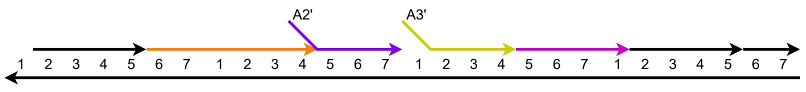
(a) Initial Sequence 1011



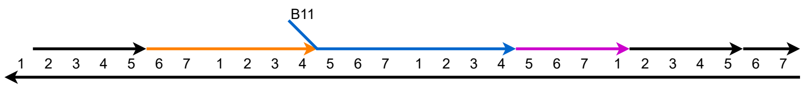
(b) Identifier A_2 captures first pair 10, A_3 captures second pair 11



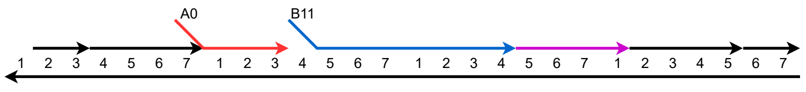
(c) covering the domain 1 between the two bit pairs



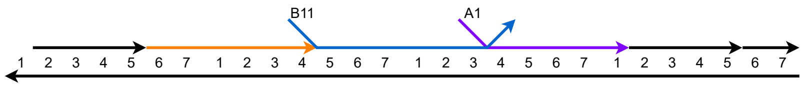
(d) Rewrite the content in the pair so that new identifiers are close to the middle



(e) Two identifier strands replaced by a single identifier if there is a perfect match



(f) Initial sequence is 0011. It will result in an open domain 4 in the cell left of the identifier



(g) Initial sequence is 1010. It will result in an overhanging domain 4 on the identifier strand itself

Fig. 7: Example implementation of search algorithm on target sequence 1011

the new identifier A'_3 covers domains (2 3 4) in the *left most* cell. Through these steps, we have essentially "moved" the identifier of the matching symbols to the middle. In the final step, we issue the new identifier strand (B_{11} 5 6 7 1 2 3 4) to the location between every two symbols. It will result in a perfect binding only if there is a match at the current symbol level. Figure 7e shows the example of a matching result. Figure 7f and 7g show two potential examples of imperfect binding, indicating a non-matching result. We can pull them out through the open domains either on the identifier itself or a nearby open domain on the base strand. Therefore, the presence of the identifier B_{11} indicates a successful match.

We can repeat the process to recognize multiple symbols at the same level. When we move to the next level $l+1$, we can use the identifiers from this level l as a starting point for rewriting. To identify a symbol $S_{l+1,c} = S_{l,a}S_{l,b}$ at level $l+1$, we simply pull out identifiers for $S_{l,a}$ at odd symbols and $S_{l,b}$ at even symbols at level l. Then we "move" the identifier to the middle. Finally, we give identifiers for $S_{l+1,c}$ to the middle of each pair and identify the symbol.

A possible weakness of our implementation is that the strand used for rewriting could potentially be very long. The longer the query string, the longer this strand. The longer the strand, the longer strand displacement takes [15]. The time required could become prohibitive. Another issue is that our search algorithm rewrites, so destroys, the data on the target strand. While it may be possible to reverse the process, so restore the original data, this process would be cumbersome and require multiple steps (see, for example, E7). Another limitation is that the algorithm cannot readily handle multiple overlapping queries within the target string.

8 Discussion

We discuss the features and implementation constraints of the proposed algorithms.

8.1 Initializing data on cells sharing the same sequences

Two of the algorithms that we discuss, namely XOR and searching, rely on different cells having different underlying DNA sequences. This allows the algorithms to perform pairwise operations that target specific cells, leaving others untouched. The other two algorithms that we discuss, namely sorting and left-shifting, do not have this requirement. As a result, sorting and left-shifting require much smaller libraries of DNA strands. Sorting and left-shifting are true "single instruction multiple data" (SIMD) algorithms while XOR and searching are not. In order to exploit the SIMD aspect of sorting and left-shifting, the underlying DNA sequences must be identical for all cells. The challenge is that any nicking operation performed on one cell would apply to other cells as well, so one cannot readily initialize the base strand with different bit values in different cells.

One approach to overcome this would be to utilize Gibson Assembly, a procedure for concatenating small DNA molecules into larger DNA molecules [16]. The small DNA molecules are synthesized with complementary "sticky" single-stranded ends. When these small DNA molecules are mixed in a solution, these sticky ends hybridize, yielding a longer DNA molecule. This approach can be used to initialize data in our DNA strands where all cells share the same sequence. Molecules containing only one cell are nicked separately to store either 0 or 1. These molecules are then orderly concatenated to build strands containing multiple cells. Please refer to H for a more detailed outline on constructing a register storing two bits.

8.2 Ability to compute any non-conflicting pairwise operation

In Section 4 and Section 6, we presented examples of algorithms that perform pairwise operations, namely sorting and shifting, respectively. Given the ability to identify pairs of bits and a universal way to rewrite a cell, we can readily implement any algorithm that performs non-conflicting pairwise operations. Such operations only entail rewriting pairs of adjacent bits. The result of the operation on a specific sequence should always be the same, irrespective of the execution order. To illustrate, consider the following operation:

$$f(0,0) = (X,X), \quad f(0,1) = (X,1), \quad f(1,0) = (X,X), \quad f(1,1) = (0,X).$$

Here X indicates a "don't care" bit value – the function f for a specific input pair does not compute the output X. The operation provided above *is* conflicting. To see why, consider its effect on the sequence 011. The second bit should change to 1 when the operation is applied to the first pair $(0,1)$. However, this bit should change to 0 when the operation is applied to the second pair $(1,1)$. Depending on the order of execution, the final result will be different. To ensure an operation is non-conflicting, for every three adjacent bits that two operations are performed on, the middle bit should be set to the same value.

Non-conflicting operations can be performed in parallel on all bit pairs. In the first step, we identify the four bit pairs described in 3.1. After this step, we supply strands with four labels covering the four bit pairs. Then, we release strands with specific labels one at a time to obtain write access to specific bit pairs. (Write access refers to a domain being exposed.) We rewrite these cells with the operation described in Section 3.2. The full operation requires rewriting all four bit pairs.

We conclude that our encoding scheme and design method are generally applicable to parallel bitwise algorithms, provided that they can be expressed in terms of such non-conflicted pairwise operations.

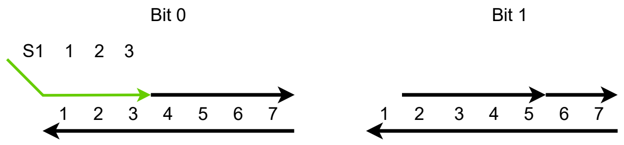


Fig. 8: One strand can be used to differentiate two bits

8.3 Converting to Different Encoding Schemes

A benefit of the encoding scheme that we are proposing is that it can easily be converted to any other similar scheme since each cell always has an exposed domain 1. In the original SIMD DNA scheme proposed in [6], the authors designed two specific encoding schemes for the two applications proposed (rule 110 and a binary counter). We suggest that our encoding scheme could be used as an intermediate form when converting to other encoding schemes, designed for particular algorithms. Figure 8 illustrates how we can use a single strand (S_1 1 2 3) to differentiate bit values of 0 from bit values of 1. We can use the technique discussed in 3.2 to re-write the data with a different encoding scheme, so long as the scheme also encodes each bit with 7 domains. Complete instructions for performing such encoding changes are given in Appendix A.

8.4 Time Complexity of Parallel Search

While the time complexity of the proposed parallel search is $O(\log(n))$ in principle, where n is the query substring length, the time complexity of our SIMD DNA implementation is somewhat worse. While the abstract search algorithm finds the query in the reference string by pairing individual characters in parallel, and thus completes in $O(\log(n))$ steps, our implementation searches for and identifies distinct symbols sequentially, that is to say, it first searches for a specific symbol across all possible locations at once, then it searches for the next symbol across all locations at once, and so on.

The abstract algorithm assumes all symbols are identified in one pass to allow for further pairing. If we consider all the different symbols in a query string, counting repeated symbols, $\frac{n}{2^i}$ symbols must be searched sequentially at level i in our implementation. Accordingly, the total number of sequential search steps could be as high as $O(n)$. However, at each level, all the occurrences of a specific symbol are identified simultaneously. At level i, each symbol represents a binary string with a length of 2^i, so there are at most 2^{2^i} distinct symbols at level i. For example, in the first level, instead of searching for $\frac{n}{2}$ symbols, we only search for four distinct symbols. In the second level, there are only 16 distinct symbols. Since we only search for distinct symbols, the number of steps in the first few levels will be greatly reduced.

Our parallel search algorithm currently only works on query strings having a length that is a power of two. However, we believe that our implementation could be modified to allow for arbitrary-length query strings. We do not provide details here, as they are cumbersome, but we outline the method as follows.

Note that, in parallel search, the query string is searched reductively: at each level, two symbols are reduced to one symbol. When working with query strings having any arbitrary length, there might be an odd number of symbols in the current level. In this case, we can add a method to identify the trailing odd symbol at the current level and replace it in the next level. The reduction can still be completed in a logarithmic number of levels.

9 Conclusion

We have presented algorithms for basic parallel operations within the SIMD DNA framework. We note that there are, in fact, two layers of parallelism possible:

1. Bit-level Parallelism: instructions applied to all bits in an array at once.
2. Data-level Parallelism: the same instructions applied to *multiple* arrays at once.

While operations on DNA are slow and error-prone, with these levels of parallelism, perhaps DNA computation could scale to a truly impressive regime. Consider the following back-of-an-envelope estimates, all admittedly widely optimistic. Suppose:

- We have 10^{12} independent cells in parallel in a single test tube;
- A single operation takes approximately 10 minutes to complete [17];
- Different cells use the same DNA sequence. Using distinct sequences for different cells, as in our search operation, can result in a solution with multiple competing DNA molecules. At larger scales, this would result in an increase in reagent volume and could diminish reaction rates.

This means that we can perform approximately 10^9 operations per second in a single test tube, already impressive. Now suppose that we have 100 test tubes. This means we can compute at 100,000 MIPS (million instructions per second). This is comparable to what very respectable existing silicon systems can achieve. The key conceptual difference between the SIMD DNA approach and other forms of DNA computing is that it exploits a substrate on which data is stored. This enables the SIMD parallelism.

Many experimental hurdles remain in demonstrating and deploying this paradigm. DNA synthesis remains prohibitively expensive. A possible alternative is to use gene-editing techniques to encode data on naturally occurring DNA [18].

References

[1] Adleman, L.M.: Molecular computation of solutions to combinatorial problems. Science, 1021–1024 (1994)

[2] Ceze, L., Nivala, J., Strauss, K.: Molecular digital data storage using DNA. Nature Reviews Genetics **20**(8), 456–466 (2019). https://doi.org/

10.1038/s41576-019-0125-3

[3] Church, G., Gao, Y., Kosuri, S.: Next-generation digital information storage in DNA. Science (New York, N.Y.) **337**, 1628 (2012). https://doi.org/10.1126/science.1226355

[4] Liu, K., Pan, C., Kuhn, A., Nievergelt, A.P., Fantner, G.E., Milenkovic, O., Radenovic, A.: Detecting topological variations of DNA at single-molecule level. Nature communications **10**(1), 1–9 (2019)

[5] Tabatabaei, S.K., Wang, B., Athreya, N.B.M., Enghiad, B., Hernandez, A.G., Fields, C.J., Leburton, J.-P., Soloveichik, D., Zhao, H., Milenkovic, O.: DNA punch cards for storing data on native DNA sequences via enzymatic nicking. Nature communications **11**(1), 1–10 (2020)

[6] Wang, B., Chalk, C., Soloveichik, D.: SIMDDNA: Single instruction, multiple data computation with DNA strand displacement cascades. In: Thachuk, C., Liu, Y. (eds.) DNA Computing and Molecular Programming, pp. 219–235. Springer, Cham (2019)

[7] Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. Proceedings of the National Academy of Sciences **107**(12), 5393–5398 (2010) https://arxiv.org/abs/https://www.pnas.org/content/107/12/5393.full.pdf. https://doi.org/10.1073/pnas.0909380107

[8] Yurke, B., Turberfield, A.J., Mills, A.P., Simmel, F.C., Neumann, J.L.: A DNA-fuelled molecular machine made of DNA. Nature **406**(6796), 605–608 (2000)

[9] Chen, T., Solanki, A., Riedel, M.: Parallel pairwise operations on data stored in DNA: Sorting, shifting, and searching. In: 27th International Conference on DNA Computing and Molecular Programming (DNA 27) (2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik

[10] Krug, J., Spohn, H.: Universality classes for deterministic surface growth. Physical Review A **38**(8), 4271 (1988)

[11] Li, W.: Power spectra of regular languages and cellular automata. Complex Systems **1**(1), 107–130 (1987)

[12] Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Transactions on Computers **C-21**(9), 948–960 (1972)

[13] Athreya, N., Milenkovic, O., Leburton, J.-P.: Detection and mapping of dsDNA breaks using graphene nanopore transistor. Biophysical Journal **116**(3), 292 (2019)

[14] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press, ??? (2009)

[15] Radding, C.M., Beattie, K.L., Holloman, W.K., Wiegand, R.C.: Uptake of homologous single-stranded fragments by superhelical dna: Iv. branch migration. Journal of molecular biology **116**(4), 825–839 (1977)

[16] Li, L., Jiang, W., Lu, Y.: A modified gibson assembly method for cloning large dna fragments with high gc contents. Synthetic Metabolic Pathways: Methods and Protocols, 203–209 (2018)

[17] Broadwater, D.B., Kim, H.D.: The effect of basepair mismatch on dna strand displacement. Biophysical journal **110**(7), 1476–1484 (2016)

[18] Tabatabaei, S., Wang, B., Athreya, N., Enghiad, B., Hernandez, A., Fields, C., Leburton, J.-P., Soloveichik, D., Zhao, H., Milenkovic, O.: DNA punch cards for storing data on native DNA sequences via enzymatic nicking. Nature Communications **11** (2020). https://doi.org/10.1038/s41467-020-15588-z

[19] Doty, D., Ong, A.: Simulating 3-symbol turing machines with simd dna. arXiv preprint arXiv:2105.08559 (2021)

Appendix A Instructions for Converting to Another Scheme

Instruction 1 identifies and distinguishes the two different bits. In instruction 1, strand (S_1 1 2 3) is issued. In bit 0, the strand will displace the short strand over domains 2 and 3 but does not edit bit 1 since domain 1 is the only open domain for binding. In instruction 2, all domains in bit 1 are replaced by a single strand covering all domains with identifier S_a. Then in instruction 3, the strand S_1 is detached, so domains 1, 2, and 3 on bit 0 are exposed. In Instruction 4, all domains in bit 0 are replaced by a single strand covering all the domains with the identifier S_b. Then any encoding scheme with 7 domains in 1 cell could be written to the bits by first detaching strand S_a and writing the encoding for bit 1, then detaching strand S_b and writing the encoding for bit 0.

Appendix B Detailed Implementation of Each Step for Parallel Sorting

Here we give an instruction set for parallel binary bubble sort with the previously defined encoding scheme. We can implement each step of the sorting algorithm in 12 individual operations. Details of the design are shown in Figure B2.

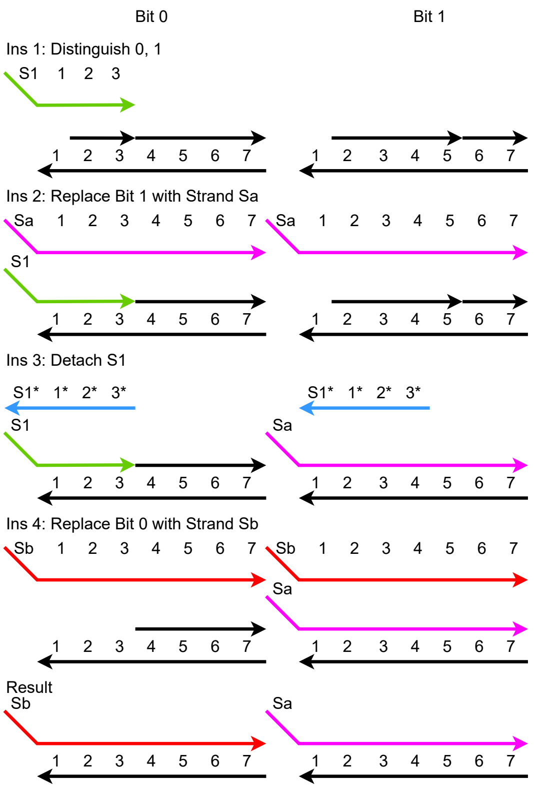


Fig. A1: Current coding scheme can be converted to another coding scheme

The 12 instructions fall into 2 stages. The first stage is “identifying.” During instructions 1-4, all the pairs (0, 1) are identified, and in both bit 0 and 1, a toehold is exposed for writing new data. More specifically, Instructions 1 and 2 identify the combination of $(1, 0)$. In instruction 1, (S_1 6 7 1 2 3) is issued to each pair of bits. In pair $(0, 0)$, S_1 and domains 6, 7 are exposed. In pair $(0, 1)$, since the only open domain is 1, it will not form a strong enough bond. In pair $(1, 0)$, only S_1 is exposed. In pair $(1, 1)$, S_1 and domains 2, 3 are exposed. In instruction 2, strand (6* 7* 1* 2* 3*) is issued to each pair of bits. Since pair $(1, 0)$ is the only pair that does not have exposure 5 or 2, this strand will detach strand S_1 in each pair except pair $(1, 0)$. After Instruction 2, the toehold between a bit value of 1 and a bit value of 0 in the pair $(1, 0)$ is replaced by a strand with an identifier of S_1. Instruction 3 seals off the domain exposed in the other pairs during Instruction 1 and 2 so that it will not be edited later. In instruction 4, the strand with identifier S_1 is detached, exposing domains 6 and 7 in the left cell containing bit 1, or domains 2 and 3, in the right cell containing bit 0. After this instruction, toeholds are exposed only in the 1s and 0s in pair $(1, 0)$. Other bits are not affected.

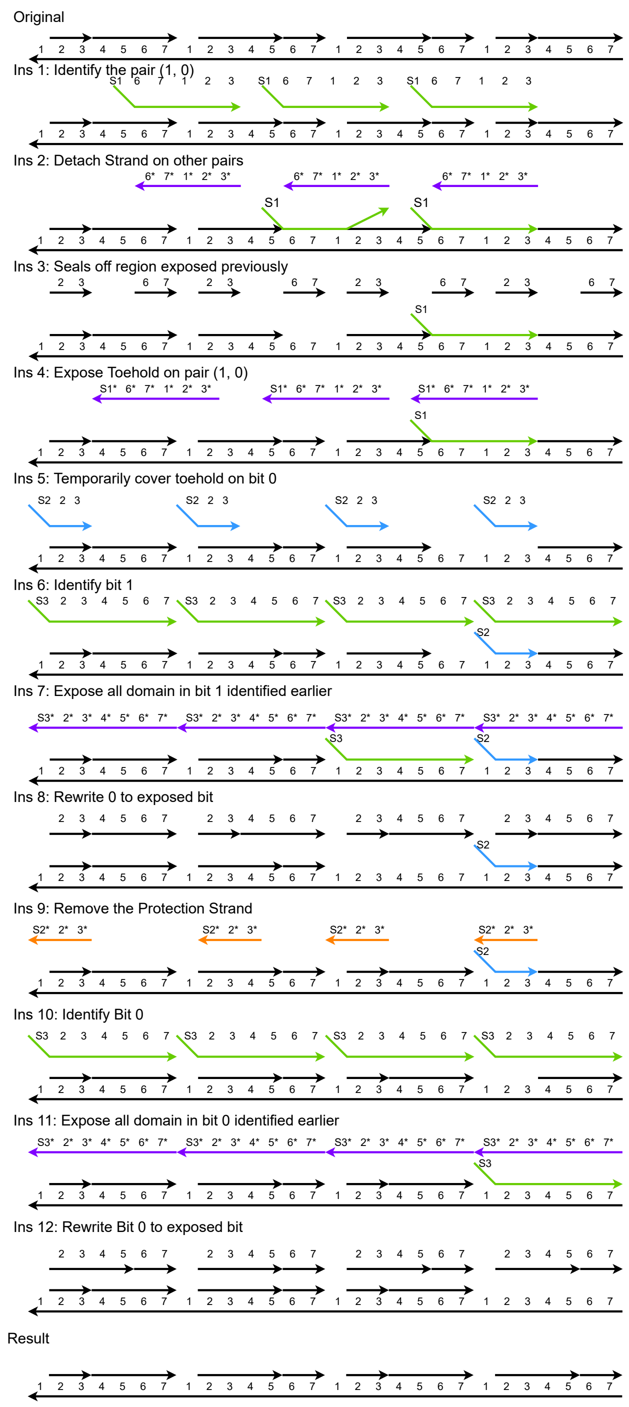

Original
Ins 1: Identify the pair (1, 0)
Ins 2: Detach Strand on other pairs
Ins 3: Seals off region exposed previously
Ins 4: Expose Toehold on pair (1, 0)
Ins 5: Temporarily cover toehold on bit 0
Ins 6: Identify bit 1
Ins 7: Expose all domain in bit 1 identified earlier
Ins 8: Rewrite 0 to exposed bit
Ins 9: Remove the Protection Strand
Ins 10: Identify Bit 0
Ins 11: Expose all domain in bit 0 identified earlier
Ins 12: Rewrite Bit 0 to exposed bit
Result
S1
S2
S3
S1* 6* 7* 1* 2* 3*
S2* 2* 3*
S3* 2* 3* 4* 5* 6* 7*

Fig. B2: Instructions for Parallel Sorting

The second stage is flipping the bits in the pair $(1, 0)$. In instruction 5, in the case of a bit value of 0, domains 2 and 3 are temporarily covered by a strand with identifier S_2 so that the writing process will not interfere with the identified 0s at this moment. In instruction 6, a bit value of 1 is replaced by a strand with identifier S_3 via the open toehold at domains 6 and 7. The strand is then detached in instruction 8, exposing all the domains of that bit. Then, the bit value of 0 is written to the location of a bit value of 1 in instruction 8. In instruction 9, the temporary cover for a bit 0 is lifted. Then, in instructions 10 through 12, a bit 1 is written to the location of a bit value of 0 using the same scheme as instructions 6 through 8. Throughout the process, only bits identified in the first stage with toeholds exposed are affected.

Appendix C Detailed Implementation of Each Step for Parallel Exclusive OR

The instructions are shown below, alongside an example of the Exclusive OR algorithm for sequence 11101 to 00000 in two iterations.

In each XOR iteration, the f(1,1) = (0,0) rewriting must be performed on non-overlapping pairs of bits. In the first iteration, the pairing is as follows: cell 0 with cell 1, cell 2 with cell 3, and so on. This means that all instruction strands only operate on these pairs. For this algorithm specifically, this can be achieved by using different sequences for the even versus the odd cells on the strand. In instruction 1, the strand (S_1 6 7 1 2 3) is issued to identify (1,0) pairs. In instruction 2, strand (6* 7* 1* 2* 3*) is issued to detach any S_1 strands with exposed domains of 6 and 7, or 2 and 3. In instruction 3, the strands (S_2 6 7 1) and (S_3 1 2 3) are issued to identify (1,1) and (0,0) pairs respectively. Finally, (0,1) pairs are identified with strand (S_4 4 5 6 7 1) for instruction 4. Now that all 1 domain toeholds are covered, strand (S_2* 6* 7* 1*) is issued in instruction 5 to detach all S_2 and expose (1,1) pairs. In instruction 6, strand (S_5 2 3 4 5 6 7 1 2 3 4 5 6 7) is issued to cover both cells in (1,1) pairs. Both S_5 and S_4 are now detached using strands (S_5* 2* 3* 4* 5* 6* 7* 1* 2* 3* 4* 5* 6* 7*) and (S_4* 4* 5* 6* 7*) in instruction 7. Then in instruction 8, all exposed cells are written to 0 using strands (2 3) and (4 5 6 7). In instruction 9, all S_1 and S_3 are detached using (S_1* 6* 7* 1* 2* 3*) and (S_3* 1* 2* 3*). By covering all exposed domains using strands (2 3) and (6 7) in instruction 10, all (1,1) pairs identified in the register are rewritten to (0,0) pairs. At this point, instructions 1-11 of the parallel sorting in section B are implemented to write all (1,0) pairs to (0,1). For these sorting steps, the cell pairing can be overlapping. The result of this whole iteration of the XOR algorithm is a DNA sequence that has the same bit parity as the input, but is more ordered (i.e., closer to being sorted), and contains the same or fewer 1's. In figure C3, the first iteration is carried out with non-overlapping pairs for cells 0 with 1, and so on. However, in figure C4, depicting a second iteration of the XOR algorithm, the pairing is: cell 1 with cell 2, cell 3 with cell 4, and so on. In the third iteration, the pairing can return to the original pairing in the

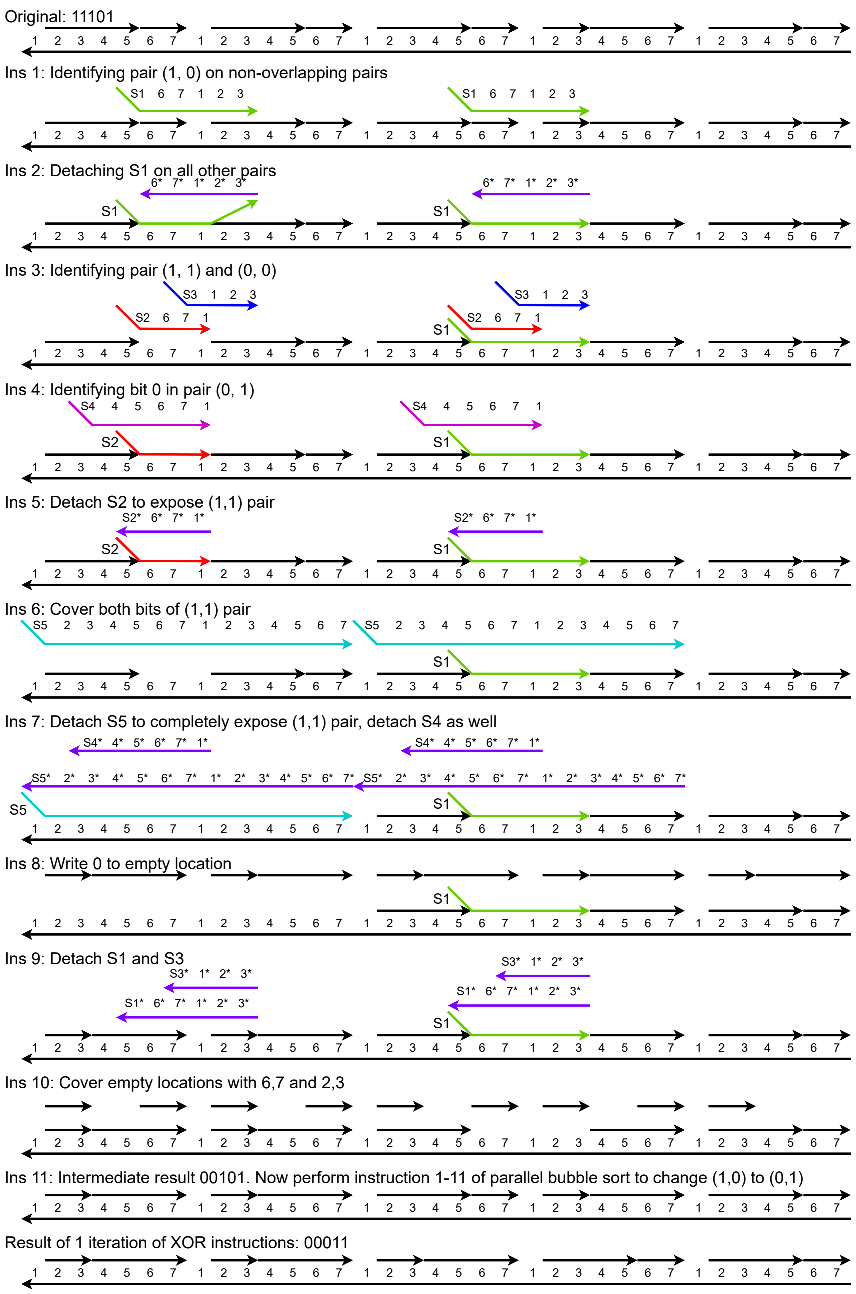

Original: 11101
Ins 1: Identifying pair (1, 0) on non-overlapping pairs
Ins 2: Detaching S1 on all other pairs
Ins 3: Identifying pair (1, 1) and (0, 0)
Ins 4: Identifying bit 0 in pair (0, 1)
Ins 5: Detach S2 to expose (1,1) pair
Ins 6: Cover both bits of (1,1) pair
Ins 7: Detach S5 to completely expose (1,1) pair, detach S4 as well
Ins 8: Write 0 to empty location
Ins 9: Detach S1 and S3
Ins 10: Cover empty locations with 6,7 and 2,3
Ins 11: Intermediate result 00101. Now perform instruction 1-11 of parallel bubble sort to change (1,0) to (0,1)
Result of 1 iteration of XOR instructions: 00011
S1
S2
S3
S4
S5

Fig. C3: Instructions for the Exclusive OR. The first iteration converts 11101 to 00011.

first iteration. For a n bit register, after n iterations of the XOR algorithm, the last cell contains the output of the n bit XOR.

Appendix D Detailed Implementation of Each Step for Parallel Left Shift cell

The instructions are shown as followed, with an example of shifting 11001 to 10011.

The first three instructions are exactly the same as those for identifying bit pairs in Section 3.1. In instruction 1, the strand (S_1 6 7 1 2 3), which identifies the different patterns of two bits, is issued to each pair of bits. In instruction 2, strand (6* 7* 1* 2* 3*) is issued, detaching strands with open domains 6 and 7, or 2 and 3. After this instruction, strands with identifier S_1 only remain at pair (1, 0). In instruction 3, we issue two species of strands at the same time: (S_2 6 7 1) and (S_3 1 2 3). (S_2 6 7 1) will bind with pair $(1, 1)$ and (S_3 1 2 3) will bind with pair $(0, 0)$. S_2 will not form a stable binding with pair $(0, 0)$ or $(0, 1)$ because the binding area is only one domain. Same goes with S_3 and pair $(1, 1)$ or $(0, 1)$. After this instruction, only domain 1 between pair $(0, 1)$ is still exposed. In instruction 4, strand (S_4 4 5 6 7 1) is issued. Through the open domain 1 between pair $(0, 1)$, the strand in bit 0 is replaced by S_4. After this step, the first bit in pair $(1, 0)$ is identified with the strand S_1, and the first bit in pair (0, 1) is replaced with the strand S_4.

Instructions 5 to 9 rewrite the first bit in pair $(1, 0)$ to 0. In instruction 5, the strand S_1 is detached, exposing domains 6, 7, 1, 2 and 3. The exposed domains 2 and 3 are sealed off in instruction 6 to not interfere with subsequent instructions. In instruction 7, strand (S_5 2 3 4 5 6 7) is issued through the open toehold on domains 6 and 7 in the bit 1 in pair $(1, 0)$, and displaces the strand in that bit. Since domains 2 and 3 are sealed off, bit 0 will not be modified in this instruction. In instruction 8, strand S_5 is detached, leaving the domains in the bit open. In instruction 9, strands (2 3) and (4 5 6 7), which represent 0, are written to the bit containing open domains.

In the final two instructions, we write 1 to the first bit in pair $(0, 1)$. In instruction 10, 3 strands are issued to each pair of bits: (S_2* 6* 7* 1*), (S_3* 1* 2* 3*) and (S_4* 4* 5* 6* 7* 1*). S_2, S_3 and S_4 are detached through these strands. Since S_4 covers the bit 0 in pair (0, 1), after this step, domains 3 and 4 are exposed in these bits, ready to be written to 1. In the final step, strands (2 3), (2 3 4 5), and (6 7) are issued to each cell. Strands (2 3) and (6 7) will fix the exposed domains from strand S_2 or S_3, and strand (2 3 4 5) will write bit 1 to the bit with domain 3 and 4 exposed. Details of the design are shown in Figure D5.

For all the pairs of $(0, 0)$ and $(1, 1)$, the first bit in those pairs will not be modified since the toehold 1 will be covered with S_2 or S_3 in the process.

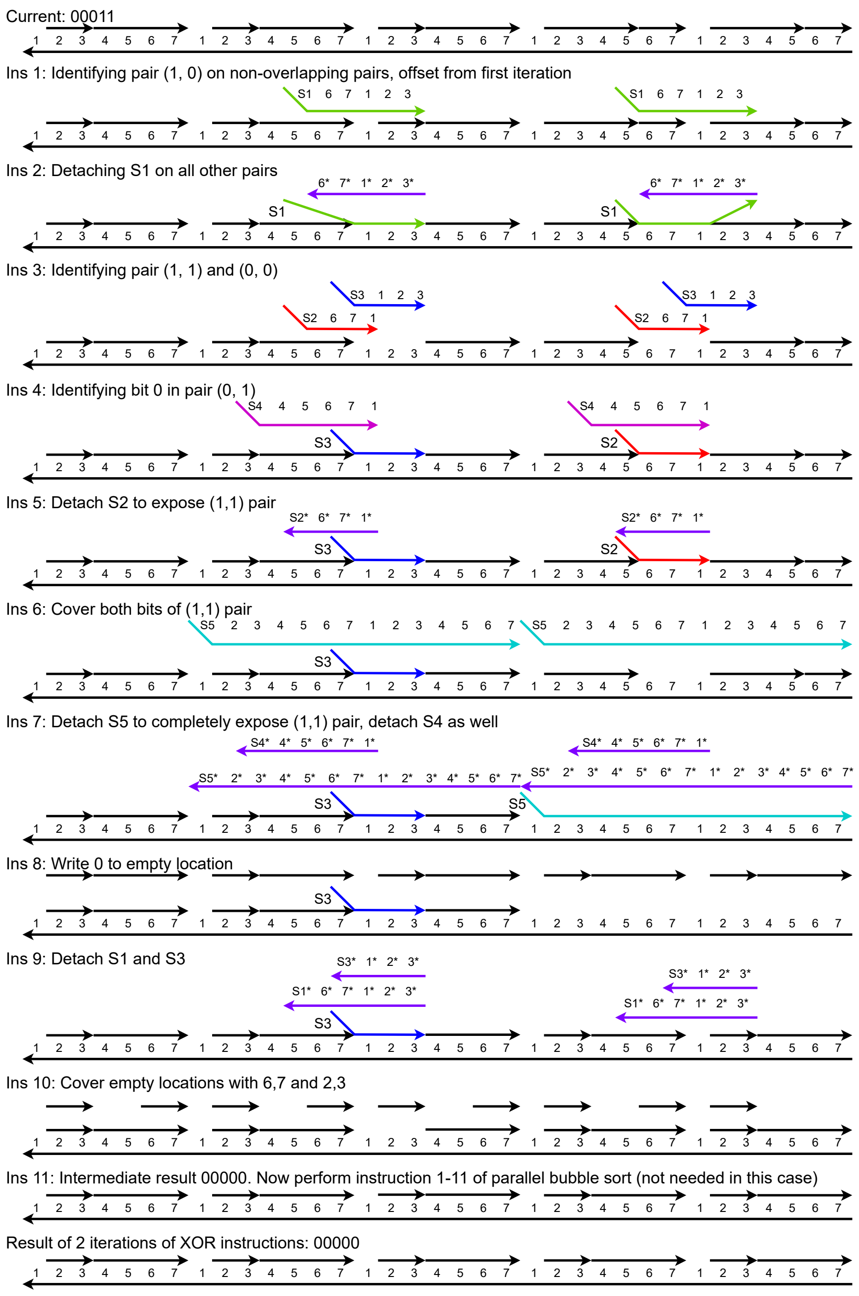

Current: 00011
Ins 1: Identifying pair (1, 0) on non-overlapping pairs, offset from first iteration
Ins 2: Detaching S1 on all other pairs
Ins 3: Identifying pair (1, 1) and (0, 0)
Ins 4: Identifying bit 0 in pair (0, 1)
Ins 5: Detach S2 to expose (1,1) pair
Ins 6: Cover both bits of (1,1) pair
Ins 7: Detach S5 to completely expose (1,1) pair, detach S4 as well
Ins 8: Write 0 to empty location
Ins 9: Detach S1 and S3
Ins 10: Cover empty locations with 6,7 and 2,3
Ins 11: Intermediate result 00000. Now perform instruction 1-11 of parallel bubble sort (not needed in this case)
Result of 2 iterations of XOR instructions: 00000
S1
S2
S3
S4
S5

Fig. C4: Instructions for the Exclusive OR. The second iteration converts 00011 to 00000.

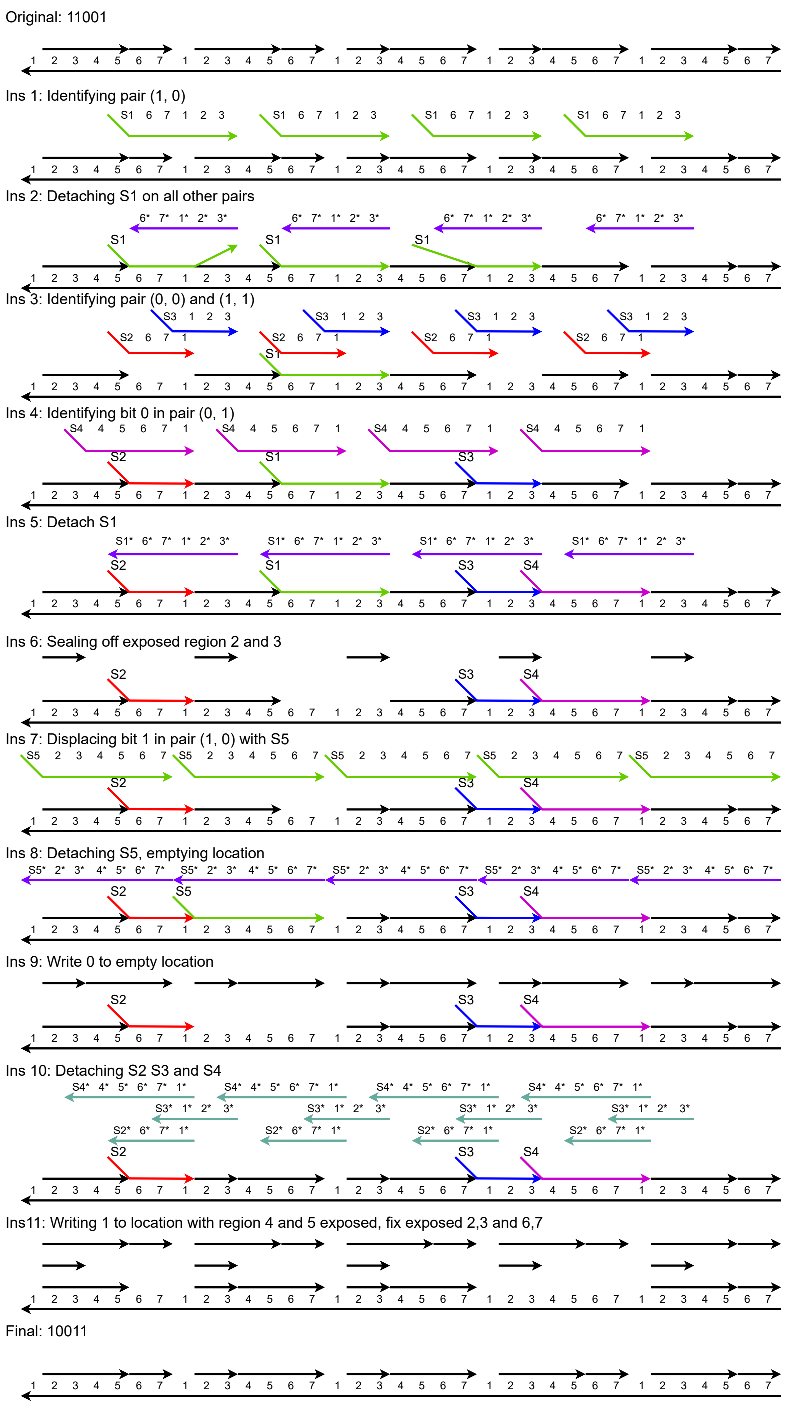

Original: 11001
Ins 1: Identifying pair (1, 0)
Ins 2: Detaching S1 on all other pairs
Ins 3: Identifying pair (0, 0) and (1, 1)
Ins 4: Identifying bit 0 in pair (0, 1)
Ins 5: Detach S1
Ins 6: Sealing off exposed region 2 and 3
Ins 7: Displacing bit 1 in pair (1, 0) with S5
Ins 8: Detaching S5, emptying location
Ins 9: Write 0 to empty location
Ins 10: Detaching S2 S3 and S4
Ins11: Writing 1 to location with region 4 and 5 exposed, fix exposed 2,3 and 6,7
Final: 10011
S1
S2
S3
S4
S5
1 2 3 4 5 6 7

Fig. D5: Instructions for the Left Shift cell

Appendix E Detailed Implementation of the Second Level in Parallel Search

Here we discuss the *second* level of the parallel search operation. The first level of the search operation uses the instructions that were described in Section 3.1, except we now only issue strands to non-overlapping bit pairs. We use identifiers $A_0 = 00, A_1 = 01, A_2 = 10, A_3 = 11$ to represent symbols in this level. For instance, to search for the target string 1011, we search for the symbol A_2 in odd symbols and A_3 in even symbols. The cases of A_2 in even symbols and A_3 in odd symbols are covered by searching with an offset.

In the first instruction of the second level, we uncover the A_2 in the odd symbols, creating an open region. In instruction 2, we use a long strand to cover the entire right half of the symbol, from the start of identifier A_2 to the rightmost cell. This strand is pulled out in instruction 3. In instruction 4, we use an identifier A_2' to cover domains 5, 6, 7 in the rightmost cell while covering all other domains.

Instructions 5 to 8 are essentially the same as instructions 1 to 4, but with two significant differences. Firstly, since A_3 is the second symbol in the current level of query, we only search for even-numbered symbols (2, 4, 6, etc.). Secondly, instead of rewriting the right half of the symbol, we write the left half. We make the new identifier A_3' to cover domains 2, 3, 4 in the left-most cell. In instruction 9, we use identifier (B_{11} 5 6 7 1 2 3 4) to recognize the two consecutive symbols A_2 and A_3. Since, in the regular encoding, no strand starts from domain 5 or ends at domain 4, it will only form a perfect binding with a matched result.

After the identifier B_{11} binds, we also need to clean up the imperfect bindings in case of a mismatch. Figure E6 shows the instructions for the cleanup process. In instruction 10, we first use the complementary strand (5* 6* 7* 1* 2* 3* 4*) to pull out the imperfect bond identifier B_{11}. Then we issue strands covering the exposed domain. We first issue strands covering fewer domains, then in following instructions, we issue strands covering more domains. As a result, we always obtain a perfect fit; the strands will not be pulled out in potentially unrelated rewriting processes.

Appendix F Example of Parallel Bubble Sort on an arbitrary bitstring

Consider the 12-bit long string $S = 110010010110$. In each iteration of bubble sort, we first identify all $(1, 0)$ pairs (shown in red) and then rewrite them to $(0, 1)$ (shown in blue). For this string, the numerous iterations of the sorting algorithm are:

$$110010010110 \rightarrow 101001001101$$
$$101001001101 \rightarrow 010100101011$$
$$010100101011 \rightarrow 001010010111$$

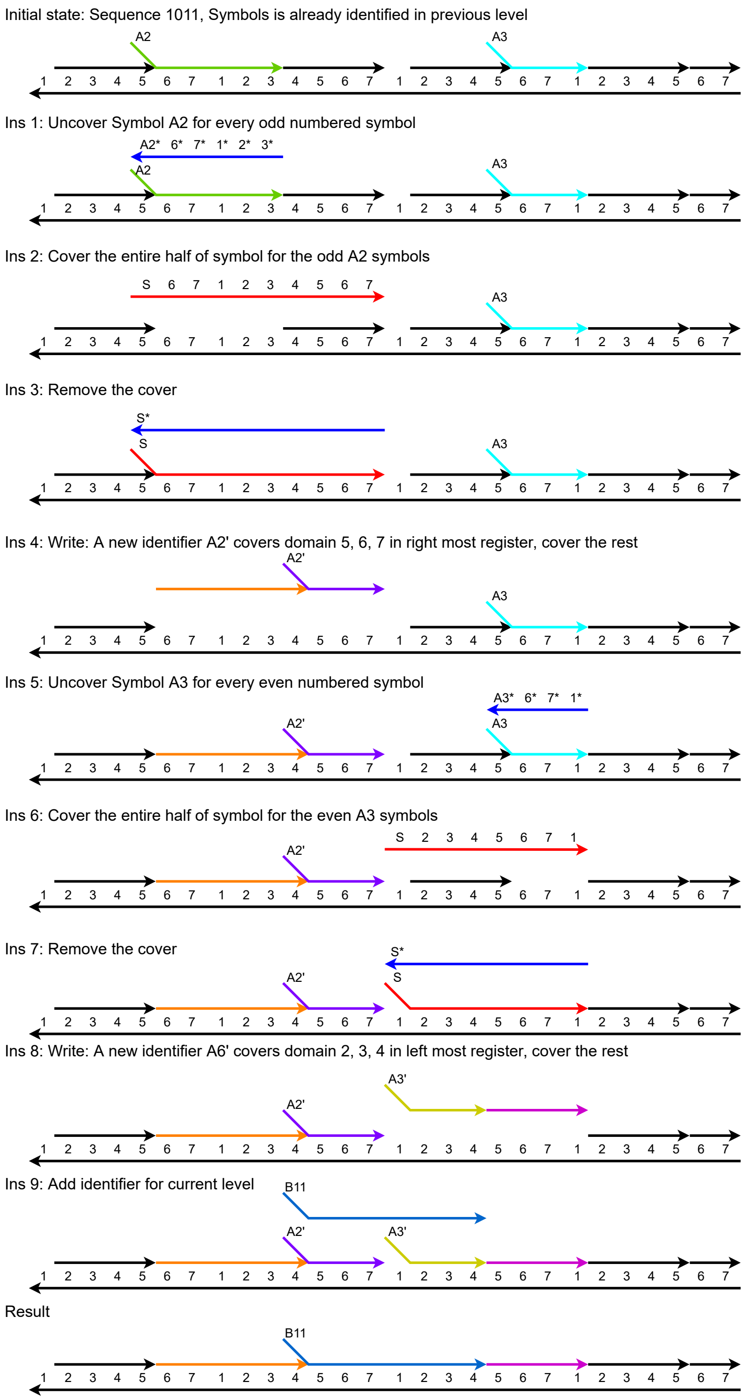

Initial state: Sequence 1011, Symbols is already identified in previous level
A2
A3
Ins 1: Uncover Symbol A2 for every odd numbered symbol
A2* 6* 7* 1* 2* 3*
A2
A3
Ins 2: Cover the entire half of symbol for the odd A2 symbols
S 6 7 1 2 3 4 5 6 7
A3
Ins 3: Remove the cover
S*
S
A3
Ins 4: Write: A new identifier A2' covers domain 5, 6, 7 in right most register, cover the rest
A2'
A3
Ins 5: Uncover Symbol A3 for every even numbered symbol
A3* 6* 7* 1*
A2'
A3
Ins 6: Cover the entire half of symbol for the even A3 symbols
S 2 3 4 5 6 7 1
A2'
Ins 7: Remove the cover
S*
A2'
S
Ins 8: Write: A new identifier A6' covers domain 2, 3, 4 in left most register, cover the rest
A3'
A2'
Ins 9: Add identifier for current level
B11
A2'
A3'
Result
B11
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7

Fig. E6: Instructions for a search operation of target sequence 1011

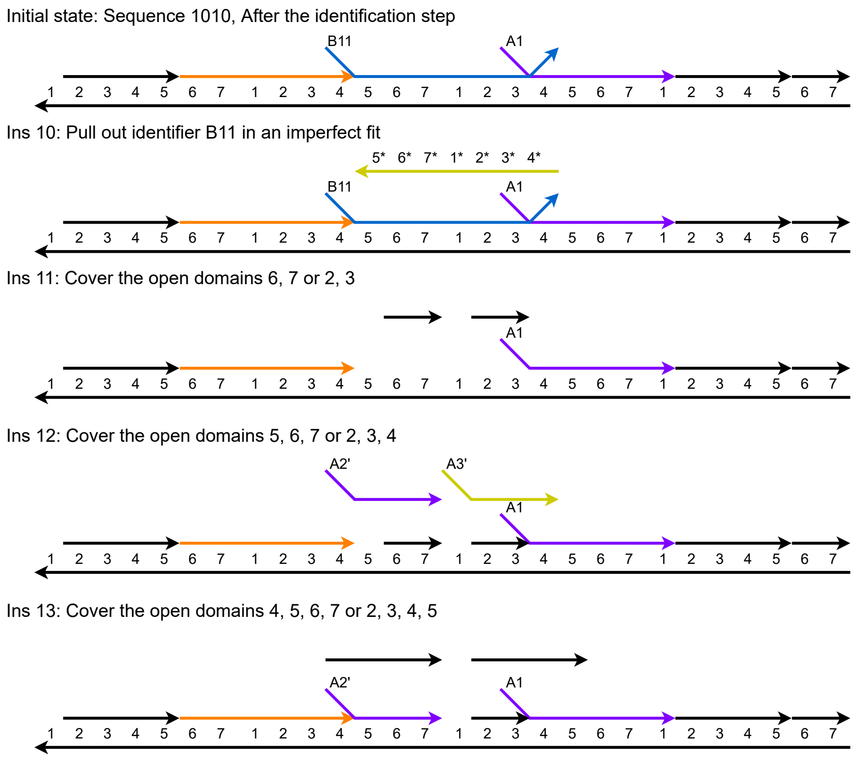

Initial state: Sequence 1010, After the identification step
B11
A1
Ins 10: Pull out identifier B11 in an imperfect fit
5* 6* 7* 1* 2* 3* 4*
Ins 11: Cover the open domains 6, 7 or 2, 3
Ins 12: Cover the open domains 5, 6, 7 or 2, 3, 4
A2'
A3'
Ins 13: Cover the open domains 4, 5, 6, 7 or 2, 3, 4, 5
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7

Fig. E7: Instructions for the cleanup process for a failed searching. These instructions won't affect the result of a successful search.

$$
\begin{aligned}
001010010111 &\rightarrow 000101001111 \\
000101001111 &\rightarrow 000010101111 \\
000010101111 &\rightarrow 000001011111 \\
000001011111 &\rightarrow 000000111111.
\end{aligned}
$$

After 7 iterations, the final sorted string is 000000111111.

Appendix G Simulating the XOR algorithm

We used the SIMD DNA simulator written by Dave Doty and Aaron Ong [19] to validate our XOR algorithm. We simulated two iterations of the XOR algorithm as shown in Figures C3 and C4. First, we operated on odd-to-even bit pairs on a strand storing 11101 to obtain 00101. Then we operated on even-to-odd bit pairs on the strand storing 00011 (the sorted result of the first iteration) to obtain the XOR 00000. To ensure non-overlapping pairing, we used different domain sequences for odd bits compared to even bits – odd bits were sequenced as domains 1 to 7, while even bits were sequenced as domains 8–14.

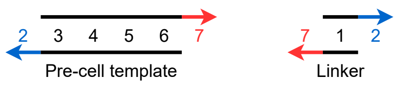
The simulator validated our algorithm: all instructions shown in Figures C3 and C4 simulated correctly. We have attached the simulation files and the predicted results for the two iterations in the supplementary data. These simulations show that the operation that rewrites non-overlapping $(1, 1)$ pairs to $(0, 0)$ preserves the parity of the register.

Appendix H Gibson Assembly of a 2 bit register

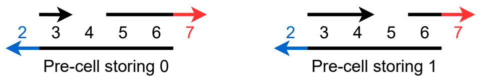
Gibson Assembly of DNA molecules is achieved through the use of "sticky ends" – single stranded sequences at the ends of these molecules that allow them to concatenate. To create registers storing unique bit sequences, we use two different molecules to start off: pre-cell molecules (domains 2 to 7, with sticky ends on domains 2 and 7), and linker molecules (domains 7 1 2, with sticky ends on domains 7 and 2). To store a bit value of 0 in a pre-cell, a toehold on domain 4 is created. To store a bit value of 1, a toehold on domain 5 is created. This is shown in Figure H8b.

Before concatenating two different pre-cells, their particular sticky ends must be "sealed" – those ends are no longer single stranded and cannot link together anymore. Sealing a particular sticky end can easily be done by adding a single strand of DNA that binds to that sticky end. For example, by sealing the sticky end on domain 2 of a pre-cell, that pre-cell can no longer concatenate with itself when the linker molecule is mixed. In Figure H8c, pre-cell A only has a sticky end on domain 7, and pre-cell B only has a sticky end on domain 7. When these pre-cells are mixed together with the linker molecule, they will bind to each other in the order A to B. This creates a pre-register of those two pre-cells. The starting end of the pre-register has a domain 1 concatenated through a "cap" molecule (domains 1 and 2, with a sticky end at domain 2) as shown in Figures H8e and H8f. After this stage, the pre-register can be treated with DNA ligase to seal all nicks. The resulting DNA strand contains the cells A and B which contain toeholds at domain 4 and 5 respectively. All 1 domains across all cells in this strand can be exposed into to toehold domains through nicking and gentle denaturing. Finally, this DNA molecule (which encodes 01 based on the pre-cell encoding scheme) can be converted to the bit encoding scheme used in this paper 2 through the procedure described in Sections 3.2 and 8.3. This entire procedure yields a 2 bit register storing the bits 0 and 1 in that order.

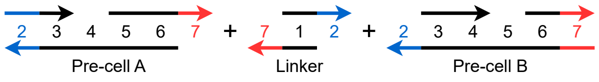
This approach can be used to construct registers of any arbitrary number of bits despite all cells having the same sequence. This is because pre-registers can also be concatenated in the same manner as pre-cells as shown in Figure H8c. For this, the sealed ends of a pre-register must be unsealed (through the use of an exonuclease) to create sticky ends again.



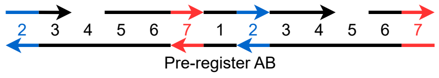
(a) The two main types of molecules used for Gibson Assembly



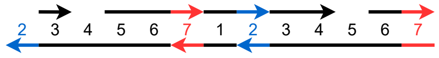
(b) How to store bit values 0 or 1 through toeholds 4 or 5 respectively.



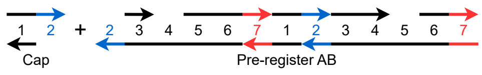
(c) Pre-cell A stores 0, Pre-cell B stores 1. They are mixed together with linker molecules to concatenate them. The blunt ends (domain 2 on A, domain 7 on B) prevent linking of two same pre-cells.



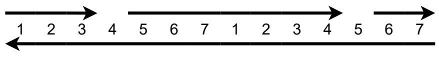
(d) The resulting pre-register AB.



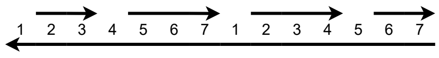
(e) Creating a sticky end on the first domain 2 of the pre-register.



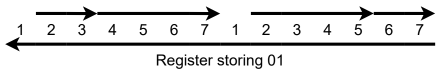
(f) Using a cap molecule to add a domain 1 at the start of the pre-register



(g) The resulting pre-register after ligation



(h) Toeholds are created on all 1 domains.



(i) The pre-cell encoding is changed to the encoding proposed in Figure 2.

Fig. H8: Using Gibson Assembly to construct a register storing 01 from cells with the same sequence.