Diverse and robust molecular algorithms using reprogrammable DNA self-assembly

– Supplementary Information A –

Damien Woods[†,*,1,2,3], David Doty[†,*,1,4], Cameron Myhrvold[5,6], Joy Hui[1,7], Felix Zhou[1,8], Peng Yin[5,6], Erik Winfree[*,1]

[†]Joint first co-authors, [*]Corresponding authors, [1]California Institute of Technology, Pasadena, CA 91125, USA. [2]Inria, Paris, France. [3]Current affiliation: Maynooth University, Maynooth Co. Kildare, Ireland. [4]University of California, Davis, CA 95616, USA. [5]Wyss Institute for Biologically Inspired Engineering, Harvard University, Boston, MA 02115, USA. [6]Department of Systems Biology, Harvard Medical School, Boston, MA 02115, USA. [7]Harvard University, Cambridge, MA 02318, USA. [8]University of Oxford, Oxford, OX1 2JD, UK.

Contents

S1 System design abstraction level: IBC model **4**
- S1.1 Mathematical definition of IBC model . . . 4
- S1.2 6-bit 1-layer IBC model . . . 8
- S1.3 Computational power of IBC model . . . 9
- S1.4 Discussion and open questions on the theory of IBCs . . . 21

S2 System design abstraction level: abstract tile assembly model and proofreading **23**
- S2.1 Definition of Cylindrical Tile Assembly Model . . . 23
- S2.2 Complete 6-bit IBC tile set before proofreading . . . 24
- S2.3 Complete 6-bit IBC tile set after 2×2 proofreading . . . 25

S3 System design abstraction level: binding-domain schematics **32**
- S3.1 Strand-level system design of SST lattice, input-adapter strands, seed attachment . . . 32
- S3.2 DNA origami seed design . . . 33
- S3.3 Input-adapter strands design . . . 38

S4 System design abstraction level: DNA sequence design **42**
- S4.1 Random sequences are not sufficient for algorithmic self-assembly . . . 42
- S4.2 Energy-based DNA sequence design . . . 42
- S4.3 Analysis of designed DNA sequences . . . 45

S5 Justification for design decisions **52**
- S5.1 Seeded growth of SST nanotubes . . . 52
- S5.2 4-bit copying tile set . . . 55
- S5.3 Unzippers . . . 55
- S5.4 Guard strands . . . 62
- S5.5 Biotin-streptavidin labelling . . . 64
- S5.6 Repeating tile types along circumference . . . 66
- S5.7 Comparison of DX, TX, and SST motifs for algorithmic self-assembly . . . 68

S6 Algorithmic self-assembly experiments **70**
- S6.1 Preparation of DNA strand stocks and reagent stocks . . . 70
- S6.2 Preparation of samples for execution of a 6-bit circuit computation . . . 71

S6.3 Executing a 6-bit circuit computation . . . 73
S6.4 Observing the result of a 6-bit circuit computation by AFM imaging . . . 75

S7 Analysis of AFM images **77**
S7.1 Preprocessing of AFM images . . . 77
S7.2 Annotating of objects in AFM images . . . 77
S7.3 Example wide-field AFM images . . . 79
S7.4 Imaging problems . . . 80
S7.5 Growth problems . . . 84
S7.6 Error rate estimates from physical principles . . . 85

S8 Circuits implemented: design, results, testing, analysis **88**
S8.1 Circuit: Sorting . . . 90
S8.2 Circuit: Parity . . . 92
S8.3 Circuit: Rule110 . . . 95
S8.4 Circuit: MultipleOf3 . . . 96
S8.5 Circuit: FairCoin . . . 98
S8.6 Circuit: Zig-Zag . . . 104
S8.7 Circuit: Palindrome . . . 106
S8.8 Circuit: Recognise21 . . . 108
S8.9 Circuit: Copy . . . 110
S8.10 Circuit: Cycle63 . . . 113
S8.11 Circuit: DiamondsAreForever . . . 114
S8.12 Circuit: Waves . . . 116
S8.13 Circuit: Rule110Random . . . 118
S8.14 Circuit: Rule30 . . . 119
S8.15 Circuit: Drumlins . . . 120
S8.16 Circuit: 2Eggs . . . 121
S8.17 Circuit: LazySorting . . . 122
S8.18 Circuit: LazyParity . . . 125
S8.19 Circuit: RandomWalkingBit . . . 128
S8.20 Circuit: AbsorbingRandomWalkingBit . . . 130
S8.21 Circuit: LeaderElection . . . 132

References **134**

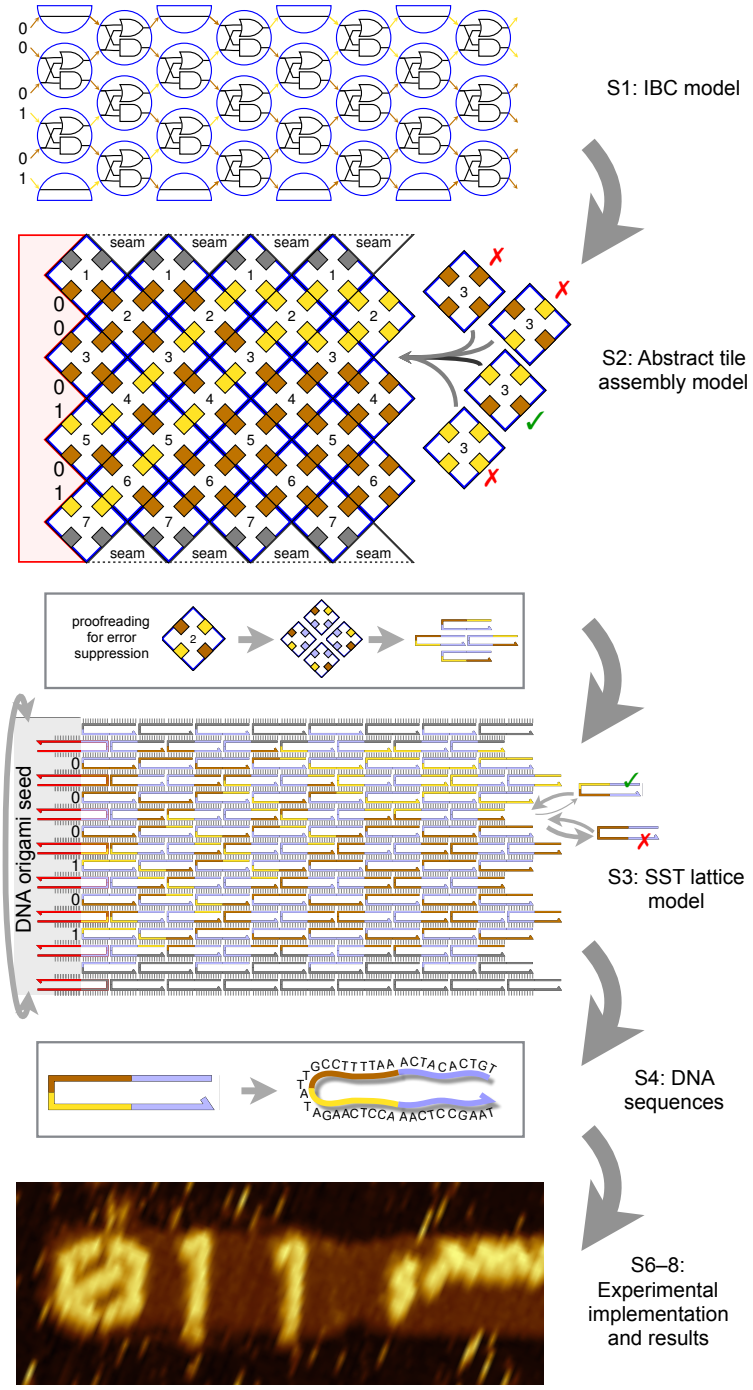


Figure S1: A visual table of contents and simple illustration of our abstraction hierarchy for designing and implementing a single circuit (Sorting on input 000011, with seed barcode 011). Section numbers (S1, S2, etc.) are shown on the right.

S1 System design abstraction level: IBC model

In this section we mathematically define the iterated Boolean circuit (IBC) model and then go on to analyse its computational power by stating and proving a few theorems. This section can be read independently of the other sections in the SI. Throughout Section S1, a word in *emphasis* means that that word is being defined.

The IBC model is intended as an abstract model for the specific type of computation by molecular tile self-assembly that is presented in this work. As such, it is closely related to abstract tile assembly models, whose capabilities for universal computation, universal construction, and intrinsically universal simulation are well known [18, 19, 35, 3]. As in those systems, an important complexity measure is the number of tile types, which scales linearly with the number of gate positions in an IBC. An n-bit ℓ-layer IBC has $O(n\ell)$ gate positions. This can be considered analogous to program-size complexity or a circuit-size complexity [20]. Another important complexity measure is the size to which an assembly must grow before obtaining an answer, that is, the number of tiles in a sufficiently grown assembly. This scales with the number of iterations needed to obtain the answer times the size of each iteration, that is, $O(n\ell i)$ for an IBC that obtains an answer within i iterations. This can be considered analogous to a space complexity. The final complexity measure to keep in mind is the length of the assembly needed to obtain an answer, which would be $O(\ell i)$, and can be considered a time complexity. Our overall question is how efficiently the IBC model can use these resources – program size, space, and time – for performing computation.

In Section S1.1 we mathematically define the n-bit ℓ-layer IBC model and in Section S1.2 we list a few observations about the 6-bit 1-layer IBC model implemented experimentally in this work. In Section S1.3 we mathematically characterise the general-purpose and computational capabilities of the IBC model, by proving that instances of the model simulate computations by Turing machines [36], Boolean circuits and branching programs [37]. The model can also be considered a subclass of bit-level systolic arrays [38] because of the locally connected planar lattice geometry, and (in a different sense) as a subclass of Boolean Automata Networks [39] because of the focus on iteration. Section S1.4 gives some future work directions on the IBC model.

S1.1 Mathematical definition of IBC model

Here we define the IBC model, using only elementary concepts (such as set, finite tuple/ordered list, infinite sequence, function, composition of functions, and random variable). The goal is to mathematically define the (more intuitive) description of the IBC model given in Figure 1b in the main text. More specifically, the goal of this subsection is to define the terms "deterministic n-bit, ℓ-layer circuit" and "randomised n-bit, ℓ-layer circuit"; i.e. generalisations of the 6-bit 1-layer IBCs implemented in our experimental work.

Preliminary notation and definitions. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of natural numbers, let $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$ be the set of strictly-positive natural numbers, and let $2\mathbb{N}^+ = \{2, 4, 6, \ldots\}$ be the set of strictly-positive even natural numbers. We call the set $\{0, 1\}$ the *binary alphabet* with two symbols, and we let $\{0, 1\}^n$, $n \in \mathbb{N}$, be the set of binary *words* of length n (there are (2^n) of them; for $n = 0$ this includes the length-zero, or empty, word), and $\{0, 1\}^* = \bigcup_{n\in\mathbb{N}}\{0, 1\}^n$ be the set of all binary words of any finite length from $\mathbb{N}$ (there are infinitely many, including the empty word).

Intuitively, a decision problem is a set of questions that have yes/no answers. Mathematically, a *decision problem*, also called a *language*, is a set of binary words $L \subseteq \{0, 1\}^*$. An algorithm A (for some formal model of computation such as a Turing machine [36, 20]) is said to solve the decision problem L, or decide the language L, if when given $x \in \{0, 1\}^*$ as input, A outputs 'yes' if $x \in L$ and 'no' otherwise.

Gate. To define IBCs, we first define what a gate is. There are two kinds of gates, two-input gates and one-input gates, defined as follows. A *two-input (two-output) gate*, or simply a *gate*, g, is a pair (f, r), where $f : \{0, 1\}^2 \to \{0, 1\}^2$ is the *function computed by* g and $r \in \mathbb{N}^+$ is the *position*. For notational convenience, we write g_r to denote a gate whose position is r and identify g_r with the function it computes, writing $g_r(i_{\text{top}}, i_{\text{bot}})$ to denote $f(i_{\text{top}}, i_{\text{bot}})$ for $i_{\text{top}}, i_{\text{bot}} \in \{0, 1\}$. We define such functions using a *truth table*, which

is a lexicographically ordered table of the four input-output pairs for the gate[1]. A *one-input (one-output) gate* is defined similarly, but the function it computes is of the form $f : \{0,1\} \to \{0,1\}$. The truth table for g_r has two rows.[2]

Circuit layer. For $n \in 2\mathbb{N}^+$, an n-bit circuit layer consists of $n+1$ gate positions $\{1, 2, \ldots, n+1\}$, where each gate position has an associated set of (gate, probability) pairs such that the sum of the probabilities in each gate position set is exactly 1. This is formalised as follows. For a gate position $r \in \{1, 2, \ldots, n+1\}$ we define the *gate set at position* r, or G_r, to be a set of pairs

$$G_r = \left\{ \left(g_r^{(1)}, p_r^{(1)}\right), \left(g_r^{(2)}, p_r^{(2)}\right), \ldots, \left(g_r^{(m)}, p_r^{(m)}\right) \right\} \tag{1}$$

where, $j \in \{1, 2, \ldots m\}$, $g_r^{(j)}$ is a gate, $m \in \mathbb{N}^+$, and $p_r^{(j)} \in [0,1] \subsetneq \mathbb{R}$ is a *probability* from the closed unit interval over the set of real numbers $\mathbb{R}$ such that $\sum_{j=1}^{m} p_r^{(j)} = 1$. An *$n$-bit circuit layer* c is a tuple (ordered list) of $n+1$ gate sets, one for each gate position:

$$c = (G_1, G_2, \ldots, G_{n+1})$$

where G_1 and G_{n+1} contain (gate, probability) pairs with one-input gates and $G_2, G_3, \ldots, G_n$ contain (gate, probability) pairs with two-input gates. In a *deterministic n-bit circuit layer*, each of the $n+1$ gate sets contains exactly one (gate, probability) pair with each probability component being 1, i.e. for all $r \in \{1, 2, \ldots, n+1\}$ we have $G_r = \{(g_r^{(1)}, 1)\}$. In a *randomised n-bit circuit layer*, at least one of the $n+1$ gate sets contains multiple, i.e. two or more, (gate, probability) pairs[3].

Circuit. For $\ell \in \mathbb{N}^+, n \in 2\mathbb{N}^+$, we define an n-bit, ℓ-layer IBC circuit C to be an ordered tuple (or list) of ℓ circuit layers

$$C = (c_1, c_2, \ldots, c_\ell) \ .$$

The IBC C is said to be deterministic if all elements (layers) of C are deterministic, otherwise C is said to be randomised.[4]

Function(s) computed by a circuit layer c. An *instance* I of a circuit layer c, is defined to be a tuple of $n+1$ gates,

$$I = (g_1, g_2, \ldots, g_{n+1}) \tag{2}$$

where for each $r \in \{1, 2, \ldots, n+1\}$, g_r is the gate component of a tuple chosen from the set G_r, which in turn is from the layer c (i.e., $g_r = g_r^{(j)}$ for some j where $\left(g_r^{(j)}, p_r^{(j)}\right) \in G_r$). Intuitively, a circuit layer instance is a list of gates, whereas a circuit layer is a list of *sets of potential* gates. Since a deterministic n-bit circuit layer has sets of size 1, there is exactly one choice for I, whereas for a randomised circuit layer there is more than one choice, hence multiple possible circuit layer instances.

See Figure S2 for an example of the names of some variables defined below. The circuit layer instance I computes the *circuit layer function* $f_I : \{0,1\}^n \to \{0,1\}^n$, defined as follows:

$$f_I(x_1, x_2, \ldots, x_n) = y_1, y_2, \ldots, y_n$$

[1]Although we here formally define gates using truth tables, in the figures in the main text and SI we use the standard (and equivalent) pictorial notation for AND, OR, XOR, NOT and constant (0/1) Boolean gates to define two-input and one-input gates. For example, the two-input gate in Figure 1b (zoom-in inset) in the main text is defined both pictorially and by its truth table.

[2]Our circuits act on bits $\{0,1\}$, in other words on an *alphabet* of size two. It is straightforward to generalise our circuits to larger alphabets of any size $k \in \{3, 4, \ldots\}$. This would in turn increase the number of functions computable by our circuits. In terms of implementation in DNA SSTs it would simply correspond to designing a larger set of DNA tile-strand sequences than for the $k = 2$ case explored in this work, something we believe would be an interesting direction for future work.

[3]Excluding deterministic circuits from the class of randomised circuits raises the question of whether there are things a deterministic circuit can do that a randomised circuit cannot do. However, randomised gate probabilities may be 0 or 1, which means that although formally different, a randomised circuit can be designed to exactly emulate a deterministic circuit.

[4]Intuitively speaking, in order to "program" an *n-bit circuit*, the programmer specifies an n-bit, ℓ-layer circuit C. For our experimental results in this paper we used only 6-bit, 1-layer circuits hence to program those circuits one specifies a single circuit layer. Implementing circuits with several layers out of DNA SSTs would require designing a larger set of sequences (domains) than for the single-layer case explored in this work, something we believe would be an interesting direction for future work.

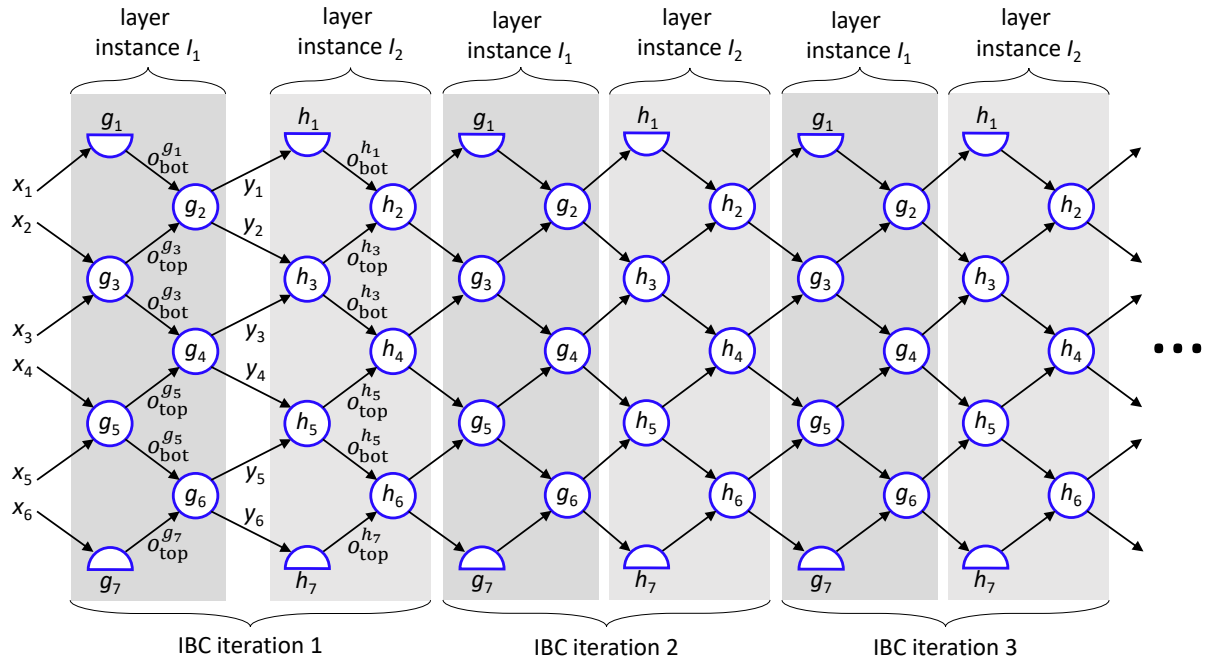


Figure S2: Illustration of 6-bit, 2-layer deterministic IBC naming convention used in the mathematical definition of the model. Example shows three IBC iterations consisting of two layers each, the first with gates named g_r for $r \in \{1, 2, \ldots, 7\}$, and the second with gates named h_r, with names of "wires" between gates shown in first iteration. In a randomised circuit, rather than a single g_r for position r, we instead choose $g_r^{(j)}$ for some $j \in \{1, 2, \ldots, m\}$, where m is the number of gate choices for position r, with a particular j chosen for each layer instance. Input bits $x_1, x_2, \ldots, x_n$ and output bits $y_1, y_2, \ldots, y_n$ are shown for function f_{I_1} computed by the first layer.

where for even $i \in \{2, 4, \ldots, n\}$ (and recall that n is also even):

$$y_{i-1}, y_i = g_i(o_{\text{bot}}^{g_{i-1}}, o_{\text{top}}^{g_{i+1}})$$

such that $o_{\text{bot}}^{g_1} = g_1(x_1)$ and $o_{\text{top}}^{g_{n+1}} = g_{n+1}(x_n)$ and for odd r in the range $r \in \{3, 5, \ldots, n-1\}$

$$o_{\text{top}}^{g_r}, o_{\text{bot}}^{g_r} = g_r(x_{r-1}, x_r) \ .$$

When we say $f : \{0,1\}^n \to \{0,1\}^n$ *is a function computed by a circuit layer* c we mean that f is a function computed by a circuit layer instance of c. Hence, each c has an associated set of functions, one function for each of its circuit layer instances, and we let $\mathcal{F}_c$ denote the set of such functions.

$$\mathcal{F}_c = \{f_I \mid f_I \text{ is a function computed by } c\}$$

For a deterministic layer c, since there is only one choice of (gate, probability) pair per gate position, $\mathcal{F}_c$ contains a single function, while for a randomised circuit layer $\mathcal{F}_c$ contains several functions, albeit a finite number. For example, for the circuit DiamondsAreForever (see Section S8.11) $\mathcal{F}_c$ contains two functions and for LeaderElection (see Section S8.21) $\mathcal{F}_c$ contains $2^5 = 32$ functions.

Function(s) computed by a single iteration of a circuit C. We say that f is a function computed by a single iteration of an ℓ-layer circuit $C = (c_1, c_2, \ldots, c_\ell)$ if

$$f = f_{c_\ell}(f_{c_\ell - 1}(\cdots f_{c_1}(x)))$$

where for each $i \in \{1, 2, \ldots, \ell\}$, f_{c_i} is a function computed by the circuit layer c_i. We let

$$\mathcal{F}_C = \{f \mid f \text{ is a function computed by a single iteration of an } \ell\text{-layer circuit } C\}$$

denote the set of such functions.

Circuit computation. For an n-bit circuit $C = (c_1, c_2, \ldots, c_\ell)$, and a length-$n$ word $x_0 \in \{0,1\}^n$ (called the *input*), a *circuit computation* is an infinite sequence of n-bit words

$$X = x_0,\ f_1(x_0),\ f_2(f_1(x_0)),\ f_3(f_2(f_1(x_0))),\ \ldots$$

where for all $t \in \mathbb{N}^+$, $f_t \in \mathcal{F}_C$ and where $\mathcal{F}_C$ is the set of functions computed by a single iteration of C.

If C is deterministic, then for each input x_0 there is a single computation X, and if C is randomised then for each input x_0 there are (possibly) multiple such X.

Intuitively, a randomised circuit computation is obtained by choosing a function randomly from $\mathcal{F}_C$ at each iteration $t \in \mathbb{N}$ according to the gate probabilities. We formalise this as follows. Suppose $I = \left(g_1^{(i_1)}, g_2^{(i_2)}, \ldots, g_{n+1}^{(i_{n+1})}\right)$ is an instance of a circuit layer c as per Equation (2), and recall from Equation (1) that each gate $g_r^{(i_r)}$, has an associated probability $p_r^{(i_r)}$. Since each randomised gate is sampled independently, we define the *probability of* I to be the product of the probabilities of each of its gates:

$$\text{Prob}[I] = \prod_{r \in \{1,2,\ldots,n+1\}} p_r^{(i_r)}$$

where, $p_r^{(i_r)}$ is the probability associated to gate $g_r^{(i_r)}$. It can be shown that $\sum_{f_I \in \mathcal{F}_c} \text{Prob}[I] = 1$. For a circuit layer instance I, we define the probability of function f_I to be $\text{Prob}[f_I] = \text{Prob}[I]$. The probability of a function $f \in \mathcal{F}_C$ where $f = f_{c_\ell}(f_{c_\ell - 1}(\cdots f_{c_1}(x)))$ is then defined to be

$$\text{Prob}[f] = \text{Prob}[f_{c_\ell}] \times \text{Prob}[f_{c_\ell - 1}] \times \cdots \times \text{Prob}[f_{c_1}]$$

A *randomised circuit computation* of circuit C on input x is defined to be a random variable whose value is the computation of C on input x when each iterated layer's function is chosen according to $\text{Prob}[f]$ as above.

This completes the definition of the IBC model.

S1.1.1 Output conventions for circuit computation

Observe that since circuit layer functions computed by n-bit IBCs have a finite domain and range (each of size at most 2^n), and since each IBC computation X is infinite, it follows that every deterministic IBC computation X must eventually enter a cycle: a repeated finite sequence of words[5]. It also follows that each deterministic IBC has a finite number of distinct disconnected cycles. Each input x is either already on one of these cycles, or eventually enters exactly one of them.

A randomised circuit C possibly avoids such cycles, but some randomised circuits such as LazySorting have the property that for *some* inputs x, there is only deterministic computation possible forward from there (any input that is already sorted). In this case we say that C is *deterministic on* x. and furthermore that with probability 1 the computation on any input eventually enters a deterministic cycle.

For an input $x \in \{0,1\}^n$ such that circuit C is deterministic on x, let $c(x) = (x_1, x_2, \ldots, x_k)$ for $k \in \mathbb{N}^+$ denote the cycle of n-bit words that C eventually enters starting from input x. For any computation X (whether from a deterministic or randomised circuit), we say X *eventually enters* cycle c if X has a finite initial portion followed by infinitely many repetitions of c.

The next two definitions are based on using such cycles to define what it means for IBCs to compute functions and decide problems.

Computing functions. Here we give a definition[6] of what it means for an n-bit IBC C to compute a function $F : \{0,1\}^n \to \{0,1\}^n$ on input $x \in \{0,1\}^n$. Intuitively, the IBC eventually outputs the word $F(x) \in \{0,1\}^n$, and then repeats that word on all future iterations.

An n-bit, ℓ-layer deterministic IBC C *computes the function* $F : \{0,1\}^n \to \{0,1\}^n$ if for each $x \in \{0,1\}^n$, $c(x) = (F(x))$, i.e., C eventually enters a cycle of length 1 with just the word $f(x)$ in the cycle, meaning that x is a fixed point of F: $x = f(x)$. If C is randomised, we say C *computes the function* F if, for each

[5] I.e. there are $i, c \in \{0, 1, \ldots 2^n - 1\}$ such that X contains the infinite suffix $x_i, x_{i+1}, \ldots, x_{i+c}, x_i x_{i+1}, \ldots, x_{i+2c}, x_{i+2c+1}, \ldots$.

[6] Note that this is different from the earlier definition of function computation by a single layer of the IBC; we now want to make sense of the notion that, although the IBC repeats layers forever, we can identify a single output after finite time.

$x \in \{0,1\}^n$, with probability 1, C's computation on x contains a word x' such that C is deterministic on x' and $c(x') = (F(x))$. (For example, for LazySorting, for any input x, the word x' reached with probability 1 is the bits of x rearranged with 1s first, then 0s.)

Deciding languages/problems. We wish to define what it means for an IBC to solve a decision problem (the terms decision problem and language were defined at the beginning of Section S1.1).[7]

We say that an n-bit deterministic IBC C *decides*, or *cycle-decides*, the language $P \subseteq \{0,1\}^n$ if C has exactly two cycles c_0 ("reject cycle") and c_1 ("accept cycle") such that for any input $x \in \{0,1\}^n$, if $x \in P$ then $c(x) = c_1$, and $c(x) = c_0$ otherwise. A randomised IBC C *decides* P if, for all $x \in \{0,1\}^n$, if $x \in P$ then with probability 1, the computation of C eventually enters c_1, and otherwise C enters c_0 with probability 1.

Note that for computation of both functions and decision problems, a deterministic IBC computing the function/problem, when viewed as a randomised IBC, computes it under the randomised definition as well, since the deterministic IBC has a single computation (necessarily followed with probability 1) that satisfies the property for the randomised definition.

S1.2 6-bit 1-layer IBC model

This subsection contains some observations on the 6-bit 1-layer IBC model that we have implemented experimentally. Note that Section S1.4 lists some open questions and direction for future work on the IBC model.

Computational capabilities. Sections S8.1–S8.21 give twenty-one 6-bit 1-layer IBCs, showcasing some of the abilities of this class of circuits. These circuits decide languages, compute functions, and generate randomised and deterministic patterns. In Section S1.1.1 we defined what it means for an IBC to decide a language; using that definition the following experimentally-implemented IBCs each decide a language (of 6-bit words): MultipleOf3, Parity, Palindrome, Recognise21 and LazyParity. The circuits Sorting, Copy and LazySorting compute 6-bit functions, via the notion of computing a function defined in Section S1.1.1. Other implemented circuits exhibit other recognisable kinds of behaviour, which do not correspond to our earlier definitions of computing a function or decision problem.[8] The randomised circuits FairCoin, RandomWalkingBit, AbsorbingRandomWalkingBit and LeaderElection each execute a randomised algorithm, while the circuits DiamondsAreForever and Waves generate randomised patterns. The following circuits generate deterministic patterns: Zig-Zag, Cycle63, Drumlins and 2Eggs. Finally, the circuits Rule110, Rule110Random and Rule30 simulate cellular automata. Taken together this suite of examples illustrate in an intuitive way that a wide variety of computational and expressive behaviours can be seen in IBCs.

Mathematically speaking, we take this a step further in Section S1.3 by proving theorems about the general purpose capabilities of IBCs.

Number of gates. Here we make a combinatorial observation: There are 1,288 gates for the 6-bit 1-layer IBC model. This number is calculated as follows. The truth-table for a 2-input, 2-output gate has 4 rows, two input columns, and two output columns (See Figure 1b in the main text for an example truth table). Since the inputs (rows) to a truth table are always written in lexicographical order (i.e. 00, 01, 10, 11 from top position to bottom position), to specify a gate one only needs to specify the 8 bits that appear in the two output columns. Hence there are $2^8 = 256$ gates that have 2 inputs and 2 outputs. For an n-bit IBC, where $n \geq 2$ is even, there are $n - 1$ gates that have 2 inputs and 2 outputs. Since each gate position has a

[7] As with other models of computation, we have some freedom to choose precisely what it means solve a decision problem or decide a language. For example, typically, a Turing machine [36, 20] M is said to solve a decision problem, if for any $x \in \{0,1\}^*$, when x is given as input to M, after a finite amount of time M halts in a special "accept" state if $x \in P$ and "reject" otherwise. But one also sees other (reasonable) definitions such as: the Turing machines accepts, respectively rejects, if it writes a 1, respectively 0, and immediately halts. Similarly, for IBCs, we have a lot of choices here; the main key point is to write down a definition that does not cheat by hiding too much computational power into input encodings and output decodings. The definition of "decides" we give here is used for IBCs we experimentally implemented, although in our proofs below we sometimes use other language decision definitions, such as the definition of *word-decides* in Section S1.3.1.3 used to interface with existing decidability definitions employed in elementary cellular automata [21].

[8] We omit formal definitions of some of these behaviours because we will not need to use them in our mathematical proofs.

set of gates that can be placed at that position and at no other, this gives a total of $256(n-1)$ gates with 2 inputs and 2 outputs, per n-bit IBC. In addition, an IBC has two single-input, single-output gate positions (the top and bottom gates of a circuit layer). Each such gate computes a function from one bit to one bit hence there are 4 such gates (constant 0, constant 1, identity, and negation). Thus, for even n, there are $256(n-1)+8$ gates for an n-bit 1-layer IBC. Setting $n = 6$, gives $1,288$ gates for the n-bit 1-layer IBCs studied in this paper.

For an n-bit ℓ-layer IBC, there are $256(n-1)\ell+8l$ distinct gates. Keep in mind that when implementing an IBC with tiles, fewer tiles are needed than possible gates, because each tile (prior to proofreading) represents an entry in the rule table for a gate at a particular position, and thus different gates can be implemented using different subsets of the same set of possible rule table entries.

S1.3 Computational power of IBC model

In this section we illustrate the general-purpose capabilities of the IBC model beyond the 21 examples we implemented. We do this by mathematically comparing the IBC model to other well-studied models of computation, such as Turing machines, cellular automata, and Boolean circuits [20]. Theorem S1.1 shows that IBCs efficiently implement any algorithm, more precisely single-tape Turing machines that run in t time-steps are simulated by $O(t^2 \log t)$-bit 1-layer IBCs via simulation of elementary cellular automaton rule 110 [21, 22]. Theorem S1.5 shows that arbitrary feedforward width-w depth-d Boolean circuits with 2-input gates are simulated by $O(w)$-bit $O(wd)$-layer IBCs. Finally, Theorem S1.7 shows that n-input, $O(\log n)$-depth Boolean circuits are simulated by 6-bit poly(n)-layer IBCs using a "global input" variant of the model, via simulation of length-poly(n) width-5 branching programs [37]. This section explains the meaning of these terms and how the results are obtained.

As is common in theoretical computer science, we formalise algorithms as "Turing machines". A Turing machine is a [36] model of computation that is powerful enough to execute any suitably formalised algorithm for a discrete computer [20]. In order to show that IBCs can run any algorithm, we will give a method that for any Turing machine tells us how to write down an IBC that emulates the Turing machine for inputs of a certain size.

Showing that one theoretical model of computation can perform the same tasks as another can be formalised as *simulation*, a standard technique in theoretical computer science to compare models of computation. The idea behind simulation is that if for every computation in system A, there is a computation in system B that in a well-defined sense acts similarly to the one in A, then B is said to simulate A. This means that if A can solve some problem, then B can too: B just needs to act like A to get the job done. Throughout this section we use the term *simulate* to give precise meaning to the intuitive sense of one model of computation, or algorithm, acting like another and rather than giving a general definition of simulation we will simply point out where and how the concept is being used. Sometimes the relationship between the computation on a simulator and simulated system is rather direct; for example below we show that families of IBCs simulate rule 110 with a small factor-2 blow-up in 'space' — in order to see that the two computations are essentially the same we simply ignore this small scaling difference. More often the simulator's computation is more convoluted than a simple blow-up in space, or time, for example the simulation of Turing machines by rule 110 (cited below). But in all cases the key point is that the computed function input-output mapping will be the same on both the simulated system and the simulator, except that we allow for some simple and well-defined input and output encoding differences between the two. The interested reader can find a variety of formalisations of simulation in the literature [20, 40, 36, 41].

As noted, we will give a method that converts any Turing machine into an IBC that simulates the Turing machine on inputs of a certain size. In doing so, we must be careful to note that no single IBC can fully carry out the algorithm of a given Turing machine, for the simple reason that the Turing machine is defined to compute on inputs of arbitrary size, may use arbitrary amounts of memory while doing so, and may take arbitrarily long to complete the computation, all without having to know these quantities in advance. In contrast, a given IBC takes inputs of a fixed size (n bits) only, uses limited memory (again n bits), and will reach a fixed point or limit cycle within a predictable time (no more than 2^n iterations). This distinction may be understood as Turing machines being a *uniform* model in that a single machine can be specified to solve problems of all sizes, while IBC are a *non-uniform* model in that new machines must be specified for solving larger problems. However, by providing a systematic construction for solving larger and larger

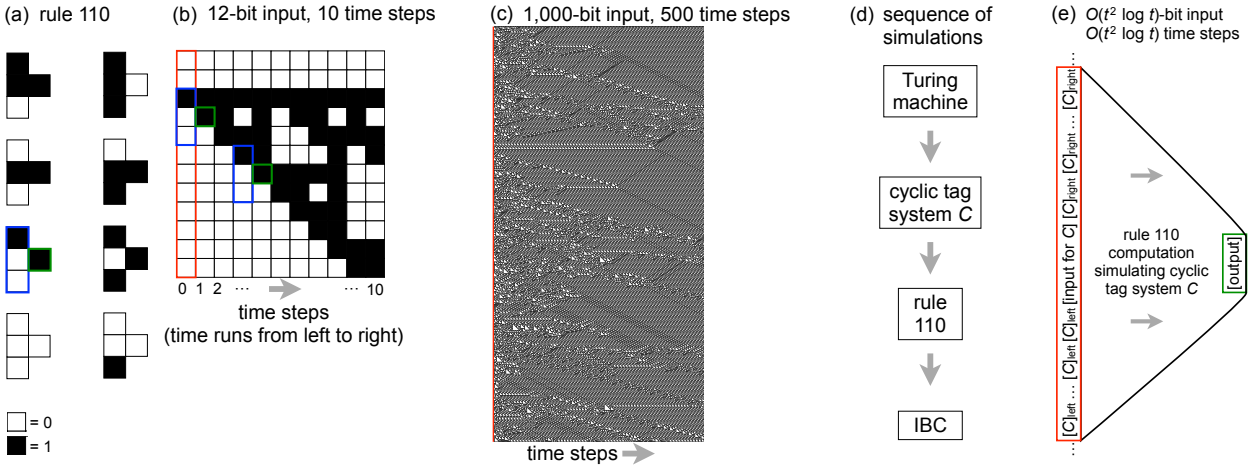


Figure S3: The cellular automaton rule 110. A configuration consists of an infinite line of cells, each in state 0 (white) or 1 (black). (a) Rule 110 ("program"): each cell updates according to is current state and that of its two neighbours. Each of the 8 possible 3-bit words has an associated update. An example is highlighted: there are three cells in blue, the centre cell has value 0 and bottom neighbour with value 0 and top neighbour with value 1, the centre cell updates itself to state 1 (green). (b) An example 12-bit initial configuration with 12 cells highlighted in red (immediately to the above and below we assume a boundary condition where cells not visible are always white). Ten parallel updates, or time steps, of the rule are shown, with time running from left to right. Updates happen synchronously (in parallel) for all cells at a given time step. (c) An example 1,000-bit initial configuration that has been run for 500 time steps, with time running from left to right. The resulting image is called a "space-time diagram" and is a record of all updates. The space-time diagram illustrates some structures and interactions that occur during the time-evolution of rule 110. (d) In order to simulate any Turing machine (i.e. algorithm) that runs in time t, the Turing machine and its input x is converted to a cyclic tag system C and input x', which are in turn suitably encoded as a rule 110 configuration [21, 42, 22]. If the Turing machine halts in t time steps, then a special word appears in the rule 110 space-time diagram after $O(t^2 \log t)$ time steps. The bottom arrow denotes the simulation result shown in this section; via this chain of simulations n-bit IBCs simulate any algorithm. (e) Illustration of a rule 110 instance simulating a cyclic tag system C, with the initial configuration of C encoded on the left, and an encoding of the output of C being produced on the right.

instances of a problem, one can meaningfully speak of IBC computation in a uniform sense by defining a notion called uniform families of IBCs – and this is what we do here. We fix a Turing machine M, and then for each input length $m \in \mathbb{N}$ to M we will show that there is an IBC of a special form that emulates M on input words of that length. Considering the full (infinite) set of input lengths, we obtain a set, or *family*, of IBCs, and because of how we designed the IBCs, their structure "looks similar": meaning there is a simple and efficient algorithm that takes the number m as input and writes out a description of the IBC that simulates M on inputs of length m. A family of IBCs with that property is called a *uniform family of IBCs*, and is more formally defined below.

S1.3.1 Uniform families of IBCs simulate Turing machines

In this section we show that the IBC model is maximally computationally powerful in the sense that IBCs are capable of executing any algorithm. Mathematically stated as Theorem S1.1, this result means that given any algorithm (or more specifically, any Turing machine) there is a family of IBCs that simulates that algorithm. For larger input sizes and larger running times of the simulated system, one uses n-bit IBCs with larger n. Although we need to scale n appropriately, the resulting simulator IBC is efficient, in a well-defined sense: n is (merely) polynomially larger than the number of steps (running time) taken by the system being simulated. Interestingly, we do not need to scale the number of distinct layers, as this full algorithmic complexity is achieved using merely single-layer IBCs ($\ell = 1$).

The proof of this "computational universality" result proceeds by taking the known result that the cellular automaton rule 110 simulates any Turing machine, and then giving an IBC simulation of rule 110.

Rule 110 definition. Rule 110 is a nearest neighbour, one dimensional, binary cellular automaton [43]. An *instance* of rule 110 consists of an infinite sequence of cells $P = \ldots p_{-1}p_0p_1 \ldots$ where each cell has a binary state $p_i \in \{0,1\}$. At discrete timesteps $t \in \{1,2,3,\ldots\}$, the state of all cells P are updated as follows. The value of cell p_i at timestep t, written $p_{i,t+1}$ is defined as $p_{i,t} = F(p_{i-1,t-1}, p_{i,t-1}, p_{i+1,t-1})$ where the function $F : \{0,1\}^3 \to \{0,1\}$ is defined as follows and is applied synchronously (simultaneously) to all cells P:

$$\begin{aligned} F(0,0,0) &= 0 & F(1,0,0) &= 0 \\ F(0,0,1) &= 1 & F(1,0,1) &= 1 \\ F(0,1,0) &= 1 & F(1,1,0) &= 1 \\ F(0,1,1) &= 1 & F(1,1,1) &= 0 \end{aligned}$$

It is known [21, 42, 22] that rule 110 is capable of simulating any algorithm, where by "algorithm" we mean a Turing machine that halts on all inputs. This simulation requires a complex but systematic encoding of both the Turing machine and its input as an initial configuration (sequence of cells) for the cellular automaton.

S1.3.1.1 Deciding a language by output: w-word decides

In Section S1.1.1 we defined what it means for an IBC to decide a language by entering into an accept or reject cycle. Here we give a definition, which is more appropriate for the proof of Theorem S1.1 than our previous definition of *cycle-decides*, where the IBC outputs a special word if and only if it accepts its input. For some $w \in \{0,1\}^*$, an IBC C *w-word-decides* the language $P \subseteq \{0,1\}^n$ if for any input $x_0 \in \{0,1\}^n$ the word w appears as a subword of x_t, for some $t \in \mathbb{N}$ where x_t is an element of the IBC's computation X if and only if $x_0 \in P$. In other words, the IBC accepts x_0 if there is a t such that w is a subword of the output x_t of the t^{th} iteration. In this definition, w may or may not appear on a repeating cycle, and the IBC may be deterministic or randomised.

In some of our results, we need to encode, or reformat, the input word before giving it to an IBC. One must be careful not to cheat: the function that does the encoding job should not itself compute the answer! Thankfully, there is a relatively straightforward way to formalise what we mean by "cheating"; the encoding function should be strictly too weak, or at least conjectured to be too weak, to solve the problem that is to be solved by the IBC [44, 45, 46]. P is the class of problems solved by Turing machines that run in polynomial time in their input length [20], and since one of our goals in this section is to show that n-bit IBCs solve any problem in P, where n is polynomial in unencoded input length, we will use encoding functions computable in logarithmic space on a Turing machine.[9] Hence we add the following technical clarification to our definition of w-word decides. Specifically, we say that the IBC C w-word decides L *via a logspace encoding* if C w-word-decides the language $L' = \{\ g(x) \mid x \in L$ and $g : \{0,1\}^* \to \{0,1\}^*$ and g is computed by a Turing machine that uses workspace $O(\log |x|)$, i.e. logarithmic in $|x|$ $\}$.

S1.3.1.2 Uniform families of IBCs

With either the "w-word decides" definition, or the 'cycle-decides' definition, for some fixed n an n-bit IBC decides a finite set (words of length n). Turing machines, on the other hand, typically decide infinite sets. Hence it will be useful to generalise language deciding by IBCs to infinite sets: We define a *family of IBCs* $\mathcal{C}$ to be an infinite set of IBCs, one IBC C_n for each input length n:

$$\mathcal{C} = \bigcup_{n \in 2\mathbb{N}^+} \{C_n \mid C_n \text{ is an } n\text{-bit IBC}\}$$

We say that *the family of IBCs $\mathcal{C}$ decides the language* $L \subseteq \{0,1\}^*$ if for each $n \in 2\mathbb{N}^+$ the circuit $C_n \in \mathcal{C}$ decides the language $L \cap \{0,1\}^n$ (i.e. words of length n). Here "decides" means either "cycle-decides" or "w-word-decides" for some w (where w is fixed for all m).

[9] The class logspace, or problems solved on Turing machines that use workspace logarithmic of input length, is a subset of P and conjectured to be a strict subset of P. If this conjecture holds, our encoding functions do not "cheat", in the sense that if the IBC model with logspace encodings can be shown to be capable of solving *any* problem in P, then at least sometimes the IBC is doing the hard part of the computation, and it's not just done by the encoding. We could have chosen an even stronger notion for encodings, using functions from a class that is provably strictly weaker than P, such as AC^0 [47, 46], but this would have complicated the presentation.

Let $\langle \mathcal{C} \rangle = \{\langle C_n \rangle \mid \langle C_n \rangle$ is the standard encoding of the IBC $C_n \in \mathcal{C}\}$, where the "standard encoding" of an IBC is an unambiguous description of the IBC written as a binary word (for example, by first ordering the gates by gate number, and then writing out each gate truth table in lexicographical order). We say that $\mathcal{C}$ is a *uniform IBC family* if there is an IBC-encoding function $f : \{1\}^* \to \langle \mathcal{C} \rangle$ where $f(1^n) = \langle C_n \rangle \in \langle \mathcal{C} \rangle$ and f is computable on a Turing machine M_f. This means that there is an algorithmic description that relates all members of a family.

This is a step in the right direction, but requiring the IBC-encoding function f merely to be computable is too weak a requirement, due to the concern outlined previously, of a too-powerful encoding function possibly being able to solve the problem on its own, thus failing to formalize the idea that the *IBC itself* solves the problem. We would like family members to be encoded by a model of computation that is weaker than IBCs themselves [44, 45, 46]. We say that $\mathcal{C}$ is a *logspace-uniform IBC family* if M_f uses workspace that is $O(\log n)$, i.e. logarithmic in n.

S1.3.1.3 n-bit 1-layer IBCs simulate rule 110

Other results in this section imply that multi-layer IBCs are computational powerful. In this subsection we demonstrate that even the 1-layer variant of the IBC model is equally as powerful as the multi-layer model, within a polynomial factor. Intuitively, Theorem S1.1 (below) states that there is a family of n-bit 1-layer IBCs that efficiently simulate any algorithm. The main ideas behind the proof are illustrated in Figures S3 and S4. The proof shows the rightmost of the following pair of simulations:

$$\text{Turing machine} \mapsto \text{rule 110} \mapsto \text{IBC}$$

where $\mapsto$ means "is simulated by". We are not only concerned with the fact that the simulation is possible, but also with its efficiency: How does the "size" of the IBCs in the IBC family scale with the running time of the simulated Turing machine? Here, size means number of (input) bits n and number of layers ℓ for an n-bit ℓ-layer IBC. It turns out that n scales polynomially, and $\ell = 1$ (constant).

Let $t : \mathbb{N} \to \mathbb{N}$ be a function and for $x \in \{0,1\}^*$ we let $|x| \in \mathbb{N}$ denote the length of the word x.

Theorem S1.1. *Let M be a single-tape Turing machine that halts on all inputs $x \in \{0,1\}^*$ in some finite number of time steps $t(|x|)$ in either an* accept *or* reject *state. Let $L \subseteq \{0,1\}^*$ be the language decided by M. Then there is a word $w \in \{0,1\}^*$, such that for each input length $m \in \mathbb{N}$ to M, there is an n-bit 1-layer IBC C_n, with $n = O(t(m)^2 \log t(m))$, that w-word-decides the finite language $L_m = \{x \mid x \in L$ and $|x| = m\}$.*

Moreover, the family of IBCs $\mathcal{C} = \bigcup_{n \in R}\{C_n\}$, for $R = \{r(m) \mid r : \mathbb{N} \to \mathbb{N}, r = O(t(m)^2 \log t(m)), m \in \mathbb{N}\}$, w-word-decides L via a logspace encoding and is logspace-uniform.

Proof. We will first show that for each $m \in \mathbb{N}$ there is an $n = O(t(m)^2 \log t(m))$ and an IBC C_n that w-word-decides the finite language L_m via a logspace encoding, by simulating the Turing machine M on words of length m.

Let M be a Turing machine of the form in the theorem statement. We modify M to get a Turing machine M' that is identical to M in all ways, except if M rejects then M' loops forever (the modification is straightforward; for any line of M's program that sends M to the reject state s_{reject}, change that line so that it instead sends M' to a new state s_{loop} and add instructions so that if M' is in state s_{loop} then it stays in s_{loop} no matter what symbol is read).

It is known that for any input $x \in \{0,1\}^*$ of length $|x| = m$, M' is simulated by rule 110 in $O(t(m)^2 \log t(m))$ steps in the following sense: there is an initial configuration c_0 of rule 110 that encodes M' and x via an encoding function computable in logarithmic workspace on a Turing machine (Lemma 3 of [22]). Starting on c_0, and after repeatedly iterating rule 110, we obtain a configuration c_i, for some $i \in \mathbb{N}$, that contains (as a subword) the special word $w = 01101001101000$ if and only if M halts; and if such a configuration c_i appears it does so after $O(t(m)^2 \log t(m))$ time steps (see references [21, 42] for the definition of w, and Theorem 4.3.2 of [48] for the time analysis[10]). Although rule 110 configurations are infinite, since we know that M' always halts, we need only simulate a finite portion of the rule 110 configuration space in order to simulate M'. Figure S3(e) illustrates why; suppose we have a rule 110 computation that simulates a

[10]Theorem 4.3.2 of [48] gives the required statement; it makes use of the construction in [21] and improves upon an earlier similar result in [22].

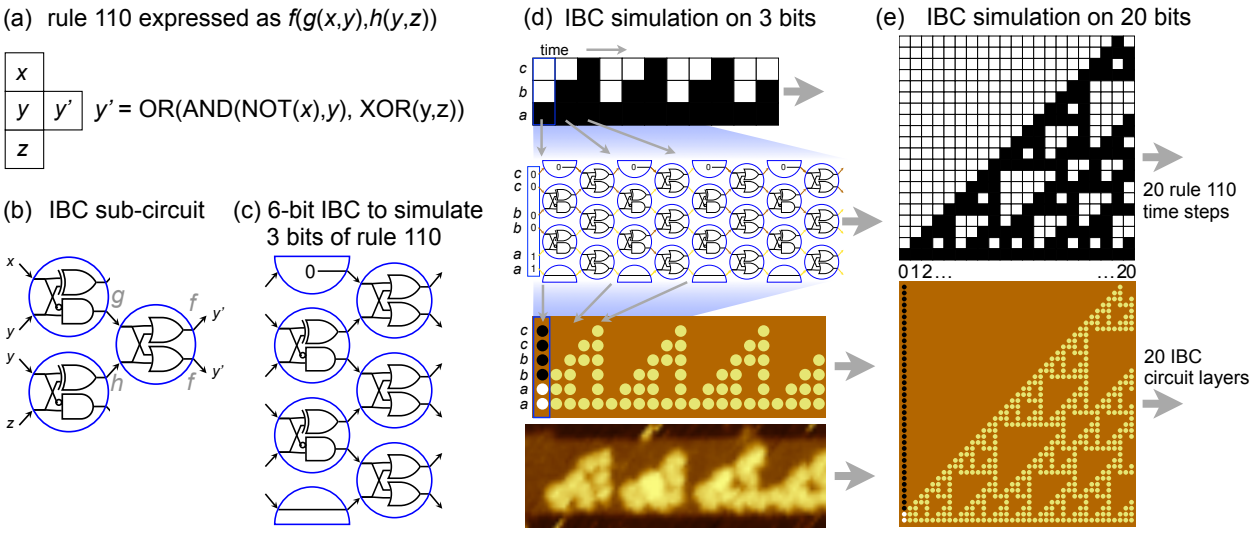


Figure S4: IBC simulation of rule 110. In this figure, time runs from left to right. (a) The rule 110 local update function $F(x, y, z)$ can be expressed as a function of the form $f(g(x, y), h(y, z))$ by setting $F(x, y, z) = \mathrm{OR}(\mathrm{AND}(\mathrm{NOT}(x), y), \mathrm{XOR}(y, z))$. (b) Design of an IBC "sub-circuit" that computes the update for a single cell: it takes as input x, y, y, z and gives y', y' as output. The principle underlying the sub-circuit is that it first computes g and h, and then takes their OR to give f. The sub-circuit repeats the value of y (and y') vertically; we say it has vertical scale-factor of 2. (c) A 6-bit IBC circuit layer that makes use of the sub-circuit in (b), along with some special top and bottom gates that compute a border condition. (d) 6-bit IBC simulation of 3-bit rule 110 instance. Top shows rule 110, then going downwards we have an IBC circuit that simulates rule 110, an example computer simulation, and finally the execution using self-assembly of DNA tiles (see Section S8.3 for more data). Here, we are using the following boundary conditions for both rule 110 and the IBC: The bottom bit is copied from one transition to the next, and if the top bit is ever 1 it flips to 0 on the text time step/iteration. (e) 20 layers of a 20-bit IBC carrying out a simulation of a rule 110 instance on 20 input bits.

Turing machine that halts. The region marked "output" in the figure contains w. Since the computation has completed in finite time $T = O(t^2 \log t)$ the region marked "output" is influenced by a region of length $\leq 2T$ in the initial configuration, hence we only need to simulate that region and we can ignore all other cells.

Figure S4 illustrates how we use IBCs to simulate rule 110 on finite-length configurations. We use two tricks. First, the local update function $f : \{0,1\}^3 \to \{0,1\}$ for rule 110 is expressed as a function of the form $F(x, y, z) = f(g(x, y), h(y, z))$ by setting $F(x, y, z) = \mathrm{OR}(\mathrm{AND}(\mathrm{NOT}(x), y), \mathrm{XOR}(y, z))$. Second, any function on 3 bits of the form $f(g(x, y), h(y, z))$ is implemented by a 3-gate sub-circuit if we allow the y-variable to be repeated (appear on two wires) as shown in Figure S4(b). When composing such sub-circuits together to form an n-bit 1-layer IBC (for even $n \geq 6$), in the way shown in Figure S4(c), we get an IBC layer that repeats each variable twice (vertically), along with some special "boundary conditions" on the first and last variable (top and bottom gates). We call this a scale-factor 2 simulation of rule 110 by an IBC. Each iteration of rule 110 corresponds to a single layer of an IBC computation.

Thus for M' acting on an input of length m, we have rule 110 simulating M', and an IBC, that we call C_n, for $n = O(t(m)^2 \log t(m))$, simulating rule 110. Moreover if the word $w = 01101001101000$ appears as a subword of some rule 110 configuration, then the word $w' = 00111100110000111110011000000$ appears[11] as a subword of one of the words that appear in the computation (sequence) of the IBC C_n; otherwise C_n iterates over a sequence of n-bit words that are repeated. We claim that w' does not ever appear as a subword. Note that w' is the "bit-doubling" of some word w in rule 110; clearly if w does not appear in rule 110, then w' will not appear at even positions, which "respect the bit-doubling boundaries" (i.e., such that w' starts and ends at boundaries of the doubled bits). The only way for w' to appear otherwise would be to start and end at an odd position, but since w contains both 0's and 1's, the bit-doubled encoding of w must have 01 or 10 as a subword. If w' did appear at an odd position, this would contradict the encoding function. Since all of the encodings we use are computable in logspace, and since the composition of logspace

[11] I.e. due to the IBC simulation of rule 110 being at scale factor 2, we set w' so that it duplicates each symbol of w.

functions are in logspace [49], we get that if M accepts (halts) then the IBC C_n w'-word-decides L via a logspace encoding.

This shows that for each $m \in \mathbb{N}$ there is an $n = O(t(m)^2 \log t(m))$ and an n-bit 1-layer IBC C_n that w-word-decides the finite language L_m via a logspace encoding.

We also immediately get that the entire family of IBCs $\mathcal{C} = \bigcup_{n \in R}\{C_n\}$, for $R = \{r(m) \mid r : \mathbb{N} \to \mathbb{N}, r = O(t(m)^2 \log t(m)), m \in \mathbb{N}\}$, w-word-decides $L = \bigcup_{m \in \mathbb{N}} L_m$ via a logspace encoding.

Finally, to see that $\mathcal{C}$ is logspace-uniform, we start with the intuition that as n increases the resulting IBCs scale in a very simple way: the IBC consists of $(n-4)/2$ subcircuits that compute the local update function f described above, and 2 small 'border' subcircuits. The intuition is formalised by defining a Turing machine Q that maps the unary number 1^n (i.e., n 1-symbols) to an encoding of the nth IBC C_n. Q begins by converting it's input 1^n into the standard binary representationfor the number n, written on it's worktape. Q then computes $i = (n-4)/2$ as the initial value for a counter we call i. On it's output tape, Q writes a description of the top "border" subcircuit, then writes, in succession, $(n-4)/2$ copies of the description of the subcircuit that computes the local update function f, using the counter i on the worktape to count from $i = (n-4)/2$ to $i = 1$. Q then outputs the bottom subcircuit. Each part of Q's computation can be executed using $O(\log n)$ cells of its worktape using standard techniques [49, 20], which gives the claim that $\mathcal{C}$ is logspace uniform. □

It is worth pointing out that our technique of simulating rule 110 with IBCs was to express the rule 110 update function $F(x, y, z)$ as a function of the form $f(g(x, y), h(y, z))$. It turns out that 192 of the 256 elementary CA can be simulated using this method. Since single-ayer IBCs are capable of simulating rule 110, and since rule 110 is Turing universal, the other 58 can be simulated, albeit indirectly, via rule 110 simulation. Nevertheless, we leave as an open question whether or not the other 58 elementary CA can be simulated by 1-layer IBCs in a more direct fashion than via the chain of simulations implied by Theorem S1.1.[12]

S1.3.2 Measuring computation: How hard is it to predict what an IBC will do?

Generally in science, we are interested in predicting the behaviour of a given system, be it a macro-scale system like the weather or a nanoscale molecular system like a set of DNA molecules. On the one hand, if a physical system, or a model of computation, is not very computationally expressive, then its state at a future time point is easy to predict (examples are systems whose state at an arbitrary time in the future is described by a closed-form formula). On the other hand, if a system is capable of arbitrary and efficient computation, thus having potentially complicated long-term dynamics, then, assuming several widely-believed mathematical conjectures, we have no way to predict its future behaviour other than explicit (and time-consuming) simulation.

One can ask such a prediction question of the IBC model. That is, given an IBC circuit and an input, what is the output or result of the computation? More generally, how difficult is it to answer this question for arbitrary IBCs? We can solve this "prediction problem" by simulating the IBC on its input, using a computer, but can we do better? For example, is there a closed-form formula to predict IBCs? Could we write a fast parallel program to simulate IBCs super-quickly, e.g. in constant, or logarithmic, or polylogarithmic, time? Theorem S1.2 below shows that we can essentially do no better than explicit step-by-step sequential simulation, assuming a number of widely accepted conjectures in computational complexity theory. This is a formalisation of the notion that IBCs, and by extension DNA molecules that simulate IBCs, are giving us maximum (up to polynomial factors) computational bang for our buck [46]. We begin with a concrete definition of the intuitive problem of prediction of the future state of an IBC.

[12] Specifically, the following 192 of the 256 elementary CA can be simulated by decomposing their local rule $F(x, y, z)$ into a function of the form $f(g(x, y), h(y, z))$: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 24, 25, 28, 29, 30, 31, 32, 33, 34, 35, 36, 38, 40, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 55, 56, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 75, 76, 79, 80, 81, 84, 85, 86, 87, 89, 90, 93, 95, 96, 98, 100, 101, 102, 103, 105, 106, 110, 111, 112, 115, 116, 117, 118, 119, 120, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 135, 136, 137, 138, 139, 140, 143, 144, 145, 149, 150, 152, 153, 154, 155, 157, 159, 160, 162, 165, 166, 168, 169, 170, 171, 174, 175, 176, 179, 180, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 199, 200, 203, 204, 205, 206, 207, 208, 209, 210, 211, 213, 215, 217, 219, 220, 221, 222, 223, 224, 225, 226, 227, 230, 231, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, where we are using Wolfram's [43] numbering scheme.

Definition S1.1 (IBC prediction problem)**.** Let $n \in \mathbb{N}$. Given a deterministic n-bit 1-layer IBC C, an input word $x \in \{0,1\}^n$, a time bound $T \in \mathbb{N}$ written in unary (the word with T 1's, i.e. 1^T),[13] and a length $\leq n$ word z; does z appear as a subword of one of the (output) words $x_0, x_1, \ldots x_{T-1}$ in the first T iterations of the computation of C on input x?

Thus, the input to the IBC prediction problem is a word of length $|C| + n + T + |z| + 3$, where $|C|$ is the length of a standard binary description of the circuit C, for example, a list of the truth tables for all its gates in a standard order, $|z|$ is the length of z, and the "+ 3" is for three separator symbols.

Our interest here is not in answering the IBC prediction problem for specific IBCs and inputs, but rather in more generally characterising how difficult this problem is. In computational complexity theory, the *difficulty* of a problem is measured using the amount of time steps and/or memory required on a Turing machine to solve the problem. P is the class of decision problems solved by Turing machines that run in a number of time steps that is polynomial of input length. A problem is P-hard, if it is at least as hard as any problem in P (solving the problem asymptotically requires at least as much resources as solving any other problem in P), and a problem is said to be P-complete if it is both in P and P-hard, meaning it is amongst the hardest problems in P. P-complete problems are an interesting class, especially in the context of physical systems. If a physical system can be "easily predicted" (such as the time for radioactivity of a sample to go below a certain level given an accurate estimate of its half-life, given by the closed-form solution $\alpha \cdot e^{-\beta t}$ for some $\alpha, \beta > 0$). then it is certainly not P-complete, and is in fact contained in one of the weaker complexity classes that are contained within P.[14]

The proof of Theorem S1.2 is almost a straightforward consequence of Theorem S1.1.

Theorem S1.2. *The n-bit 1-layer IBC prediction problem is* P*-complete.*

Proof. First, wish to show that the IBC prediction problem is in the problem class P by exhibiting a polynomial time Turing machine M that solves any instance of the problem. As per Definition S1.1, we are given an IBC C, input x, word z and a time bound T in unary.

For each iteration $t \in \{0, 1, \ldots, T-1\}$ of the IBC C starting at $t = 0$, the Turing machine M stores an encoding of the n-bit word x_t. To simulate the next iteration, M then computes x_{t+1} using C as a guide and in time linear in n, as follows. On its work tape, M stores an encoding of C: There are n segments of the work tape, immediately after iteration t the i^{th} segment contains (a) two copies of the value of the i^{th} bit of x_t, and (b) a copy of the three gates that are used to compute the i^{th} bit of x_{t+1}. An encoding of the three gates can be written down using $k \leq 24$ bits, (i.e. the sum of the lengths of the three truth tables). To compute the i^{th} bit of x_{t+1}, the Turing machine M reads all 3 or 4 bits of x_t (from the 'first copy') that influence the value of x_i (in $O(k)$ Turing machine steps), and then in $O(k^2)$ steps computes the required bit, which it writes in the 'second copy' position. Iterating over all n variables (values of i) to get the word x_{t+1} takes $O(k^2 n) = O(n)$ steps since k is a constant that is independent of n. After completing this step, for each i, the first copy is overwritten with the value of the second copy.

The next step is to check if z is a subword of X_t. There is a simple algorithm[15] to do this in $O(|z|n) = O(n^2)$ time on M (by having z by on the input tape, and explicitly checking for each $i \in \{1, 2, \ldots, n - |z|\}$ if $z = x_i x_{i+1} \cdots$).

If, for any t, the word x_t contains z as a subword, then M halts and outputs "yes", if not M applies another iteration. This process continues until either a yes answer is given, or until iteration $t = T - 1$, at which point M answers "no". On the Turing machine M, this simulation runs in time $O(Tn^2|C|) = O(Tn^3)$, which is polynomial in the input length $|C| + n + T + 3$, where $|C|$ is the length of the encoding of C in binary. Hence the IBC prediction problem is in the problem class P.

[13] The reason we write T in unary that we must choose some way to represent T and this particular choice (along with our other choices) leads to the IBC prediction problem being in the complexity class P. Providing T in binary would make the revised IBC prediction problem harder (e.g. by making it be hard for a complexity class conjectured to contain problems not in P), but would simply be another arbitrary choice about what prediction problem to study and would not change anything intrinsic to IBCs. A rough analogy is that choosing the encoding for T is akin to setting the scaling of an axis on a computational-hardness plot.

[14] For the sake of a concrete and straightforward discussion we are considering only deterministic systems that obey classical physics. This discussion can be reformulated for the stochastic systems and quantum systems.

[15] There are faster algorithms for this 'subword containment' task, but they are more complicated to describe and not required here.

Second, we wish to show that the IBC prediction problem is P-hard. We will show this by giving a reduction[16] from any problem in P to an IBC prediction problem. Let $L \in \mathsf{P}$ be a problem in P, then there is a Turing machine M that decides L and runs in time that is polynomial in the length of its input. Polynomial time Turing machines may use an arbitrary (finite sized) tape alphabet, if M' does not use a binary alphabet we modify it to get a Turing machine M' that is identical to M but where M' uses the binary input alphabet $\{0,1\}$. This modification can be easily done so that M' runs in a number of steps $t(|x|)$ that is polynomial in the input length $|x| \in \mathbb{N}$. Setting M in the hypothesis of Theorem S1.1 equal to M', we obtain for each input length $|x| = m$ an n-bit 1-layer IBC C_n, for $n = O(t(m)^2 \log t(m))$ that w-word decides L (using the word w in the proof of Theorem S1.1) via a logspace encoding. Thus in Definition S1.1, for any $x' \in \{0,1\}^n$ we obtain an instance of the IBC prediction problem by letting $x \in \{0,1\}^{O(t(|x'|)^2 \log t(|x'|))}$ be the word obtained from the encoding described in the proof of Theorem S1.1 (i.e. the input encoding to the IBC that simulates rule 110), $C = C_n$ where $n = O(t(|x|)^2 \log t(|x|))$, $T = O(n)$, and $z = w' = 00111100110000111110011000000$. We have given a general method that reduces (or "embeds") any problem in P into an IBC prediction problem, thus the IBC prediction problem is P-hard.

We have shown the n-bit 1-layer IBC prediction problem is in P and is P-hard, hence it is P-complete. □

S1.3.3 IBCs simulate Boolean circuits

Theorem S1.1 shows that IBCs are capable of implementing arbitrary algorithms. Indeed the power of such a universality result is that we can immediately combine it with another powerful result, namely that Turing machines simulate a wide variety of models of computation, to show that IBCs simulate other models of computation. In particular, since the IBC model is a restricted kind of Boolean circuit model, it is perhaps of particular interest to compare the two models. The size of a Boolean circuit is measured as the number of wires in the circuit.[17] It is known that Boolean circuits of size s are simulated by single-tape Turing machines that run in time $O(s^2)$: the idea behind the simulation is as follows. A circuit c can be encoded as a binary word $\langle c \rangle$: a sequence of subwords where each subword contains a unique gate identifier g, the type of the gate (e.g. AND, OR, NOT), and the identifier of the 1 or 2 gates that serve as input to gate g, and an empty spacer to encode the eventual value assigned to g. A Turing machine that is given $x, \langle c \rangle$ as input where x is an input word for c, can search through $\langle c \rangle$ continually updating the values for the gates, starting with the input gates and ending with the output gate(s) of c. For a single-tape Turing machine this can be done in time quadratic in the length of $x, \langle c \rangle$, and this in time $O(s^2)$ where s is the size of (number of wires in) c.

Putting this together with Theorem S1.1 gives:

Corollary S1.3. *Let c be a Boolean circuit of size s. Then c is simulated by a $O(s^4 \log^2 s)$-bit 1-layer IBC.*

By allowing multiple layers, i.e. ℓ-layer IBCs for $\ell > 1$, we can do better than Corollary S1.3 in the sense of using asymptotically fewer than $s^4 \log^2 s$ bits. We begin by showing that an arbitrary feedforward Boolean circuit c can be simulated within a single iteration block of an IBC. (For an ℓ-layer IBC C the term *iteration block*, or simply *block*, is defined to be C, i.e. the ℓ-layer circuit.) The primary issue is that standard circuit models allow unrestricted (feedforward) connectivity, while the IBC is constrained by exclusively short-range local geometry. Our method is similar to that of Goldschlager [50], who showed the P-completeness of the problem of predicting the value of a *planar* Boolean circuit, by showing a logspace transformation of an arbitrary Boolean circuit into a planar circuit. Our construction explicitly accounts for the width and depth of the given Boolean circuit. We also are not restricted to AND, OR, and NOT gates, so have no need of a "crossing gadget" as in [50]; an IBC gate can trivially cross two bits.

Our first construction shows that routing wires with fanout (i.e., copying some input bits to the output, possibly in a different order, and possibly removing some inputs and duplicating others) comes at a cost that is at most linear.

[16] A reduction is an algorithm for converting one problem L_1 into another problem L_2. The existence of such a reduction shows that any algorithm for L_2 can be used to solve the problem L_1. Reductions are a method of showing that one problem (L_2 in this case) is at least as hard as another (L_1).

[17] Each connection between two gates is a wire. The *number of gates* is also sometimes used as a size measure, but we prefer number of wires since (a) the latter is an upper-bound on the former and (b) what we really care about here is the size of the circuit description when encoded as an input word to some other computational device like a Turing machine, and most of that description is concerned with describing the connectivity of the circuit.

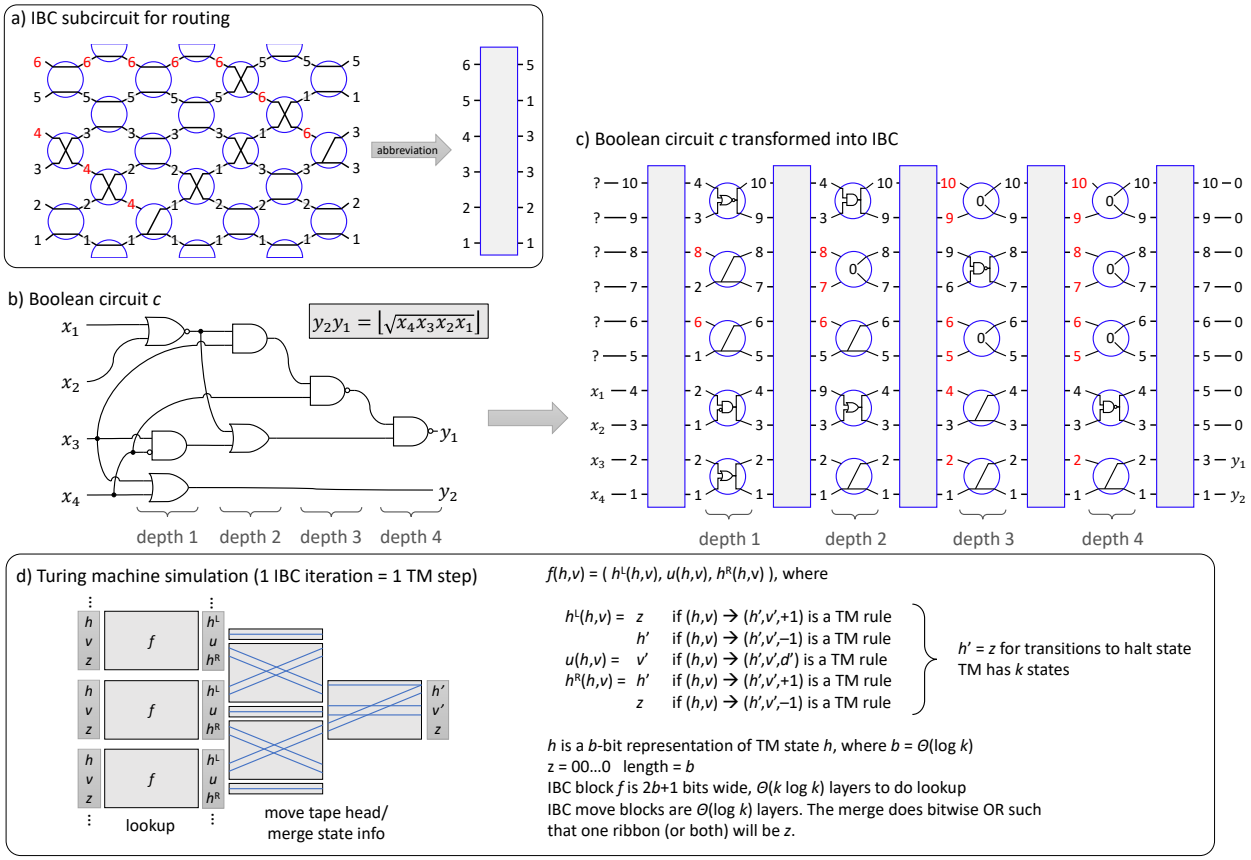


Figure S5: IBC simulation of Boolean circuits. (a) Routing signals in $O(w)$ layers. Gates are determined by sorting and merging from the destination identities back to the origin identities, using a variant of the Odd-Even Transposition Sort. Red target identities indicate those that appear in the origin but not the destination. (b) An example feedforward circuit with 2-input Boolean gates and unbounded fanout, with gates arranged according to depth. This circuit is depth 4 and width 5, counting wires in the width as necessary to transmit information (i.e. effectively an identity gate in each layer). (c) Construction of an IBC block that implements the same circuit by alternating routing blocks and layers for gates at a given depth. For a depth d circuit with width w, this construction results in a $2w$-bit, $O(wd)$-layer IBC block, assuming the number of inputs is less than $2w$. Red target identities indicate those whose signals are ignored by the subsequent gates. Thin vertical rectangles indicate routing blocks. Note that as a diagrammatic shorthand, because we use IBC gates whose truth tables always produce the same bit for both outputs, we show just a single logic gate inside the circles. (d) Construction of an IBC iteration block that simulates a single step of a binary-tape Turing machine with k head states. A computation bounded by space s is simulated by an IBC iteration block with $O(s \log k)$ bits and $O(k \log k)$ layers.

Lemma S1.4. *For $i_1, i_2, \ldots, i_n \in \{1, 2, \ldots, n\}$, let $f(x_1, x_2, \ldots, x_n) = (x_{i_1}, x_{i_2}, \ldots, x_{i_n})$ be a function describing a chosen routing of inputs to outputs, possibly including fanout. Then there is an n-bit ℓ-layer IBC with $\ell < n$ whose single iteration function is f.*

Proof. Figure S5(a) illustrates the construction. The target identities $i_1, i_2, \ldots, i_n$ are laid out vertically at the right side of the IBC block, and gates are determined backward toward the input, with the gate type at each location determined based on sorting or merging the target identities. When target identities are distinct and already in order, the gate computes an identity function; when target identities are out of order, the gate performs a swap; and when target identities are the same, the gate ignores one input and fans out the other input. When merging, an unused target identity is chosen for the unused input. Eventually, this process will result in all target identities being distinct and in order. The resulting fixed IBC circuit performs the desired routing with the necessary fanout. What remains to be shown is that the IBC circuit has n layers or fewer. If there is no fanout – i.e. the target identities are all distinct – then the construction

process is exactly an Odd-Even Transposition Sort [51],[18] which requires no more than $n/2$ layers. In the case that there are duplicated target identities, we claim that all merges will be performed within the first $n/2$ layers of the construction (i.e., the rightmost layers), after which another $n/2$ layers is sufficient in the worst case to ensure that the unused targets are routed to their origin. □

It is now straightforward to implement an arbitrary feedforward Boolean circuit within a single IBC iteration block. The intuition is that a Boolean circuit may be laid out as a series of layers, with the wiring between layers provided by the above routing construction. To be more precise, we define the *depth* of a gate in a Boolean circuit to be 1 more than the depth of its deepest input, where the Boolean circuits' inputs are depth 0. The depth of the Boolean circuit is the maximal depth of a gate. A *layered* Boolean circuit layout places each gate in a layer corresponding to its depth. The *width* of a layered Boolean circuit layout is the maximum, over layers, of the number of gates in the layer plus the number of wires going through the layer. (A simplifying convention is to have no wires going through layers, but add IDENTITY gates in each layer to replace those wires.) An example Boolean circuit is shown in Figure S5(b). The layered convention simplifies the construction of an equivalent IBC.

Theorem S1.5. *Let c be a feedforward Boolean circuit of depth d and width w, where the number of inputs is $m \leq 2w$. Gates implement arbitrary 2-input Boolean functions, and fanout is unbounded. Then the function computed by c is implemented as a single iteration of a $O(w)$-bit $O(wd)$-layer IBC C. Specifically, C has width $2w$, the inputs may be provided on any choice of m input wires, and the outputs produced on any choice of output wires.*

Proof. Our construction (which might be possible to optimise further) uses a single IBC gate for each gate or wire in a layer of c, connecting each layer with the routing construction of Theorem S1.4, for a total of at most $2w(d+1)+d$ layers. Note that computing gates produce the same output on both output wires, as do the gates that serve as pass-through wires; this extra fanout may simplify routing between layers, but is not essential. Figure S5(c) illustrates this method. □

Note that the above construction, for a Boolean circuit c that computes function $f : \{0,1\}^n \to \{0,1\}^m$, in general will not yield an IBC C that computes f in the sense defined in Section S1.1.1, because that notion entails iterating the function computed by a single iteration until a fixed point is achieved. (Also, the IBC's single iteration function is some $g : \{0,1\}^{2w} \to \{0,1\}^{2w}$.) However, it is straightforward to modify c to produce c' of similar width and depth (larger by no more than linear in m) that maps input words $00^m x$ to outputs $1y0^n$ where $y = f(x)$ while mapping inputs $1y0^n$ to $1y0^n$. This results in an IBC that hits a fixed point after one iteration.

Performing the entire circuit computation within a single iteration of an IBC fails to utilize the power of iteration, resulting in unnecessarily large implementations for many types of computation. Much more compact implementations could be achieved by designing circuits such that, when iterated, they achieve a fixed point when the desired computation is complete. As an example, of this effective use of iteration, consider a circuit that implements a single step of a Turing machine on a finite tape.

Theorem S1.6. *Given a Turing machine M with a binary tape and k head states that uses space bounded by s for inputs of size bounded by m, there is an $O(s \log k)$-bit $O(k \log k)$-layer IBC C that simulates M in the following sense: using a linear-time encoding e, for all inputs x to M that result in $M(x)$, providing input $e(x, H_1)$ to C will reach a fixed point $e(M(x), H_0)$ where H_1 is the initial head state for M and H_0 is the halt state.*

Proof. The basic idea for this construction is that the bits of the IBC will directly represent s bits of the Turing machine's tape as well as $b = O(\log k)$ bits representing the Turing machine's head state[19]. In order to ensure that updates are local and efficient, the n IBC bits will divided into s cells each containing 1 bit for the corresponding Turing machine's tape cell and $2b$ bits that (a) store the Turing machine's head state

[18] Note that this refers to sorting the *integers* $\{1, 2, \ldots, n\}$, not sorting bits as in the SORTING IBC, and that the sorting is used to construct the circuit, but the circuit itself just implements fixed wiring with no sorting taking place when it executes, since the swaps and copies are not dependent on the bits.

[19] Our construction is very much in the spirit of classic results relating Turing machines to circuit complexity [44]; however we don't use that construction directly because here we must take into account the localized geometry of circuit layout required in for the IBC model.

when the Turing machine is reading that cell, (b) are otherwise zero, and (c) provide scratch space for the update computation. Specifically, at iteration t, cell i contains word $h_i^t v_i^t z_i^t$ where h_i^t is the b-bit binary encoding of the head state if the Turing machine is reading cell i at step t, z_i^t is always b zeros, and v_i^t is the value of the Turing machine's tape cell i at step t; the halt state is represented by all zeros. This encoding is illustrated at the left of Figure S5(d), which also illustrates the main components of the IBC iteration block. The construction is straightforward: for each cell, the IBC starts with an identical sub-block that performs a hard-coded table lookup based on the Turing machine's rule table; the new state information is routed to the adjacent cells; and the new state information coming from both sides is merged into its standard location. The lookup table, with input bits hvz, provides new bits $h^L v' h^R$ where h^L is the new head state if the head should move left (else it is zero), h^R is the new head state if the head should move right (else it is zero), and both are zero if the Turing machine should halt. Thus, after all cells have performed their lookup, exactly one cell will have exactly one head state encoded (or none if the machine has halted). Thus all but (at most) one merge sub-blocks will simply copy the cell value bit and set the head state bits to zero. The routing and merging clearly takes $O(\log k)$ layers. To conclude the proof, we argue that the table lookup can be done in $O(k \log k)$ layers. Note that there are (at most) $2k$ entries in the Turing machine's rule table, since the tape alphabet is binary. To implement a single entry lookup to match a specific case of hv requires $O(\log k)$ layers to non-destructively perform the match and place the result in the scratch space, then $O(\log k)$ layers to OR the hard-coded result into $O(\log k)$ scratch bits storing the accumulated output. Since exactly one entry lookup will be positive, the OR of all $O(k)$ entry lookups will be the desired result. At the end of all the entry lookups, $O(\log k)$ layers are used to route the result in the scratch space to the intended output locations.

To conclude the proof, note that the encoding e from a Turing machine state to an input word for the IBC is linear time, and that iteration of the IBC iteration block will reach a fixed point encoding the halted state of the Turing machine, if the Turing machine halts. □

It is instructive to compare the resource usage aspects of Theorem S1.6 to those of Theorem S1.1. Consider a space-bounded Turing machine M that uses space $s(x)$, and time exponential in its space usage i.e. time $t = 2^{\Theta(s)}$. The construction of Theorem S1.6 yields an $O(s)$-bit $O(1)$-layer IBC that must be run for t iterations, while the construction of Theorem S1.1 yields an $O(t^2 \log t)$-bit 1-layer IBC that must be run for $\Theta(t^2 \log t)$ iterations. Thus, for a molecular implementation of M, the latter construction requires exponentially more tile strands to be designed for the iteration block, requires an exponentially larger initial seed, and consumes exponentially more tile strands during growth (primarily due to the width rather than the length). Hence, by allowing only a small number of extra layers in our IBC, by going from 1-layer to $O(k \log k)$ layers, we could see large reductions in the number of molecules required in molecular computations. This observation provides motivation for future work on implementing IBCs with more than one layers.

S1.3.4 A natural variant of IBCs simulate branching programs

We have seen that IBCs are about as powerful as they could be (a) when limited to 1 layer but unbounded width, and (b) when neither limited by layers or width, in which case the computation is more efficient (in terms of the product $w \times l$ needed to compute a given function, which is linearly proportional to the number of molecular species that must be designed) and does not need a complex input or output encoding. In contrast, the third natural limit – that of width bounded by some constant k – does not support IBC computation of arbitrary functions, for the simple reason that at most k bits of input can be provided. This limit is not specific to the IBC model; it would apply to any bounded-width circuit model where all inputs must be provided in the initial layer. Given our current molecular implementation as fixed-circumference DNA nanotubes, it might be seen as disappointing if the bounded-width constraint made the IBC model uninteresting computationally.

Fortunately, in circuit complexity theory, the topic of bounded-width circuits is well studied [37, 52] and provides profound insights into the nature of computation – however, to do so, it is necessary to consider a circuit model where inputs may be provided at any layer of the circuit, not just the first layer. In this section, we examine the consequences of similarly allowing inputs to be provided to specific gates at any layer within each iteration of an IBC. From the perspective of molecular implementation, this is a natural choice. To provide input on a seed assembly when implementing the standard IBC model, a specific strand (or set of strands) is added to the solution for a zero in a given bit position, and a different strand (or strands) is added

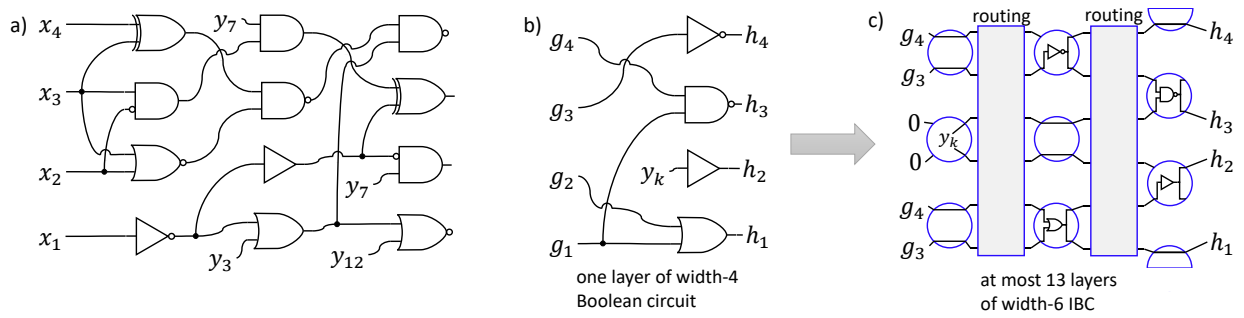


Figure S6: Simulation of width-4 Boolean circuits by width-6 global-input IBCs. (a) Three layers of an example width-4 Boolean circuit (in the figure "width" is vertical). (b) A single layer of a width-4 Boolean circuit in which at most one global input (here, y_h) is used. (c) Generic construction of a width-6 global-input IBC block that implements a generic single layer of a width-4 Boolean circuit of the type shown in (b). The "routing" modules are explained in Figure S5. Note that as a diagrammatic shorthand, because we use IBC gates whose truth tables always produce the same bit for both outputs, we show just a single logic gate inside the circles.

if that bit position should be a one; these strands are incorporated exactly once in the final assembly. If we allow a new class of gates that output a given input bit, this could be implemented similarly such that if a given input bit is zero, then a specific strand (or strands) is added to the solution, while a different strand (or strands) is added if that input bit is one; however this time, each such strand would be incorporated every time a gate "asks for" that input, which could be multiple times within each iteration block[20]. We now formalize this enhanced model and show that it has surprisingly strong computational power, even for width 6.

Definition of the global-input IBC model. The *global-input* IBC model is defined identically to the standard IBC model, but augmented with an additional type of gate and an additional type of input. For $n \in 2\mathbb{N}^+$ and $\ell, m \in \mathbb{N}$, in an n-bit ℓ-layer m-global-input IBC, the *seed input* $x \in \{0,1\}^n$ consists of n bits specifying the initial values of wires, and the *global-input* y consists of $m \in \mathbb{N}$ bits that can be accessed by *global-input* gates. A global-input gate g_r within a specific layer has an associated *gate position* $r \in \{1, 2, \ldots n+1\}$ and an associated *input choice* $y_i \in \{0,1\}$; each of the gate's one or two outputs has the value y_i (i.e. the global-input gate ignores its one or two input wires). We say that a m-global-input IBC C with seed input $x_0 \in \{0,1\}^n$ computes a function $F : \{0,1\}^m \to \{0,1\}^n$ if for every global input $y \in \{0,1\}^m$ and every resulting circuit computation $X = x_0, x_1, x_2, \ldots$ of C, there exists $t \in \mathbb{N}$ such that $F(y) = x_t = x_{t+1} = x_{t+2} = \ldots$, i.e. the circuit's fixed point is $F(y)$. For a Boolean function $F : \{0,1\}^m \to \{0,1\}$ we simply require that the output bit $y \in \{0,1\}$ appear at some fixed index $j \in \{1, 2, \ldots n\}$ in each of the words $x_t, x_{t+1}, x_{t+2}, \ldots$.

Width-6 global-input IBCs simulate width-4 Boolean circuits. To characterize the computational power of bounded-width global-input IBC, we make use of the remarkable result by Barrington [37] that polynomial-length width-5 branching programs compute exactly Nick's class NC^1, the class of n-bit Boolean functions computable by depth $O(\log n)$ Boolean circuits of 2-input gates[21]. In the same paper, Barrington

[20] Nonetheless, there is an important difference distinguishing molecular implementations of the standard IBC model and the global-input IBC model: In global-input implementations, every assembly in the same test tube must process the same input, but in the standard implementation, each seed could contain a different input. For example, the natural molecular implementation of global-input IBC cannot accept randomised global input unless each input is read and used exactly once; but for the class of fixed-width computation discussed here, it is essential to read each input multiple times – in which case the molecular implementation could not ensure that the same tile choice is used in each instance. As another caveat, note that we want the number of input molecules to scale with the number of input bits, not the number of times that the input is used in the circuit; in that case, the global-input gates must be implemented such that the molecules are not encoded to a specific layer (as a given global input bit may be read by multiple distinct gates in different positions and layers). We do not anticipate that this would be problematic experimentally so long such gates are sufficiently well separated spatially in the circuit.

[21] Nick has many classes: $\mathsf{NC}^1, \mathsf{NC}^2, \ldots, \mathsf{NC}^k$, the latter being functions computable by Boolean circuits of depth $O(\log^k n)$. He also has the union of them all, NC, i.e. functions computable by polylogarithmic-depth Boolean circuits, which by necessity are also of polynomial size.

showed that polynomial-length width-4 Boolean circuits (that allow inputs to be provided to any gate) also compute the exactly same class, by showing that they can simulate width-5 branching programs. Here we show that polynomial-layer width-6 global-input IBC also simulate such width-4 Boolean circuits, and thus width-5 branching programs. As a corollary[22] one obtains that uniform families 6-bit global-input IBCs decide any problem in NC^1.

Theorem S1.7. *Let $F : \{0,1\}^m \to \{0,1\}$ be the Boolean function on m bits computed by a width-4 Boolean circuit of depth d, then F is computed by a 6-bit $O(d)$-layer global-input IBC.*

Proof. The construction will be a direct layer-by-layer implementation, as illustrated in Figure S6. We first note that a width-4 Boolean circuit may read as few as 0 or as many as 4 distinct inputs at each layer, however, for our purposes it is convenient if each layer reads at most 1 input. Converting an arbitrary width-4 Boolean circuit to this form is straightforward by splitting each layer into up to 4 layers that each read at most 1 input, and using IDENTITY gates to pass values from layer to layer. Each layer in the resulting width-4 Boolean circuit can be converted to an equivalent block of layers for a global-input IBC as shown in Figure S6(c). An initial layer utilizes a global-input gate, if required, and then up to 6 layers are used to route and fanout wires to the first two compute gates, one of which may receive the global input. The remainder of the wires pass through the initial wire values in some order. A second block of up to 4 layers routes the initial wire values to the second two compute gates. At this point, the output of the original width-4 Boolean circuit layer is equal to that of the width-6 global-input IBC. Note that the IBC's seed input was not utilized, so every iteration yields the same result, which is therefore immediately the fixed point. □

As the construction for Theorem S6 did not make use of the IBC's capability for iteration, which was essential for Theorem S1.1 and Theorem S1.6, it would be interesting to ask whether iteration similarly provides extra computational power in the context of bounded width.

S1.4 Discussion and open questions on the theory of IBCs

We have shown some results on the capabilities of IBCs. Sections S8.1–S8.21 show that the 6-bit 1-layer IBC model has a wide variety of computation capabilities, some of which could be used as inspiration for future work. Furthermore, results earlier in this section show the general purpose capabilities for the n-bit single-layer and/or multi-layer model. Here, we suggest a number of directions for future work.

In terms of their computational complexity, IBCs are well-characterised by the class P: they have a P-complete prediction problem (Theorem S1.2). In other words, for any $\ell \in \mathbb{N}^+$, the class of n-bit ℓ-layer IBCs that run for a number of iterations that is polynomial in n decide exactly those problems in P. To probe deeper, one can take a more fine grained approach and ask questions such as: For any $n \in \{2, 4, 6, \ldots\}$ and any $\ell \in \mathbb{N}^+$, are there languages decided (or functions computed, in any reasonable sense) by n-bit $\ell + 1$-layer IBCs that are not computed by n-bit ℓ-layer IBCs? For fixed ℓ, is there a clean characterisation of the functions computed by n-bit ℓ-layer IBCs for each $n \in \{2, 4, 6, \ldots\}$? Likewise, for fixed n, is there a clean characterisation of the functions computed by n-bit ℓ-layer IBCs for each $\ell \in \mathbb{N}^+$? Since our IBCs use the binary alphabet $\{0, 1\}$ one can also ask if there is a benefit to having a larger alphabet. For instance, is there a trade-off between number of layers and number of symbols?

Consider the problem PALINDROMES, defined as the set of even length words from $\{0, 1\}^*$ that read the same forwards as backwards. If we disallow any input encodings, does there exist a fixed number of layers ℓ for which the PALINDROMES problem can be solved by ℓ-layer IBCs? Indeed, is it solvable for $\ell = 1$?

[22] We omit a proof of this corollary to save on unnecessary formalism, and instead sketch an overview here. To obtain a proof, one first defines the notion of a family of 6-bit $\ell(n)$-layer n-global-input IBCs, where $\ell : \mathbb{N} \to \mathbb{N}$ and with family members parameterised by n, i.e.: $\mathcal{C} = \{C_n \mid n \in \mathbb{N}$ and C_n is a 6-bit $\ell(n)$-layer n-global-input IBC$\}$. Then one observes that for any $L \in \mathsf{NC}^1$ (via Barrington [37]) there is a family of width-4 Boolean circuits $\mathcal{B}$ that decide L, also parameterised by n such that $B_n \in \mathcal{B}$ decides $L \cap \{0,1\}^n$. By applying Theorem S1.7 to each member of $\mathcal{B}$ one obtains a family of 6-bit global-input IBCs that also decide L. A related issue is one of uniformity. Like many problem classes defined via Boolean circuits, NC^1 comes in both uniform and non-uniform flavours. "Uniform" means that there is some suitably simple algorithmic method that given the number $n \in \mathbb{N}$ in binary outputs a description of the n^{th} circuit [47, 20]. Non-uniform classes may have no such associated method. For larger and larger (global) input sizes our IBC construction is simple enough to be uniform in this sense. Likewise, here, for the case for any language L in uniform-NC^1 [47] we get that there is a uniform family of 6-bit global-input IBCs that decide L, although we are omitting a formal proof.

Since input encodings are disallowed, one can not immediately make use of Corollary S1.3 nor Theorem S1.6 (i.e., simulation of rule 110 or Turing machines), and since ℓ is fixed one can not immediately make use of Theorem S1.5 (Boolean circuit simulation). Note that our 6-bit 1-layer IBC PALINDROME (Section S8.7) for deciding the 6-bit PALINDROME problem was found by computer search, and not by a principled design easily extended to larger input sizes. The obvious methods to decide the PALINDROMES problem on Turing machines or Boolean circuits do not easily carry over to IBCs. PALINDROMES seems to require the ability to simultaneously communicate over long distances, cross signals over each other in 2D and do many bit comparisons, which seems challenging with IBCs since the alphabet is minimal (binary), the connectivity is local, and there are a fixed number of layers (e.g., $O(1)$, or even just 1).

To think about such issues in a more general setting, it might be worth looking to communication complexity theory [53, 54]. In particular, there has been some fascinating work on analysing the power, and simulation abilities, of cellular automata using communication complexity [55, 56]. In this setting, Alice and Bob wish to cooperatively decide some problem L, of even length words. Roughly speaking, for any n-bit input word, Alice is given the first $n/2$ bits of the input, Bob the latter $n/2$, and the communication complexity of the problem is defined to be minimal number of bits they need to communicate to cooperatively to solve it. PARITY, here defined to be the set of all even-length binary words with an odd number of 1s, has communication complexity of 1 since Alice and and Bob can compute the parity (a bit) of their respective halves of the input, then Alice sends Bob her parity bit which Bob XORs with his to give the answer. PARITY is also easy for IBCs to decide: the 6-bit 1-layer IBC PARITY (Section S8.2) has an obvious extension to n-bit 1-layer IBCs for arbitrary even n. On the other hand, the communication complexity of PALINDROMES is maximal, in the sense that Alice and Bob need to communicate $n/2$ bits to solve it [54]. Although this discussion is suggestive of using communication complexity to further characterise the power of having few or multiple layers (and perhaps input-encodings) in IBCs, the picture is not so straightforward since there are computational problems with low communication complexity, but high computational complexity (e.g. as measured by required time on a Turing machine), which would presumably require multi-layer, or input-encoded, IBCs, despite the ease of their associated communication problem. Nevertheless, throughout this section we have already explored a number of links between time/space/circuit resource-bounded complexity theory and IBCs, and we contend that exploring the connections between IBCs and *self-assembly in general* on the one hand, and communication complexity on the other hand, may prove fruitful avenue as a future research direction to better understand the limitations and power of both IBCs and algorithmic self-assembly.

In our work we have given a number of randomised IBCs, but in this section we omitted any analysis of the computational power of randomised IBCs. We leave this as an open direction for future work.

We have shown that a natural prediction problem for IBCs is P-complete. Condon [57] has defined a notion of P-completeness that is stricter than the one we used and is fine-grained enough to potentially forbid even polynomial speedups on processors with a polynomial number of parallel processors. Is the IBC prediction problem strictly P-complete in the sense of Condon?

Although we define the IBC model using terminology from Boolean circuits, the model could also be described as a Boolean Automata Network model [39].[23] In Boolean Automata Networks one defines a network of nodes with arbitrary connectivity with each node having a state from $\{0, 1\}$ and where sates are updated based on a (possibly probabilistic) function of the states of neighbours and according to some synchronous or asynchronous update schedule. The n-bit ℓ-layer IBC model can be defined as a Probabilistic Boolean Automata Network with restricted gate connectivity (we should connect the outputs of layer ℓ to the inputs of layer 1 and otherwise have our usual IBC wiring within layer(s)) and a synchronous update schedule (starting at the inputs update at each "half-layer" until the last, then iterate). It remains as future work to explore connections with Boolean Automata Network theory and IBCs.

[23] We thank Guillaume Theyssier for pointing this out to us.

S2 System design abstraction level: abstract tile assembly model and proofreading

S2.1 Definition of Cylindrical Tile Assembly Model

Building on the mathematical theory of tiling [58], observations of algorithmic patterning within crystals [59], and statistical mechanical models of crystal growth [60], the abstract Tile Assembly Model (aTAM) [12] is a model of self-assembly where square tiles attach to a growing structure according to simple local rules encoded within the tiles. See [12, 3, 61] for the original formal definition of the aTAM. This model is capable of universal computation [62, 2, 63] as well as universal construction [18, 19, 64], and even has the surprising capability of being able to simulate any instance of itself with only a constant-factor simulation overhead in both space and time, i.e., the model is intrinsically universal [35]. Here, we define a variant of the model more appropriate to the experiments of this paper, distinguished by two differences with the standard model:

1. Tiles grow on a topological cylinder, rather than the Euclidean plane.
2. All glues are of the same strength.[24]

The second constraint is the motivation for the first, and is itself motivated by the ease of designing many DNA sticky ends of length 10 or 11 with equal binding strength, compared to the difficulty of finding many such sticky ends close to one binding strength s and many close to $2s$. In the discrete plane $\mathbb{Z}^2$, every point outside of a rectangle is adjacent to at most one point within the rectangle. Thus cooperative binding between two single-strength glues on different tiles cannot initiate growth outside the rectangle. Without double-strength glues to initiate such growth, any assembly can grow at most to fill its bounding rectangle. However, in a cylindrical topology, it is possible for growth to continue unboundedly using only cooperative binding (see Figure 1 in main text).

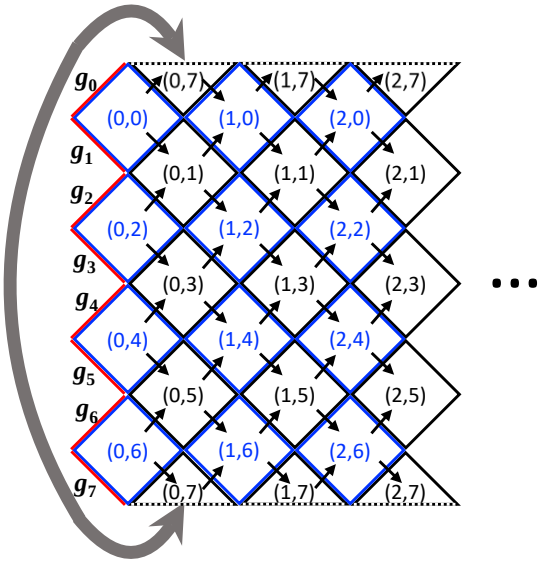


Figure S7: Points in the cylindrical lattice $\mathcal{C}_8 = \mathbb{N} \times [8]$, shown here "unrolled" as in Figure 1d in the main text, where points (n, r) with $r = 7$ depicted "split in half". Each point $(n, r) \in \mathcal{C}_8$ is a column n (unbounded) and a row r (between 0 and $h - 1$). Each arrow from a point $p \in \mathcal{C}$ to a point $q \in \mathcal{C}$ represents that $p \to q$, which means that when a tile binds at position q, one of its glues binds to a tile already at position p. A seed (sequence of h glues) is shown on the left.

[24] I.e. we do not allow so-called "double-strength" glues, nor stronger glues. These strong glues are a feature of the aTAM [12] where we are permitted to have bonds strong enough to bind without cooperating with another glue.

Let $h \in \mathbb{N}^+$ and define $[h] = \{0, 1, \ldots, h-1\}$. Formally, the *(one-way infinite length) height-h cylindrical lattice* is the space $\mathcal{C}_h = \mathbb{N} \times [h]$. We define a relation $\rightarrow$ on $\mathcal{C}_h$, intuitively capturing the idea that for $p, q \in \mathcal{C}_h$, if $p \rightarrow q$, then a tile that binds at position q is binding to a tile already present at adjacent position p. For all $(n, r), (n', r') \in \mathcal{C}$, we write $(n, r) \rightarrow (n', r')$ if $(r' - r) \equiv 1 \mod h$ and either n is even and $n' = n$, or n is odd and $n' = n + 1$. A finite portion of $\mathcal{C}_8$ is shown in Figure S7. Note that each position $p = (n, r) \in \mathcal{C}_h$ where n is either positive or odd has exactly two positions $q_1, q_2 \in \mathcal{C}_h$ such that $q_1 \rightarrow p$ and $q_2 \rightarrow p$ (the two "inputs" of p) and all positions $p \in \mathcal{C}_h$ have exactly two positions $q_3, q_4 \in \mathcal{C}_h$ such that $p \rightarrow q_3$ and $p \rightarrow q_4$ (the two "outputs" of p).

A *tile type* is an unrotatable unit square with four sides, each labeled by a *glue* (often represented as a finite string). We assume a finite set T of tile types, but an infinite number of copies of each tile type, each copy referred to as a *tile*. An *assembly* is a positioning of tiles on the discrete cylindrical lattice $\mathcal{C}_h$. A *height-h seed* is a sequence of h glues $\sigma = (g_0, g_1, \ldots, g_{h-1})$. A tile t *can attach* at position q to an assembly α, producing an assembly β, if α has no tile at position $q \in \mathcal{C}_h$ and either 1) $q = (n, r)$ where $n > 0$ or r is odd, α has tiles t_1 and t_2 at the two respective positions $p_1, p_2 \in \mathcal{C}_h$ such that $p_1 \rightarrow q$ and $p_2 \rightarrow q$, and t matches the abutting glues of both t_1 and t_2 in β, or 2) $q = (0, r)$ where r is even, and t matches glues g_r and g_{r+1} of σ.[25] In this can we say β *is producible from* α *in one step*. For example, Figure 1c in the main text shows a tile that can attach at position $(3, 2)$. An assembly β is *producible* from σ if either 1) β is empty (i.e., has no tiles) 2) there is a producible assembly α such that β is producible from α in one step.

S2.1.1 Terminology for square tile-based self-assembly

Ideally, only assemblies producible by the above definition actually appear experimentally. In reality, sometimes a tile binds when only one glue matches. If we have some assembly α and a tile t in the assembly, the *strength* with which t is attached is the number of neighboring tiles with which t shares a matching glue, and we say an attachment is *strength-k* if the tile is attached with strength k immediately after the attachment. With this definition, tile attachment events at the moment they occur (as described above), are always strength-2. Preventing erroneous strength-1 attachments by one of the four domains on an SST is one of the main concerns addressed by seeded growth (Section S5.1) and preventing erroneous strength-1 attachments by two domains (one matching and one mismatching) is one of the main concerns addressed at the sequence design level of abstraction (Section S4).

Each tile t has four glues, as noted above, the two of glues that t uses are called input glues, and the pair that do not are called output glues. For example, in the tile t in the top left of Figure S8, the two input glues are n and w and the two output glues are e and s.

S2.2 Complete 6-bit IBC tile set before proofreading

Figure 1c in the main text gives the essential idea of how to translate 6-bit, 1-layer IBC gates into square aTAM tiles. Each line in the gate truth table becomes a single tile. Another way to view this transformation is to obeserve that (for the 6-bit case) the tile lattice consists of 8 rows of tiles: 5 rows where each tile has a bit on each of its 4 sides (16 tile types per row), 2 rows where the tiles a bit on 2 sides (4 tile types per row), and 1 row of seam tiles that encode zero bits (1 tile type). This gives $5 \cdot 16 + 2 \cdot 4 + 1 = 89$ tile types.

It is worth noting that the calculation given in Section S1.2 shows that there are 1,288 gates for the class of 6-bit, 1-layer IBCs; yet here we needed only 89 tile types. An intuitive reason why there are fewer tile types than gates is that many pairs of gates share lines of their respective truth tables. Under our transformation from gates to tiles, this means that two distinct gates could map to some of the same tile types.

In the next subsection we will find that when we apply an error-reducing proofreading transformation to our tile set the set the number of tile types will increase from 89 to 355.

[25] Note that we model glues as binding if they are equal, rather than defining some notion of complementarity as with Watson-Crick base pairing. This works in conjunction with the unrotatable constraint on the tile, so that each tile has a well-defined north, east, south, and west side. In reality of course, tiles will rotate. What we are capturing here is that a glue is implicitly also labeled by its direction, so that glue 0 on the north is different from glue 0 on the east, and that we design a north 0 sticky end to be complementary to a south 0 sticky end, but not to east 0, west 0, or north 0. Thus tiles cannot possibly match complementary glues unless they are rotated by the same amount, and it simplifies discussion and figures simply to label glues without also specifying the direction.

S2.3 Complete 6-bit IBC tile set after 2 × 2 proofreading

A primary source of faults in experimental implementations of the aTAM is an *tile attachment error* (called a *growth error* in other theoretical literature on algorithmic tile assembly [2, 25, 11]), defined to be the binding of an "erroneous" tile t_e with input glues g_1 and g_2 to a binding site with two available glues g'_1 and g'_2 that are the input glues of another "correct" tile t_c, such that $|\{g_1, g_2\} \cap \{g'_1, g'_2\}| = 1$. In other words, the correct tile to attach is t_c, which matches *both* input glues, whereas an erroneous tile t_e binds instead, despite matching only *one* input glue. Such an error represents a failure of the intended effect of cooperative binding. In our circuits, such errors typically manifest as bit flips, e.g., a 1 bit appearing or disappearing.[26] See Figure S40a.

A strategy for reducing growth errors is known as *proofreading*, due to Winfree and Bekbolatov [25]. It is a method of transforming one tile set T into functionally equivalent tile set T_p, such that under a realistic model of the chemical kinetics of tile assembly known as the kinetic Tile Assembly Model (kTAM) [2] (see Section S7.6), T_p will provably have a lower rate of tile attachment errors than T.

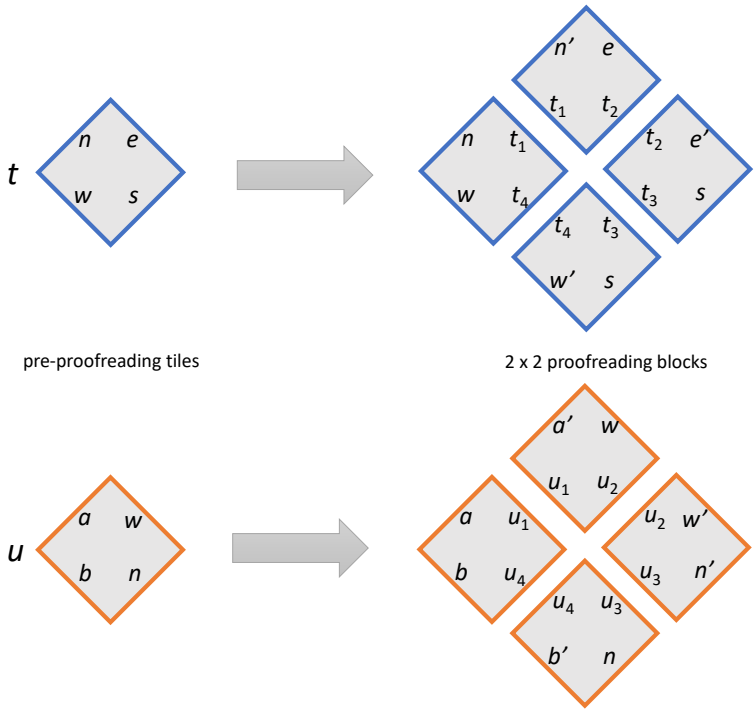


Figure S8: 2×2 proofreading. Each tile type is transformed into four tile types, each of which is called a *proofreading block*. Glues internal to the proofreading block are unique. Glues external to the proofreading block are based on the glues of the tile type, so they may be shared between proofreading blocks. Since each proofreading block side has two glues, there are two versions (primed and unprimed) of each of the glues from the original tile set.

Figure S8 shows an example of 2×2 proofreading on a set of two tile types. Let $k \geq 2$. In $k \times k$ proofreading, we replace each tile type in T with k^2 tile types in T_p. Each tile in the assembly formed by T is (assuming error-free growth) represented by a $k \times k$ block of tiles in T_p. Glues internal to the block are unique to that block and to their relative position within the block. Glues external to the block represent the glue of the tile in T that the block replaces, except that there are k versions of the glue, one for each tile incident to the side of the block. The key idea is that $k \times k$ proofreading logically enforces that if a tile attachment error happens in a block, then $\geq k$ tile attachment errors happen; it is not possible to fill in the block with a number

[26] It is possible for a growth error not to flip a bit, if the function computed by the gate is not injective (a.k.a., one-to-one). Suppose there are two inputs (b_1, b_2) and (b'_1, b'_2) to the gate g such that $g(b_1, b_2) = g(b'_1, b'_2)$, yet either $b_1 \neq b'_1$ and $b_2 = b'_2$, or $b_1 = b'_1$ and $b_2 \neq b'_2$. In this case, the erroneous tile could have the same outputs as the correct tile, in which case the growth error would not cause a logical error in terms of which bits appear on wires in the circuit. The circuit RULE110 (see Section S8.3) has this property: gates 2, 4, and 6 send both of the inputs 01 and 11 to the output 11.

of glue mismatches between 1 and $k-1$. Since tile attachment errors are rare, several tile attachment errors in a row are even rarer, and it is more likely for an erroneously bound tile to detach before the block experiences additional tile attachment errors sufficient to fill in the block. To a rough approximation, if the rate of tile attachment errors in T is ϵ (e.g., $\epsilon = 0.01$), then the thermodynamic rate of tile attachment errors in T_p is about ϵ^k (e.g., $\epsilon^2 = 0.0001$). However, kinetic phenomena prevent the full thermodynamic error reduction from being achieved when $k > 2$ with the described proofreading construction; more advanced proofreading schemes [65, 66] for which error rates provably decrease exponentially with k have been proposed. See [24] for a recent review of the physics of proofreading in self-assembly.

In addition to the theoretically provable error reduction by proofreading, simple variants of the proofreading construction have been experimentally demonstrated to reduce tile attachment errors [67, 8, 10, 11].

S2.3.1 2×2 proofreading for the complete 6-bit IBC tile set

We employ 2×2 proofreading. In our case, T is the set of 89 tiles described in Section S2.2 and T_p is the set of $89 \cdot 2^2 = 356$ tiles obtained by the 2×2 proofreading transformation. We then modify T_p slightly so that the proofreading block straddling the seam of the nanotube (where the unzipper strands will "cut" the tube open; see Section S5.3) has the same tile type in the two positions of the proofreading block that are removed during unzipping[27], leading to a tile set T_p' with only 355 tile types.

S2.3.2 Implementing randomised circuits

This section describes the two techniques we used to implement IBC randomisation via programming of molecular concentrations of SSTs. These are called the "first-tile randomisation" method and the "block randomisation" method.

Let us say that a tile type is "correct" at a given binding site if it can attach by exactly two glues. When there is a unique correct choice at every binding site, the kinetic Tile Assembly Model predicts that algorithmic self-assembly has lowest error rates when the on-rate of the correct tile is just slightly larger than its off-rate [2]. (If they are equal, then no net forward growth occurs, so there is a trade-off between assembly speed and error.) Once tile sequences have been designed and buffer salt concentrations chosen, the primary experimental knob that can be tuned to adjust the off-rate is temperature: higher temperature means a higher off-rate.

When there may be multiple distinct tile types that can correctly bind at a site, the statement must be adjusted: the lowest error rates occur when the rate at which the site becomes occupied (by any kind of correct tile) should be only slightly larger than the rate at which such just-occupied sites become emptied (by the tile disassociating). The difference here is that the rate for occupation of a site now becomes the sum of the individual on-rates for each tile type that can bind correctly.

The on-rate of tiles of a specific type binding to a specific site is typically assumed to be proportional to the concentration of that tile type with a rate constant independent of the tile type and local site geometry. Therefore, if there is more than one correct tile type that could bind at a site, the net on-rate for some correct tile attaching at a site is proportional to the sum of concentrations of correct tile types for the site. Thus, there is a well-defined global per-site on-rate only if there is a single concentration c (in our case $c = 100$ nM), such that, for each binding site, c is the sum of the concentrations of all tile types that can bind correctly at the site. If instead this concentration varies between binding sites, then it's not clear how to choose the growth temperature: either some binding sites will have an on-rate correspondingly larger than the off-rate, leading to a faster-than-necessary growth rate that is associated with higher error rates [2], or other binding sites will have the reverse problem of an on-rate smaller than the off-rate, limiting the growth rate near that site, which provides more time for errors to occur elsewhere [65]. The solution seems simple for randomised circuits: to implement a randomised choice between two gates g_1 and g_2 ($\Pr[g_1] = p$ and $\Pr[g_2] = 1 - p$) corresponding respectively to tile types t_1 and t_2, set concentrations to $[t_1] = (100 \cdot p)$ nM and $[t_2] = (100 \cdot (1 - p))$ nM, so that $[t_1] + [t_2] = 100$ nM, the concentration of deterministic tiles.

[27] This was done because in early experiments, unzipping was a delicate protocol that would sometimes work on one set of sequences but fail on another, so we wanted to ensure that in the final set of sequences, the same unzippers could be used that had worked previously. Although this alteration would disrupt the error reduction achieved by proofreading in a "normal" proofreading block, there is significantly less possibility of tile attachment errors in the proofreading block along the seam. This is because the input and output glues on that block are unique.

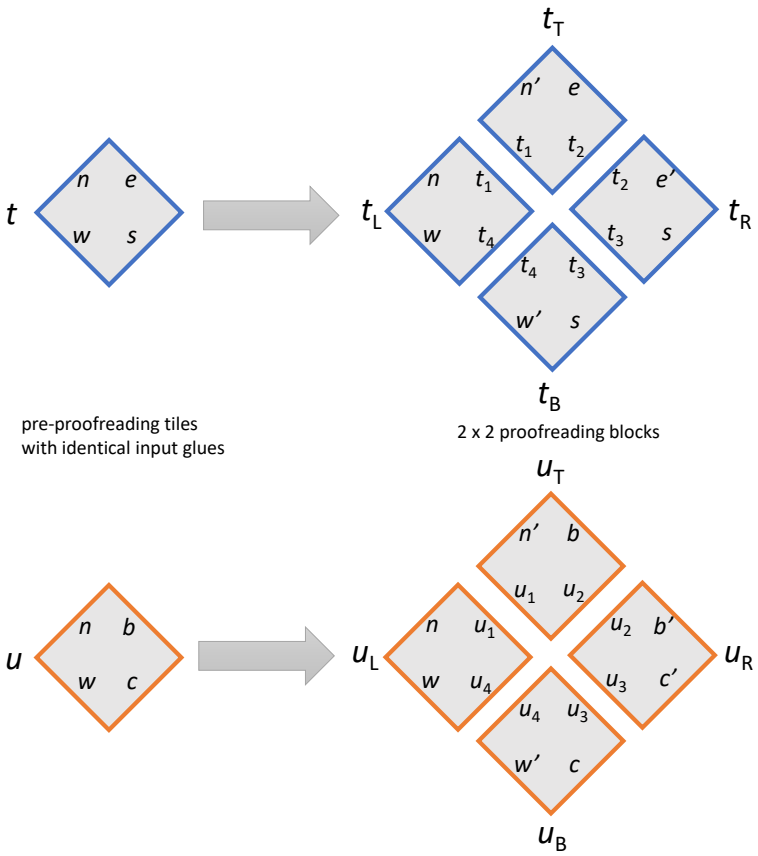


Figure S9: Randomised proofreading. The two tile types t and u share input glues (denoted n and w). This implies that the leftmost tile types t_{L} and u_{L} in each proofreading block share both input glues, but tile types on the top of each proofreading block (t_{T} and u_{T}) share only one input glue n' in common, and similarly for t_{B} and u_{B} sharing only input glue w', and for t_{R} and u_{R}, sharing no input glues.

However, proofreading introduces a twist. Assuming error-free algorithmic growth, where no tile ever attaches by only one glue, then in each proofreading block implementing a randomised gate, only the leftmost position in the block (for instance, tile types t_{L} and u_{L} in Figure S9) actually has randomised binding. Once a choice is made at the leftmost position, for each of the other three positions in the block, only a single tile type can attach by two glues. See Figure S9.

This leads to a dilemma that we illustrate below with a running example: there are at least two obvious ways to set concentrations to implement randomness. In our experiments we used both with success. Nevertheless, as we discuss below, each method has theoretical disadvantages when it comes to suppression of errors.

To set up our simple running example, first note the names of the tile types in the two proofreading blocks in Figure S9. Suppose that we wish the probability of the top proofreading block to be 90%, that is, the associated IBC gate has probability $p = 0.9$. Then assuming a concentration of 100 nM for tile types implementing deterministic gates, we should set $[t_{\mathrm{L}}] = 90$ nM and $[u_{\mathrm{L}}] = 10$ nM. But how to set the other six? We used two methods.

Method 1: The block randomisation method. In our running example, when using the block randomisation method we would set $[t_{\mathrm{T}}] = [t_{\mathrm{B}}] = [t_{\mathrm{R}}] = 90$ nM and $[u_{\mathrm{T}}] = [u_{\mathrm{B}}] = [u_{\mathrm{R}}] = 10$ nM. Hence, in the block randomisation method, all four tiles in a proofreading block have the same concentration. This can have serious consequences. Recall that temperature has been delicately tuned such that for tiles at 100 nM, the on-rate and the off-rate are very near to each other, i.e. the crystal growth is near its equilibrium melting

temperature. With these rates, growth is only slightly biased forward; tiles attach and fall off again, on average arriving more often than they fall off, so the crystal grows over time. At left site, either u_{L} or t_{L} may attach correctly, so the arrival rate of *some* correct tile is still as would be expected for 100 nM, and thus the growth rate through that site will remain balanced with a slight forward bias. However, at the top, bottom, and right sites, the situation is different, e.g., with $[u_{\mathrm{T}}] = 10$ nM, the rate of filling the site with u_{T} (and there is no other two-glue-matching choice) is 10 times lower than the rate of emptying the site (which is roughly equal to the rate of filling the site with a 100 nM tile type). Thus, self-assembly stalls (i.e. temporarily pauses) when attempting to fill this proofreading block: the top, bottom, and right tiles will fall off faster than they can be filled. Another way to look at it is that a just-attached tile u_{L} will fall off 10 times faster than it will be secured by the completion of the rest of the proofreading block. Thus, crystal growth is likely to stall after u_{L} attaches, with the most frequent result being that u_{L} falls off completely. The stalled growth, in turn, provides more opportunities for errors elsewhere before the nanotube completes its growth. Also note that the competing block (with tiles $t_{\mathrm{L}}, t_{\mathrm{T}}, t_{\mathrm{B}}$, and t_{R}) will stall less because 90 nM is only slightly less than 100 nM – and therefore, a second major problem is that the probability of successfully completing the block with the lower-concentration tiles ($u_{\mathrm{L}}, u_{\mathrm{T}}, u_{\mathrm{B}}$, and u_{R}) will be substantially lower than 10%. On the other hand, an advantage of the method is experimental ease; one simply prepares 4-tile proofreading block-mixes in advance and then combines pairs of block-mixes at appropriate concentration ratios.

Method 2: The first-tile randomisation method. Now, in our running example, suppose instead that we set $[t_{\mathrm{T}}] = [t_{\mathrm{B}}] = [t_{\mathrm{R}}] = [u_{\mathrm{T}}] = [u_{\mathrm{B}}] = [u_{\mathrm{R}}] = 100$ nM. Then $[c_{\mathrm{T}}] + [t_{\mathrm{T}}] = 200$ nM, and $[u_{\mathrm{T}}] = 100$ nM. Hence, in the first-tile randomisation method, three of the tiles in a proofreading block are at one concentration and one of the tiles (the "first", or leftmost, tile) is at some other concentration. The advantage of this method is that now, every site in the proofreading block (and thus every site in the lattice) has balanced rates for an empty site being filled by some correct tile and for a just-filled site to become empty, and thus the growth process never stalls unless there is an actual error. Thus, the tile attachment ratios for the first tile, as set by their concentrations, is preserved for the probability ratio of complete blocks, as desired. A disadvantage is that there is more bench-work involved in preparing first-tile randomisation samples than block randomisation method samples[28].

In summary, the first-tile randomisation method appears superior in theory and is the recommended approach for implementing randomised tile assembly with proofreading, although the block randomisation method was also tried for some circuits, and successfully so, in experiments performed before we fully appreciated the potential problems with the method.

Choice of randomisation method in our experiments. The block randomisation method was used in the implementation of the following randomised IBCs: Rule110Random, LazySorting, LazyParity, RandomWalkingBit, AbsorbingRandomWalkingBit, and LeaderElection. All of these circuits involved randomized gates with choice probability exactly 0.5, which the block randomisation method is expected to implement faithfully, although with some stalling that could increase error rates. The first-tile randomisation method was used in the implementation of the following randomised IBCs: DiamondsAreForever, Waves, and FairCoin. These circuits employed randomised gates with probabilities varying from 0.1 to 0.9, which the first-tile randomisation method is expected to implement faithfully and with low error rates. It should be noted that all randomised circuits that we implemented had at most two choices per gate position. Section S8 has the details of what gate probabilities were used for each randomised circuit. In contrast to the theory, experimentally measured error rates were not statistically distinguishable between circuits that used the two randomisation methods.

Although the first-tile randomisation method appears superior to the block randomisation method, it is not clear whether first-tile randomisation is optimal over *all* possible settings of tile concentrations. It is an open theoretical question to determine, given a tile assembly system using both randomisation and proofreading, the tile concentrations minimising tile attachment errors, according to the kTAM. Some work has been done in the special case without randomisation or proofreading, assuming (unlike in our setup)

[28]In block randomisation all four tiles within a proofreading block are at the same concentration so they can be pre-mixed in advance and the mix can then be used in the appropriate ratio relative to other proofreading-block mixes. Whereas, in first-tile randomisation, we treat the first/leftmost tile of each randomised proofreading block separate from the other three tiles in that block – which may necessitate extra experimental steps and/or dealing with small volumes appropriate for a single tile.

that every assembly has only a single site where a tile can attach by two glues [68].

Quantitative effects of the block randomisation method. Here, we briefly examine the theoretical consequences of block randomisation from the perspective of thermodynamic equilibrium. Tile self-assembly is inherently a kinetic process and is not guaranteed to achieve equilibrium on the time scale of an experiment. However, when performing self-assembly near the melting temperature of a crystal, equilibrium assessments can provide excellent estimates of the relative concentrations and relative probabilities of related assemblies, such as algorithmic crystals with and without algorithmic errors. Further, such error estimates may in many cases be taken as lower bounds for the error rates of crystals grown in non-equilibrium conditions, where kinetic effects are expected to increase the error rate.

First, let us use this approach to estimate the error rate for a deterministic process. Let each tile type have concentration c, and assume that this concentration is constant during growth. Consider a proofreading block u with tiles $u_{\mathrm{L}}, u_{\mathrm{T}}, u_{\mathrm{B}}$, and u_{R} and a set of "mismatch blocks" $v^1, v^2, \ldots v^m$ that each share one input side (either one) with u. Assume, as is the case for the IBC tile set, that if a mismatch block is fully incorporated in a location where u should be incorporated, further growth does not require any further mismatches. We will consider a correct assembly A that contains block u somewhere in the interior and that u is fully surrounded, and we will consider an assembly A_i that has the same shape and initial portion as A, but with u replaced by some v^i and the following growth of A_i containing no further errors. Thus, each mismatch assembly A_i will have exactly two mismatched binding domains, and because the circuit will be processing different information subsequent to incorporation of the erroneous block, the tiles in A_i downstream of v_i may be different from those in the same location in A. We are interested in the relative probability of A versus an assembly similar to A but with an error at the noted position of u, that is the equilibrium concentration ratio

$$\epsilon^{det} = \frac{\sum_{i=1}^{m}[A_i]}{[A]}$$

will be taken as a proxy for the per-block error rate. This estimate neglects other error mechanisms, and for simplicity ignores the concentrations of mismatched assemblies in the denominator; these will indeed be negligible contributions when the error rate is small.

Recall that for a binding reaction $X + Y \rightleftharpoons Z$ with binding energy $\Delta G^\circ < 0$, the equilibrium concentrations satisfy $\frac{[X][Y]}{[Z]} = e^{\Delta G^\circ/RT} u_0$ where $u_0 = 1$ M if that's the reference concentration for the measured energies. Based on this, and working out a telescoping product, the concentration of an assembly A with seed σ and tiles $t_1, \ldots, t_N$ is

$$[A] = [\sigma] \left(\prod_{j=1}^{N} \frac{[t_j]}{u_0} \right) e^{-\sum_{j=1}^{N} \Delta G_j^\circ / RT} \tag{3}$$

where $\Delta G_j^\circ < 0$ is the energy with which tile t_j bound to the assembly[29]. In our model we will consider binding energies to be additive, with each correct bond contributing $-G_{se}RT$ and each mismatched bond contributing $-sG_{se}RT$ with $s = 1/2$. Our conclusion in the end will not depend on s, but this is left as an exercise to the reader. Under these assumptions, the concentration for an assembly with one mismatched block is

$$[A_i] = [A]e^{-2(1-s)G_{se}} = [A]e^{-G_{se}}$$

and thus the (fairly accurate) proxy for the per-block error rate is

$$\epsilon^{det} = me^{-2(1-s)G_{se}} = me^{-G_{se}}$$

which reduces to the familiar value when $s = 0$. For the deterministic IBC, $m = 2$ for the 2-input, 2-output gates.

Now, we turn to the case of a non-deterministic assembly process with block randomization. For the site in question, assume that correct growth would allow either block u with tiles $u_{\mathrm{L}}, u_{\mathrm{T}}, u_{\mathrm{B}}$, and u_{R} and probability p, or block t with tiles $t_{\mathrm{L}}, t_{\mathrm{T}}, t_{\mathrm{B}}$, and t_{R} and probability q, where $p + q = 1$. Therefore $[u_{\mathrm{L}}] = [u_{\mathrm{T}}] = [u_{\mathrm{B}}] = [u_{\mathrm{R}}] = pc$ and $[t_{\mathrm{L}}] = [t_{\mathrm{T}}] = [t_{\mathrm{B}}] = [t_{\mathrm{R}}] = qc$, where c is the concentration for deterministic

[29] Note that because of detailed balance, the result of this calculation is independent of the order in which the tiles attach.

tiles, as above. The set of "mismatch blocks" $v^1, v^2, \ldots v^m$ will have tiles with concentrations $f_1 c, \ldots, f_m c$, with $0 < f_i \leq 1$ depending on the probabilities of the gates from which the competing blocks are derived. We will do two things: first, we will consider the relative (equilibrium) probabilities of correct assemblies incorporating u versus t, and second, we will look at the error rate based on the relative probabilities of incorporating mismatch blocks.

Let A_u be a correct assembly (i.e. no mismatches) with the shape of A but with u incorporated at the site of interest, and let A_t be a correct assembly of the same shape but with t at that site, and further assume that the distribution of tile type concentrations in the rest of A_u and A_t are the same. This is true, for example, if the given site is the only randomized choice in the assembly (so all other tile types have concentration c), and it is statistically likely if the parts of A_u and A_t before and after the given site involve many independent random choices (as would be the case for a random walk, for example). In this case, applying equation (3) yields

$$\frac{[A_u]}{[A_t]} = \frac{[u_{\mathrm{L}}] \cdot [u_{\mathrm{T}}] \cdot [u_{\mathrm{B}}] \cdot [u_{\mathrm{R}}]}{[t_{\mathrm{L}}] \cdot [t_{\mathrm{T}}] \cdot [t_{\mathrm{B}}] \cdot [t_{\mathrm{R}}]} = \frac{p^4}{q^4}$$

because all the other concentrations cancel out and the overall energies are the same. Note that this fraction is not the desired p/q except when $p = q = 1/2$.

Making the same assumptions about mismatch assemblies $A_1, \ldots, A_m$, we are interested in

$$\epsilon^{rand} = \frac{\sum_{i=1}^{m} [A_m]}{[A_u] + [A_t]} = \frac{\sum_{i=1}^{m} f_i^4 c^4 e^{-2(1-s)G_{se}}}{p^4 c^4 + q^4 c^4} = \frac{\sum_{i=1}^{m} f_i^4 e^{-2(1-s)G_{se}}}{p^4 + q^4} \ .$$

Whether this is higher or lower than the deterministic case depends on the probabilities involved.

Consider, for example the RandomWalkingBit circuit. A gate position in the middle uses randomization of a COPY gate and a SWAP gate with probability 1/2. To analyze in terms of proofreading blocks, we must consider the lookup table entries. COPY is $(00 \to 00, 01 \to 01, 10 \to 10, 11 \to 11)$ while SWAP is $(00 \to 00, 01 \to 10, 10 \to 01, 11 \to 11)$. Let u be the $01 \to 01$ block and t be the $01 \to 10$ block, both with the same probability $p = q = 1/2$. Then the mismatch blocks would be $v_1 = 00 \to 00$ with probability $f_1 = 1$ (because both COPY and SWAP use it) and $v_2 = 11 \to 11$ with probability $f_2 = 1$ (for the same reason). Thus, with $s = 1/2$, here

$$\epsilon^{rand} = \frac{(1+1)e^{-G_{se}}}{2^{-4} + 2^{-4}} = 16e^{-G_{se}}$$

which is 8 times higher than the deterministic error rate for the same physical conditions. If in contrast we consider the error rate when block $00 \to 00$ is incorporated (which, though at a randomized gate location, nonetheless is deterministic for those inputs!), then the mismatch blocks are now $01 \to 01$ and $10 \to 10$ from the COPY gate and $01 \to 10$ and $10 \to 01$ from the SWAP gate, with concentration fractions $f_1 = f_2 = f_3 = f_4 = 1/2$. So we instead get:

$$\epsilon^{rand} = \frac{(2^{-4} + 2^{-4} + 2^{-4} + 2^{-4})e^{-G_{se}}}{1^4} = \frac{1}{4}e^{-G_{se}}$$

which is 8 times lower than the deterministic error rate for the same physical conditions. In a long RandomWalkingBit assembly with a single randomly walking 1, roughly 2/5 of blocks[30] will assemble with the 8 times higher error rate, and the others will assemble with the 8 times lower error rate, for an overall average error rate of

$$\epsilon^{rand} \approx \left(\frac{32}{5} + \frac{3}{20}\right) e^{-G_{se}} = 6.55e^{-G_{se}}$$

which is more than 3 times higher than the deterministic error rate of $2e^{-G_{se}}$. As this prediction depends on details of gate randomisation within the circuit and of the expected resulting patterns generated by correct growth, other circuits could have higher or lower predicted error rates. Regardless, one can anticipate that the worst-case error rates for gates with more biased probabilities will be worse that the estimates here for

[30] Each layer has 7 gates, 5 of which are 2-input gates. Ignoring the 1-input gates and treating the case where the random walking 1 never goes through them, in each layer there will be three 2-input gates that receive input 00, and two 2-input gates that receive input 01 or 10.

$p = q = 1/2$, and that the consequences of non-equilibrium phenomena – such as stalling at randomised gates – will introduce additional disruptions to correct growth.

It is therefore striking that, as mentioned above, we did not observe a statistically significant increase in error rates for the experiments that employed the block randomisation method.

If one attempts an analogous analysis for the first tile randomization method, the gate choice probabilities work out correctly for all values of p, and the error rates match the deterministic case.

S3 System design abstraction level: binding-domain schematics

In this section we give the binding-domain schematics level of abstraction, which is below the abstraction level of aTAM proofreading tiles and above that of DNA sequence design.

S3.1 Strand-level system design of SST lattice, input-adapter strands, seed attachment

Concerns at this level of abstraction include choosing (a) appropriate domain lengths (in number of bases) for good DNA crossover positions between helices and for structural stability, for SSTs, input adapters, and origami seed staples/scaffold, and (b) appropriate positions on SSTs for biotin modifications. As shown in Figure 1 in the main text, each square tile, post-proofreading, is mapped to an abstract SST strand. We used the SST motif from ref. [13], which has domains of length 10 and 11 bases. Altogether 355 SSTs were designed. The number 355 was calculated in Section S2.3.1, at the abstraction level of 2×2 proofreading tiles; here at the SST binding-domain level of abstraction we simply convert each square (proofreading) tile to an abstract SST strand with four binding domains.

Figure S10 gives the domain-level design of our SST set, and the caption describes some of the key features. SST strands are designed to grow a nanotube lattice, so that the top of the SST lattice in Figure S10 is

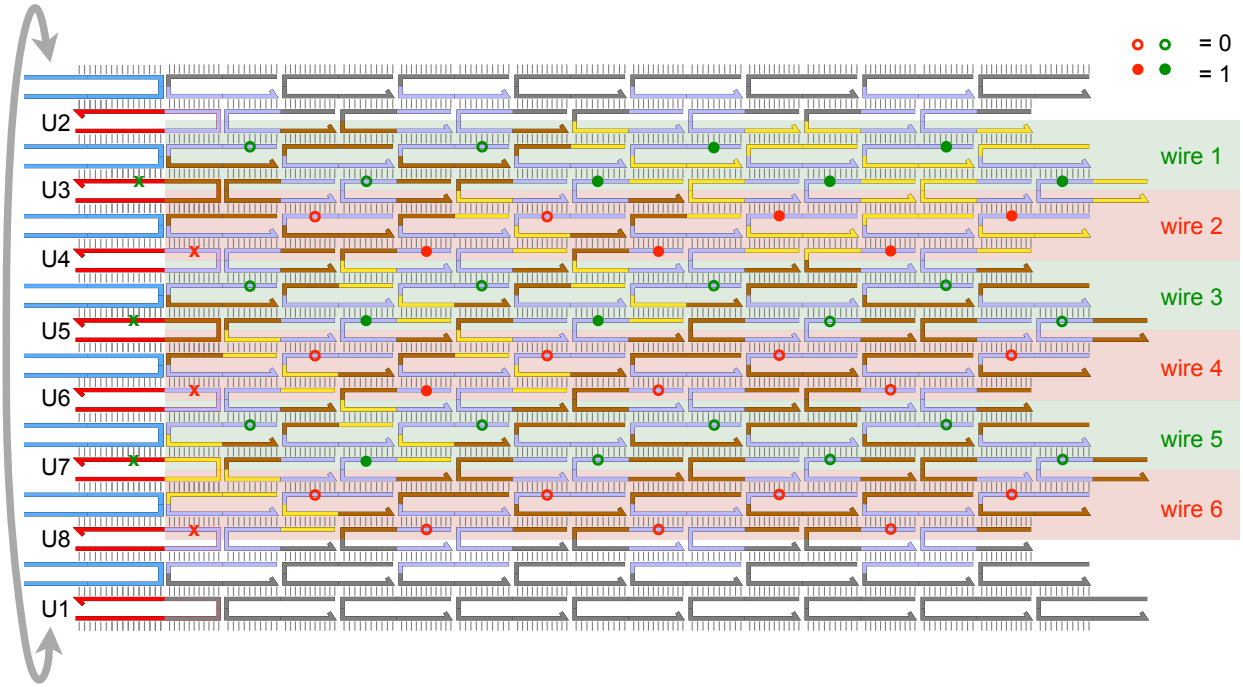


Figure S10: SST lattice showing from left to right the DNA origami seed scaffold (light blue), input-adapter strands (red, plus other colours), and SSTs (yellow, brown and blue), seam (grey), seam block (grey), and biotin locations (red and green disks/circles, coloured according to their wire index being even or odd). Each of the four **domains** on an SST are colour coded: yellow represents bit 1, brown represents bit 0, blue represents a domain that is unique to the proofreading block it is in. The presence of **biotins** is indicated by red and green solid disks, and is used to denote the encoding of a 1 bit on the proofreading block *output* domains that are on the same wire. A biotin is present on a strand if its closest bit-encoding output domain, along the same strand, is yellow (encodes bit 1), and otherwise the biotin is not present (denoted as a hollow circle, to help visualize all possible positions where biotins *could* be on other arbitrary computations). Relevant portions of the DNA origami seed **scaffold** strand are shown as light blue on the left-hand side. **Input-adapter** strands are shown with red on the domains connecting to the scaffold, and the domains binding to tiles are colored using the same bit convention as the tiles. Although input adapters do not have biotin modifications, red or green '`x`'s are used to indicate locations were biotins would have been had we chosen to include them.

wrapped around to bind to the bottom by a special "seam" strand. The input seed structure is a barrel-shaped DNA origami. Input adapters are attached to the scaffold by 32 bases, see Section S3.3 for more details on the input-adapter design. Biotin positions were chosen according to the following criteria:

1. Biotin modifications should be internal to a strand (rather than at the 3′ or 5′ end) for binding-energetic reasons and placed on a T base that is facing up (away from the mica) when the nanotubes are unzipped and stuck to mica (see Section S5.5.1).

2. Biotin modifications should be on domains that do not encode bits, i.e. on domains that are internal to a proofreading block. The reason is as follows. During the process of self-assembly a tile that has as its two input glues two domains internal to a proofreading block, and is binding to a correctly formed lattice, matches on either no inputs sides or else both input sides. Tiles that bind with two input glues that encode bits do not have this property; they match on either no input sides, one input side or both input sides. This situation is a key consequence of growth via algorithmic self assembly, and makes bit-encoding input sides more energetically sensitive than (hardcoded) internal proofreading block domains. Since we know that biotins affect the energetics of binding (see Section S5.5.1), and since our estimates on that effect could be coarse and inaccurate, we chose to not put biotins on bit-encoding domains.

The domain level design for the DNA origami seed is given in Figure S11, for unzipper strands is given in Figures S32 and for guard strands is given in Figure S33.

Naming conventions for tiles, glue and proofreading blocks are given in Section S9.

S3.2 DNA origami seed design

Here we describe our 16-helix barrel-shaped DNA origami design. A number of other DNA origami barrel designs have appeared in the literature [69, 70, 71, 26].

The DNA origami seed design is shown in Figure S11. The seed is shaped like a tube, and before imaging is "unzipped" (through DNA strand displacement to remove staple crossovers between the helices numbered 2 and 17) so as to lay flat into a rectangle when deposited on mica. The DNA origami was designed using cadnano version 2.2 ([72], http://cadnano.org/), although Figure S11 was generated using cadnano version 1. Every insertion represents two bases that are not indicated in Figure S11. Full staple sequences are specified in Section S12.

Figure S11 depicts the side of the rectangle on the *outside* of the tube; equivalently, the side of the rectangle that faces down onto the mica. Therefore, Figure S11, rotated 90° clockwise, appears "upside down" compared to the images of the DNA origami seed depicted in most AFM images in this paper. That is to say, in both Figure S11, and in the images in which the view is rotated to make the origami barcode appear right-side up, the tiles grow from the right end of the origami. However, the barcode would appear upside down if depicted in Figure S11; Figure S15 depicts the positions of the staples with a vertical reflection, so that in Figure S15, the helix labeled "2" in Figure S11 is on the bottom and helix 17 is on the top.

S3.2.1 Designing shape of origami seed to be a tube

The design shown in Figure S11 is close to the original designs of Rothemund [14] for making rectangles with 24 or 32 helices. In this section we discuss how we modified this design to create a tubular shaped origami, which we subsequently call a "barrel" to distinguish the DNA tile nanotube that grows from it, although both are shaped like a tube.

First, suppose that the crossovers from helix 2 to helix 17 did not exist in Figure S11. The cadnano design has "insertions" and "deletions", which represent that a position visually depicted in the software, which normally represents a single base pair, instead represents two or zero base pairs, respectively. The insertions and deletions have two independent functions:

twist correction: With no insertions or deletions, all staple crossovers are 16 bp apart. To create a flat rectangle, all staple crossovers should be coplanar. This means each staple crossover between two helices should point the opposite direction as the one to the left, so an odd number of half-turns between the crossovers is required. Three half turns is close to 16 base pairs, implying 10.67 bp/turn.

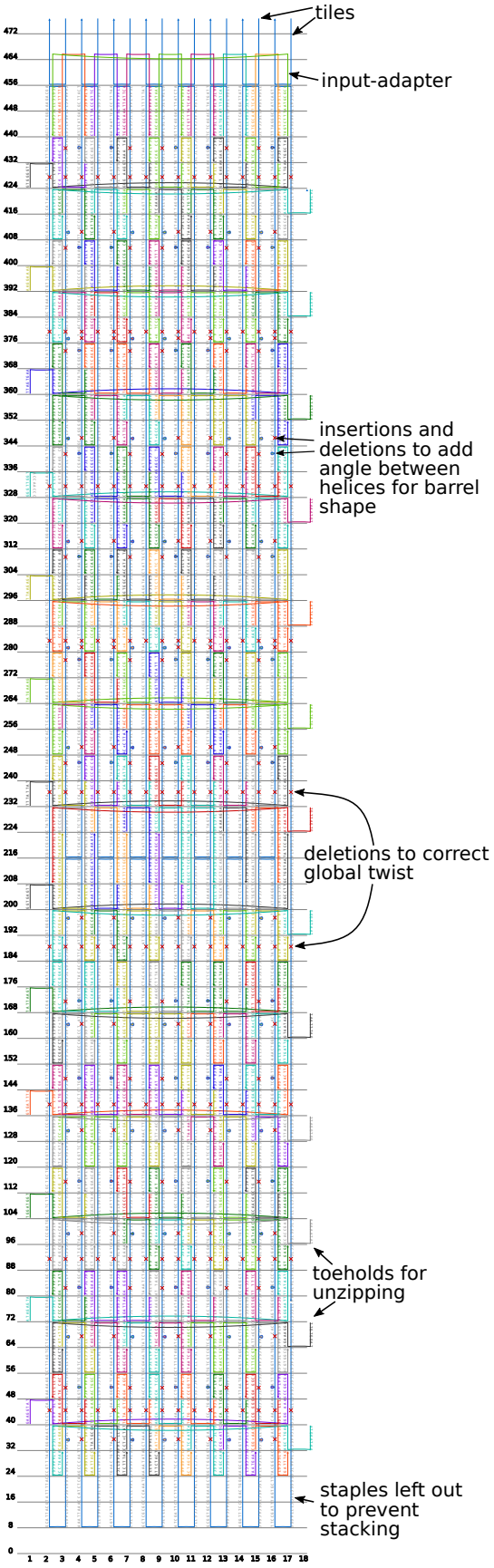


Figure S11: Design of DNA origami seed in cadnano. Helices of origami are numbered 2 through 17 (shown on bottom); "Helices" numbered 1 and 18 are used to show toehold extensions on some staples. Horizontally oriented columns, numbered 8 through 455 and shown on left, are position of bases along the long axis. Staple names show their starting and ending helix and column, e.g. the staple beginning (with its 5' end) at helix 18 and column 39 is named `ST18[39]3[31]`, which is how staple sequences in Section S12 can be cross-referenced with this figure. Due to a limitation in cadnano, staples with insertions (pictured as small loop-outs) do not show the two base pairs in the insertion region; see Section S3.2.1. Tiles are depicted as blue strands on the top side of the origami, starting at column 456. Toeholds on some staples, used for unzipping (Section S5.3.1), shown on "helices" 1 and 18. The pattern of insertions and deletions is explained in Section S3.2.1.

Compared to B-form DNA's natural helicity of 10.4–10.5 bp/turn [73], this undertwist in each helix induces a global supertwist to the rectangle. (See Figure S13, top.)

We use a method due to Woo and Rothemund [74] to counteract the supertwist. By using deletions in every helix, in between one out of every three staple crossover pairs (at columns 44, 91, 139, 188, 236, 283, 331, 379, 427), the twist between these modified pairs of adjacent crossovers is 10 bp/turn instead of 10.67. Averaging over all pairs of adjacent crossovers along a helix (1/3 having 10 bp/turn, and the remaining 2/3 having 10.67 bp/turn), the average value is 10.44 bp/turn, closer to the natural unstrained helicity of B-form DNA. Thus, with only these deletions (and lacking the crossovers between helices 2 and 17), one expects the origami to be shaped like a flat rectangle. (See Figure S13, middle.)

barrel formation: It is possible to take a flat DNA origami rectangle and bend it into a tube simply by adding crossovers between the top and bottom helix [26]. However, this conformation would be strained. We desired a DNA origami seed that, even in the absence of crossovers linking the top and bottom helices of the rectangle, would naturally bend into a tubular shape (though without the crossovers, would remain a rectangle topologically). As demonstrated by Han et al. [71], this can be done by adjusting the relative positions of crossovers between helices, so that if all crossovers between helix i and $i+1$ lie in a plane P, the crossovers between helix $i+1$ and $i+2$ lie in a plane P' at a nonzero angle to P; see Figure S12.

This induced curvature was desired for two reasons:

consistent orientation: There are two ways to roll a rectangle into a tube by connecting its top and bottom edges. Only one of these is correct (the other conformation would present the sticky ends in the reverse of their intended cyclic order). Bending the rectangle favors the correct orientation.

encouraging seeded growth: Previous work using DNA origami as seeds for double-crossover (DX) tile ribbons and nanotubes inferred an energetic barrier for the initial layer(s) of tiles growing from the seed, presumed to be due to a mismatch between the geometry and structure of the DNA origami and the DNA tile lattice [8, 26]. For DNA origami as seeds for SST nanotubes, we hypothesized that reducing the strain of the DNA composing the barrel would lead to better seeded growth. Despite this effort, the presence of a remaining energetic barrier to SST growth from our DNA origami seed is supported by our estimate that 38.9% of seeds experienced little to no growth of tiles; we presume that the barrier would have been worse with a seed that had more strain. However, we have not confirmed this hypothesis.

These insertions and deletions achieve a global tubular shape in the following way (this is best understood by following along in Figure S11): along the length of a helix, between each pair of adjacent crossovers, an insertion is added if the crossover to the left is up and the crossover to the right is down. A deletion is added if the crossover to the left is down and the crossover to the right is up. If we assume 10.5 bp/turn, then each base adds $360/10.5 = 34.3°$ of rotation. This insertion/deletion then has the effect of shifting the angle between these adjacent crossovers by 34.3°, making this angle 145.7° instead of 180°, angled into the page. Rather than do this on every helix, it is done on two out of every three helices, starting on helix 3, so 10 total out of 16. This has the effect of making the total rotation from the top helix to the bottom helix equal to $10 \cdot 34.3° = 343°$, close to the 360° needed to bring the top and bottom helices into contact. Thus the expected shape, even in the absence of crossovers connecting the top and bottom helix, is a barrel. (See Figure S13, bottom.)

The single stranded tiles are symmetric, so it is expected that their cross-section is a regular 16-gon, with an angle of $180 - 360/16 = 157.5°$ in every triple of helices. (More precisely, lines drawn between the centers of adjacent helices form a regular 16-gon.) One would expect some strain in the tiles in a 16-helix tube, since single-stranded tiles using alternating domain lengths of 10/11 have a "natural" conformation as a 12-helix tube due to the 150° angle expected between helices [13], compared to the 157.5° induced by a 16-helix tube. (See Figure S12, left side). However, the strain should be symmetric around the circumference of the tube, so in the cross-section, a regular 16-gon is still expected.

In contrast, the strategy used in the DNA origami seed, alternating coplanar helices (180°) every third helix, with an angle of 145.7° on the other two, should create an irregular 16-gon. (See Figure S12, right side). It is not known whether this discrepancy between barrel shapes accounts for the problems with

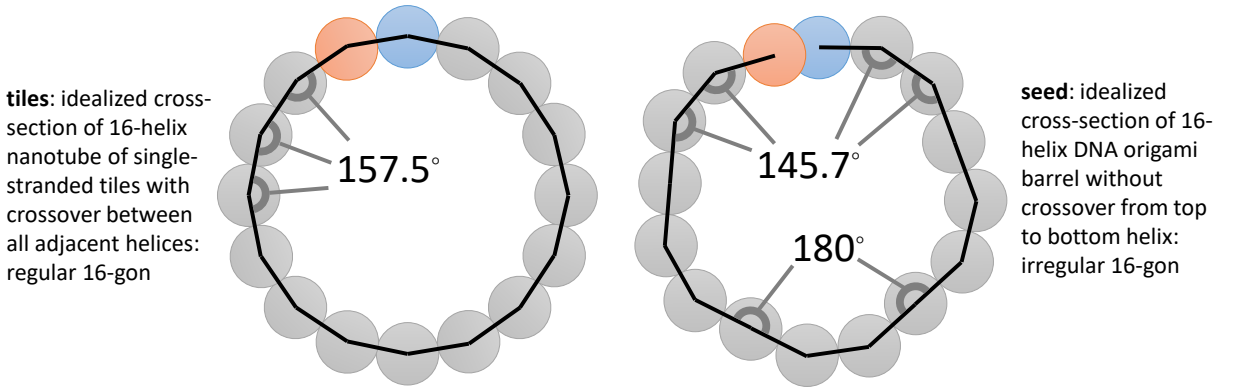


Figure S12: Cross-sections showing designed (not experimentally measured) angles between helices for (**left**) single-stranded tile tube with a crossover between all pairs of adjacent helices, and (**right**) DNA origami with *no* crossover between top helix (blue) and bottom helix (red). Due to the asymmetry of the origami design, it's not clear what angles to expect when the top and bottom helix are joined by crossovers, but certainly some will be closer to 180° and some closer to 145.7°, so the 16-gon would still be irregular.

seeded growth that we observed. It is possible that using crossovers between the top and bottom helices of a naturally rectangular DNA origami, to force the rectangle into a tube shape (as demonstrated by Mohammed and Schulman [26]), or using an origami with 150° angles on *every* helix (naturally favoring a 12-helix barrel) would have a lower energetic barrier to seeded growth than our design. Although the design of Mohammed and Schulman is more strained, it is also more symmetric, so possibly a better fit for the tiles. However, we note that Mohammed, Velazquez, Chisenhall, Schiffels, Fygenson, and Schulman [75] observe a similar phenomenon in which 60%–80% of the seeds have no significant growth ([75, SI Figure S11]), although with a different multi-stranded tile motif known as a double-crossover tile [76].

The method in these papers [26, 75] to bias the orientation of the rolled barrel is to place DNA hairpins (extensions of some staple strands) on one side of the origami. Steric hindrance and electrostatic repulsion then makes it energetically unfavorable for the hairpins to be on the inside of the barrel. This method is not appropriate for our experiments, in which we want to flatten the origami seed onto mica (with the "outside" facing down), with digits written on the up-facing surface using biotin, to which streptavidin attaches (as discussed below). For this we require the down-facing surface of the origami to be flat.

Modeling of effect of insertions and deletions on barrel shape. Figure S13 shows the shape and strain predicted by CanDo.

Experimental evidence for barrel shape of origami seed. If the DNA origami seed is naturally shaped like a barrel, then the following prediction is sensible. After being "unzipped" so that there are no crossovers between the top and bottom helix, the origami will remain curled as in Figure S13(bottom), so most origamis should contact the mica surface first on the "outside" of the barrel. Electrostatic attraction would then pull the rest of this side of the origami down flat, with the inside of the barrel facing up.

We modified 11 staples with a 5′ biotin[31] (to which streptavidin can be attached after mica deposition to determine the biotins' location) to create an asymmetric pattern on the origami: a letter σ. Each 5′ end is at a point where the backbone is oriented toward the inside of the barrel. If the inside of the barrel faces up, then the σ should appear in the proper orientation. If the inside of the barrel faces down, then the σ should either appear backwards or perhaps not at all due to inaccessible biotins, since biotins would face towards the mica surface in this case—although data from experiments on the complete 6-bit IBC tile set suggests that streptavidin will eventually bind even when biotins are attached on the inside of a rolled-up

[31] For the σ labeling on the origami, the relevant staples were extended by TT (two T nucleotides) prior to the biotin on the 5′ end. However, the later barcode digit patterns on the origami did not include the TT extension.

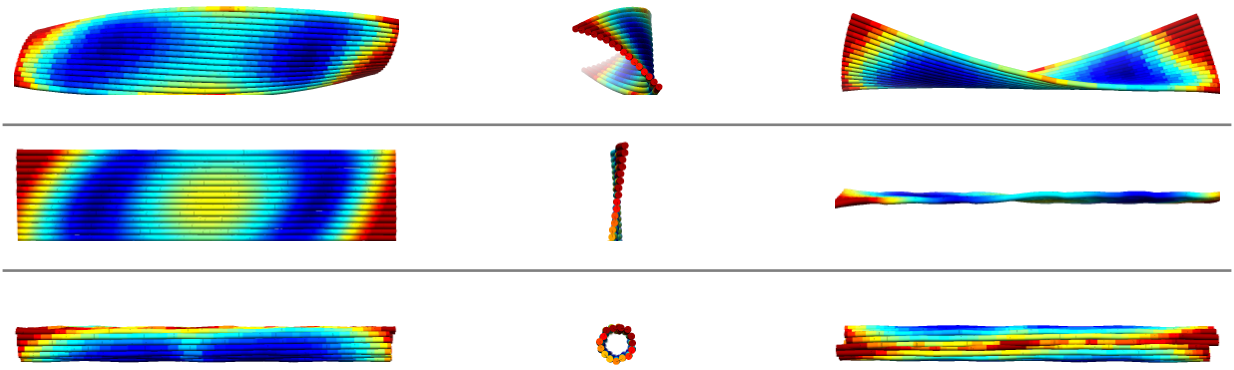


Figure S13: Predictions using CanDo [77] (https://cando-dna-origami.org/) of the shape and thermal fluctuations (red implies more fluctuations) of the DNA origami seed. Three structure predictions are shown, each from three different angles. In all structures, no crossovers exist between the top and bottom helix, to predict the structure if those crossovers were not pulling the origami into a barrel shape. (At the time these predictions were produced, the CanDo model made certain assumptions about helix layout that would be violated by including these crossovers.) **top:** Twisted rectangular seed, using staples shown in Figure S11, but lacking any insertions or deletions. **middle:** Twist-corrected rectangular seed, with only the deletions at columns 44, 91, 139, 188, 236, 283, 331, 379, 427, labeled "deletions to correct global twist" in Figure S11. **bottom:** Twist-corrected and barrel-curved rectangular seed, with all deletions and insertions shown in Figure S11.

tube. This is consistent with previous studies wherein biotins on short DNA linkers[32] have been observed to allow binding by streptavidin when only the opposite side of origami is exposed [78, 79].

Figure S14 confirms this prediction. The vast majority show the tail of the σ on the expected side of the origami. The remainder are missing too many streptavidins to indicate the orientation unambiguously; none show a reversed σ.

S3.2.2 Rotation of scaffold strand

The M13mp18 scaffold strand is a circular strand of DNA consisting of 7249 nucleotides. The "nick" shown at helix 17, column 77 of Figure S11 is a fiction used by the cadnano software. In reality, the number of bases corresponding to the scaffold strand in Figure S11 is only 7024, not depicting a single-stranded loop of length 245 that occurs between the bases at columns 77 and 78 of helix 17. As M13 is a fixed DNA sequence, the only "sequence design" for the origami core involves choosing the rotation of the circular sequence at which to start where the "5′ end" is shown in Figure S11. We start at position 5588 of M13 relative to the starting position given in the GenBank entry for M13mp18; https://www.ncbi.nlm.nih.gov/nuccore/X02513.1, GenBank submission X02513.1. As observed by Rothemund in his original DNA origami design [14], a hairpin with a stem length of 20 base pairs is known to occur from positions 5515 to 5557, so this choice of rotation implies that the hairpin will not be attached to any staples, thus will not compete with the staples to destabilize the structure. Note also that this hairpin is positioned on the far end of the seed from the input-adapter strands, to minimize the odds that it interferes with initial tile binding.

S3.2.3 Seed barcode design for multiplexed AFM readout

Python code was written that allows the experimentalist to easily specify which staple positions (pixels) are "on" and should be labelled with biotins (see Figure S15) and which are "off". From this, 96-well pipetting plate maps were automatically generated to allow the user to easily select out which staples should be 5′-biotin-labelled and which not. For each 3-digit origami seed label to be used in an experiment (e.g. 001, 332, etc.), 208 staples were mixed (in purified water) to give a stock staple mix with a targeted concentration of

[32] Wu et al [78] tested poly-T linkers of lengths from $n = 0$ to 10 nucleotides with 5′ biotin on a C6 linker, like ours, and found minimal labeling within an hour for $n = 0$, but measurable (length-dependent) kinetics for labeling at all other lengths.

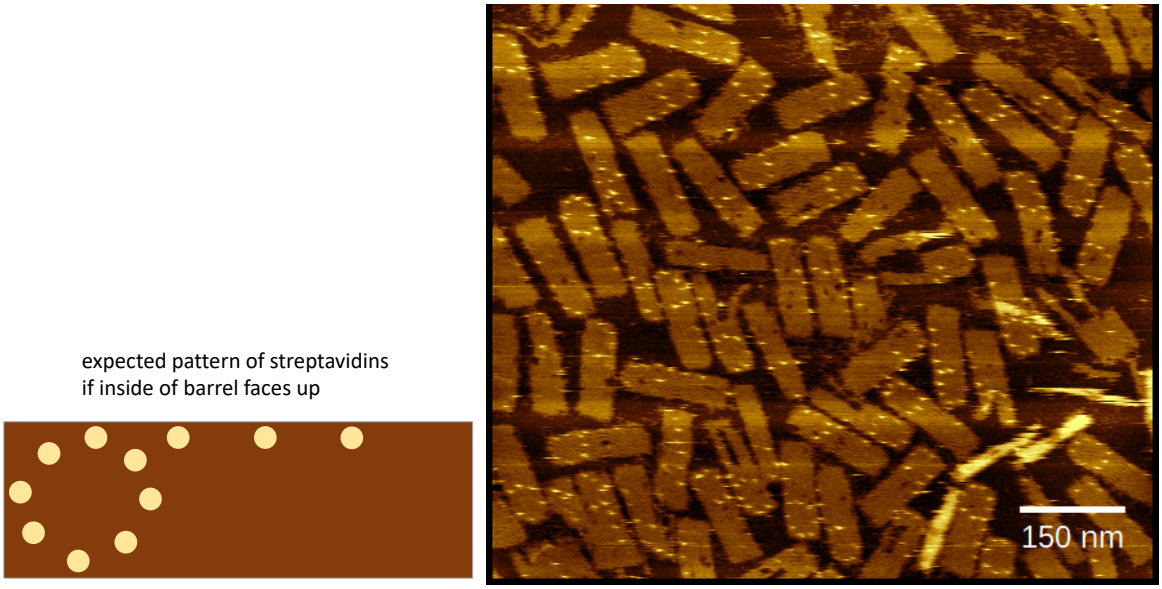


Figure S14: DNA origami seeds, 16-helix barrels, unzipped, with streptavidin-labeled biotins, spelling the letter σ (correctly oriented when viewing the side of the origami on the *inside* of the barrel). This is the expected orientation if the origami remains barrel-shaped after being unzipped. (The thinner, lighter-colored (i.e. larger height by AFM) objects are origami barrels that did not unzip and unroll.)

1 μM per staple in the mix. These stock staple mixes were prepared in advance of experiments and stored in the fridge.

S3.3 Input-adapter strands design

The *input-adapter* strands bind both to the DNA origami scaffold strand and to the first half-layer of tiles. Since they are the mechanism by which the DNA origami seed nucleates growth, and since our experiments showed a large fraction of seeds[33] had no significant growth of tiles, we considered and tested a number of different designs for input-adapter strands. Unfortunately we found no significant variation in seeded growth rates among different input-adapter designs.

The two main designs tested are shown in Figure S16: input-adapter crossovers inside the origami (left), or outside of the origami (right).

We predicted three potential problems with the "crossover inside the origami" design, explained in Figure S17. Therefore we did most experiments using the "crossover outside the origami" design shown on the right side of Figure S16. Unfortunately, however, *both* designs appeared to have an energetic barrier to seeded growth, showing only about half of seeds with significant growth of tiles.

Both designs share one feature in common, which is that the input adapter is bound to scaffold by 32 base pairs, significantly more than the 20 or 22 base pairs used when tiles attach to the end of the tube. This choice was intentional, appearing important to encourage seeded growth (although the necessity of this design feature was not tested) for the following reason. Tiles grow at a temperature where binding by ≈ 20 bp is slightly favorable (i.e., the on-rate is slightly larger than the off-rate of tiles attached by two binding domains). But at this temperature, we want each seed complex to be stable, so input-adapter strands should have many more base pairs holding them to the scaffold so that input-adapter strands are unlikely to detach from the seed. In our designs, each input-adapter is bound by two domains of length 16 bp each, so 32 bp total.

As with staples, there is no explicit "sequence design" for input-adapter strands, once the scaffold and tile sequences are designed. Input-adapter domains are simply set to be the complement of the domain they are

[33]Seeding fraction was not formally quantified on earlier experiments, but it was similar to the complete 6-bit IBC tile set, for which the percentage of well-seeded seeds 61.1 %.

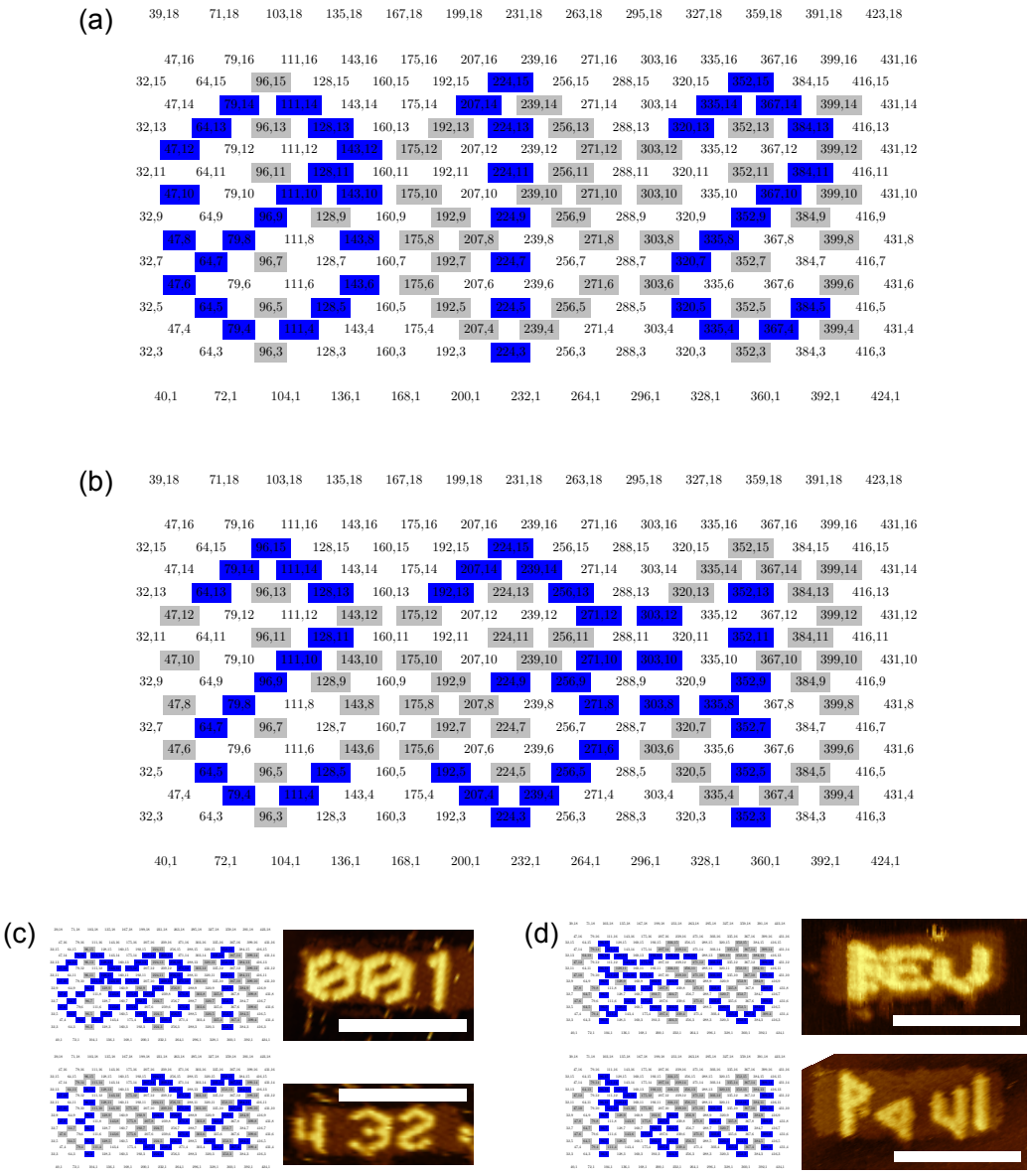


Figure S15: Origami barcode design, highlighting barcodes 012 (a) and 234 (b). The point of view is from the inside of the barrel, i.e., as if the barrel is unzipped, laying flat on a surface with the inside facing up and the point of view looking down from above. Input-adapter strands are located on the right hand side of each origami. (x, y) coordinates for all 208 staple $5'$ ends are shown as cadnano [72] coordinates. For each of the positions shaded in grey or blue there is a staple with a $5'$ biotin modification, and another staple without a biotin modification. Each unshaded position has merely a single staple without a biotin modification. To design an origami barcode, the user specifies a set of staple positions that should have biotins, i.e. the blue positions in the above two examples. (c) Two example origamis (samples were annealed with scaffold and staples, as well as input-adapter strands and tiles). (d) Two example origamis (samples were annealed with scaffold and staples, and no other strands). Scale bars: 100 nm.

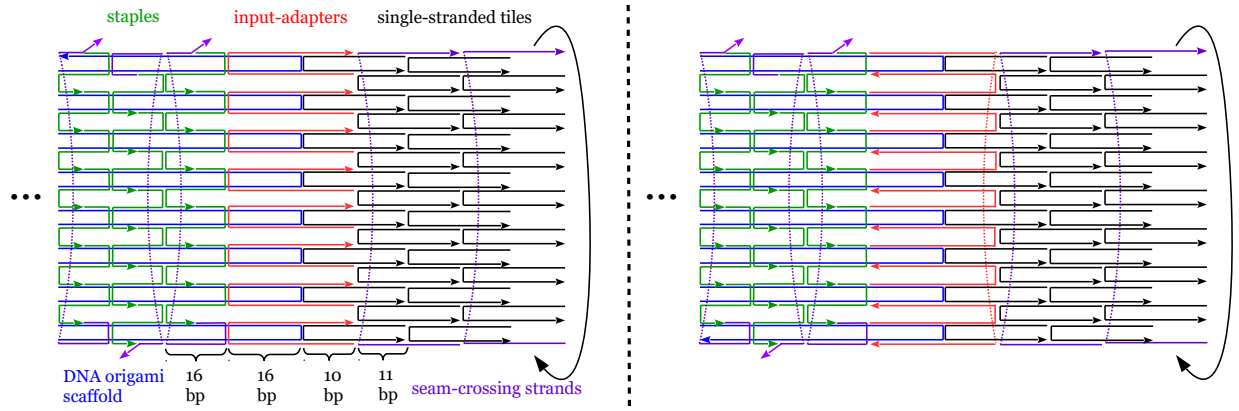


Figure S16: Two possible designs for input-adapter strands. Most experiments, including all 6-bit IBC experiments, used the design on the right. **left:** Crossovers are inside the DNA origami seed, and each tile binds both of its left-side binding domains to the same input-adapter strand. **right:** Crossovers are outside the DNA origami seed, and each tile binds its two left-side binding domains to two different input-adapter strands. This design is intended to prevent the three potential problems hypothesized to occur with the left design, described in Figure S17.

intended to bind, in either the scaffold or first "half-layer" of tiles. Our tile sequence design software did not explicitly check for problems in the input-adapter strands. However, we checked each input-adapter sequence and found that none had significant secondary structure. We believed that the tile sequence designer would ensure the tile-binding portion of the input-adapter strands was well-behaved, and that the scaffold-binding region would be more resilient because of its greater length.

It is possible, of course, that unintended secondary structure within or between input-adapter strands is responsible for seeded growth problems, and that incorporating input-adapter checks into the tile sequence design could have increased the seeded growth rate.

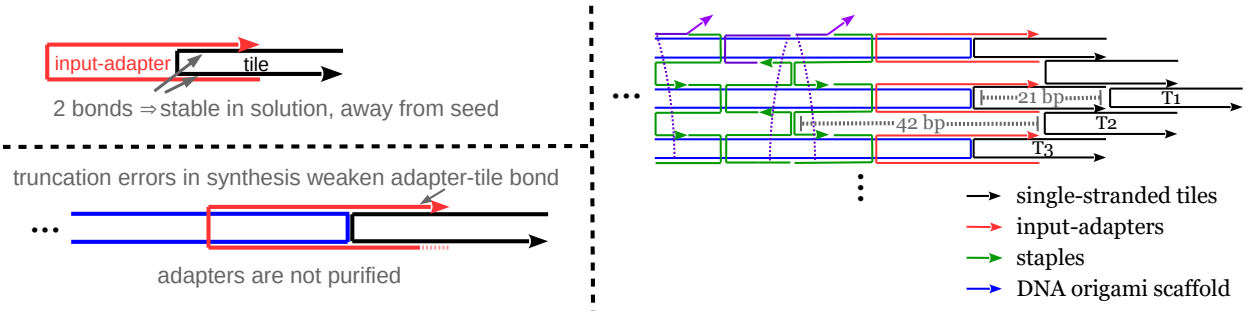


Figure S17: Three potential problems with input-adapter strands with crossovers inside the origami. **top left:** Since the optimal growth temperature favors tile binding by two bonds, a input-adapter/tile complex in solution will be relatively stable, making it harder for that tile to bind to an adapter on a seed. **bottom left:** Since input-adapter strands are not purified, and IDT reports that the most common DNA synthesis error is truncation at the 5′ end, this could weaken the tile attachment. **right:** Most tile attachments to a growing tube resemble the attachment of tile T1 above: the attachment extends two adjacent DNA helices held together by a crossover 21 bp to the left. Attachment of tiles in the first half-layer, such as T3, have an adjacent scaffold crossover holding the two helices together. In contrast, tiles in the second half-layer, such as T2, join two helices held together by a crossover 42 bp to the left. Prior to attachment, electrostatic repulsion could hold these helices further apart than in the case of tiles T1 and T3, weakening the binding strength of T2. By putting input-adapter crossovers on the outside of the scaffold as the right side of Figure S16, each tile in both of the first half-layers bind to DNA helices being held together by an adjacent crossover.

S3.3.1 Input bits encoded by input-adapter strands are not readable by AFM (no biotins)

Finally, the two domains on the input-adapter strands that bind to tiles, and thus act like strand output domains, represent input bits (see Figure S19(a) for the definition of input and output domains). However, we chose not to biotinylate the adapter strands whose output represents 1. This is the reason that the input bits are not directly "visible" in any AFM images, but only the first layer of growth on the input. Note that domains that have biotins in the SST tiles were designed to have energetically stronger binding sequences, so as to compensate for the energetic penalty we observed when a biotin is in place, and this has the consequence that when those same domains appear in input-adapter strands without biotins, those domains will be energetically stronger for binding. This may facilitate growth from the seed.

S4 System design abstraction level: DNA sequence design

The goal of DNA sequence design is to obtain a set of molecules that implement the intended geometry and interactions the domain-level SST model described in Section S3.

S4.1 Random sequences are not sufficient for algorithmic self-assembly

There has been much success at using random sequences, or lightly designed sequences (e.g. disallowing repeated subsequences longer than some length) for tile based self-assembly with SST motifs [15, 80, 17]. For these kinds of non-algorithmic systems there are reasonable hypothesis as to why random sequences might be beneficial, such as that having a mixture of weaker and stronger binding domains enables the very strongest of domains to provide locations for nucleation of a few nuclei during a slow anneal [32].

However, Figure S18(a) gives strong evidence that (almost) random sequences will result in intolerably high tile-attachment error rates for algorithmic self-assembly. Specifically the figure shows that, with almost random sequences[34], many of the possible incorrect attachments will be more favourable than correct attachments. One might wonder if we could dispense with sequence design and exploit proofreading to prevent such errors; but we can not: indeed, proofreading can only benefit us in a regime where the rate of erroneous attachments is lower than the rate of correct attachments. And in Figure S18(a) we see that for (almost) random sequences the implied attachment rates for erroneous attachments will in many cases be higher than those for correct attachments. The upshot is that we require significant sequence design, and, although we have not yet explained how it is achieved, Figure S18(b) gives a spoiler to tell us how our designed sequences look when examined for the same criteria.

S4.2 Energy-based DNA sequence design

In this section we describe the sequence design energy models, and energy thresholds within those models, with the goal of designing DNA sequences that are both capable of desired interactions and not unduly prone to undesired interactions. In the main text it is noted that the three main principles employed to inhibit growth errors are: (i) ensuring that desired interactions are isoenergetic, (ii) minimizing erroneous binding through minimizing mismatch binding energies, and (iii) employing proofreading tile sets in the logical design. Also, because SSTs are floppy, there are other undesired interactions, including (iv) minimizing off-lattice interactions (unintended binding of tiles to themselves or each other) and (v) minimizing near-lattice interactions (unintended interactions of strands at the lattice growth frontier).[35]

An important point to note is that a variety of different models are used for different criteria; one of the main reasons for this was to balance concerns about computational efficiency (some models are faster to evaluate than others) against concerns about how well the resulting strands would perform in self-assembly experiments (some models are perhaps more predictive than others). It should be noted that as part of the design process, designed sets of sequences were subsequently analysed using a suite of models and criteria that was broader than the design models and criteria, and in the end, the sequences that were chosen performed satisfactorily on that entire analysis suite (Section S4.3).

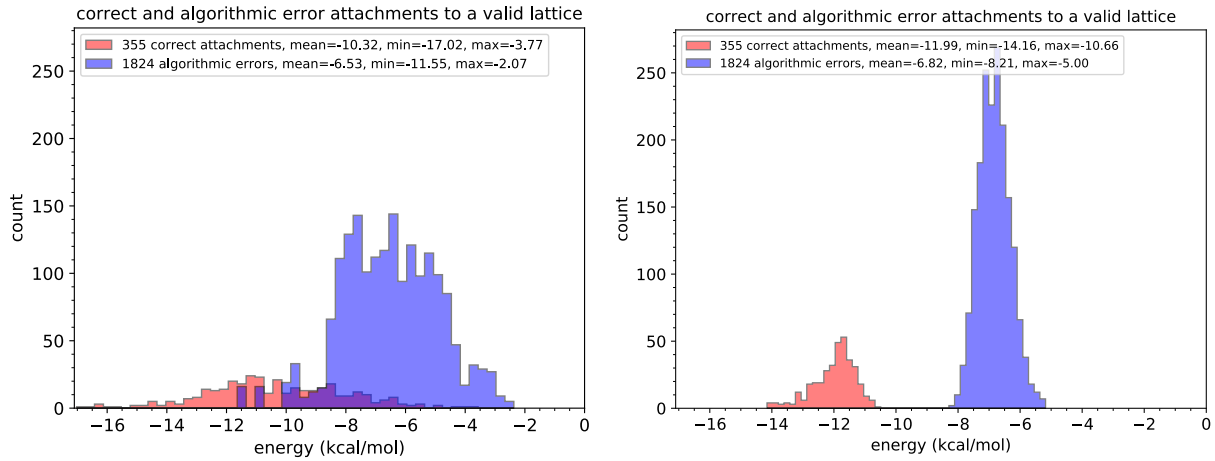
All energy values and thresholds are in units of kcal/mol, and energetics calculations were performed at a temperature of 53.0 °C. NUPACK `pfunc` energies were computed using the parameter sets `dna1998.dH` and `dna1998.dG` (that ship with NUPACK) and invoked via: `pfunc -T 53 -multi -material dna`. RNAduplex energies were computed from the parameter set `dna_matthews1999.par` (that ships with ViennaRNA), and invoked via: `RNAduplex -P dna_matthews1999.par -T 53 -noGU`.

S4.2.1 DNA sequence design criteria

Sequence design criteria were as follows:

[34] These random sequences were not entirely random: they used a 3-letter code (each strand used either alphabets A,T,C or A,T,G — a common rule of thumb to reduce internal secondary structure within strands); and the subsequences GGGG and CCCC were not permitted (a common rule to thumb to prevent formation of G-quadruplex structures and DNA synthesis difficulties).

[35] In fact interactions that fall under (iv) and (v) could be categorised under (i) and/or (ii), but it is useful for clarity of exposition to treat these kinds of interactions separately. Also, note that (iii) is described in Section S2.3.



(a) Random DNA sequences.

(b) Designed DNA sequences.

Figure S18: Evidence that using random, or almost random, DNA sequences leads to high tile attachment error rates during self-assembly. (a) Random sequences for our 355 tiles, but with a 3-letter code (each strand using either the alphabet A,T,C or the alphabet A,T,G) and where runs of CCCC and GGGG were forbidden within a domain. The red histogram shows binding energies for a tile correctly binding to a valid lattice, by its two input domains (see Figure S19(a) for the definition of input domains). The blue histogram shows a tile erroneously binding to a correct lattice where one of the tile's input domains correctly matches and the other mismatches (called a tile attachment error). Binding energies were calculated using the function `lattice_binding_spacer()` described in Section S4.3.3. Since more negative energy implies more favourable binding, it can be seen in (a) that many of the erroneous tile attachments (blue) are stronger than many of the correct tile attachments (red). (b) The same analysis applied to our designed DNA sequence set of 355 tile-strands for comparison; all erroneous tile attachments (blue) are significantly weaker than all correct tile attachments (red). Plot (b) is explained in Section S4.3.3.

(i) **Ensure that desired interactions are isoenergetic:**

(a) Use domains that bind into perfect duplexes with roughly equal binding strength. Model: for computational efficiency, a simple nearest-neighbour sum of stack energies was used with stack energies taken from [81] (i.e. using ΔH and ΔS values from Table 1). Length 10 and 11 sequences were filtered to include only those sequences ("domains") that lie in the following binding energy ranges: -9.05 ± 0.15 kcal/mol per domain, and -10.15 ± 0.15 kcal/mol per internal-biotin-modified domain (the reason for a "strength boost" for biotin labelled domains is described in Section S5.5.1). In addition, each domain sequence (of length 10 or 11) was flanked by an A or T base; with the goal of having the stack energy between adjacent domain domains be in a tight energy range.

(b) Prevent secondary structure within a tile/strand. When a SST binds to a growing lattice, any secondary structure internal to the strand may need to be undone for binding to succeed, thus inducing energetic penalties. To prevent such energetic penalties, the sequence designer had a threshold of -1.65 kcal/mol for allowable strand secondary structure computed using NUPACK's partition function calculation (called "NUPACK `pfunc`" below). In addition, a "near" 3 letter code was used for strands: each strand used either sequences A,T,C or A,T,G (with up to a single letter exception per strand). Adhering to a 3 letter code facilitates finding sequences that have little intramolecular secondary structure.

(c) Upper bound the binding energy for correct binding to a lattice. Here the goal is that no correct binding event should be weaker than any incorrect binding event. Above, Point (ia) is designed to make bindings be isoenergetic according to a simple, and computationally efficient, method. A more computationally expensive method involves computing the energy for a SST binding correctly with its two input domains (`dom1` and `dom2`, see Figure S19(a) and (b))), using temperature `temp` = 53.0 °C, where "`s1+s2`" denotes the concatenation of strings s1 and s2, to a growing lattice:

```
def eval_lattice_binding_energy(dom1,dom2,temp):
```

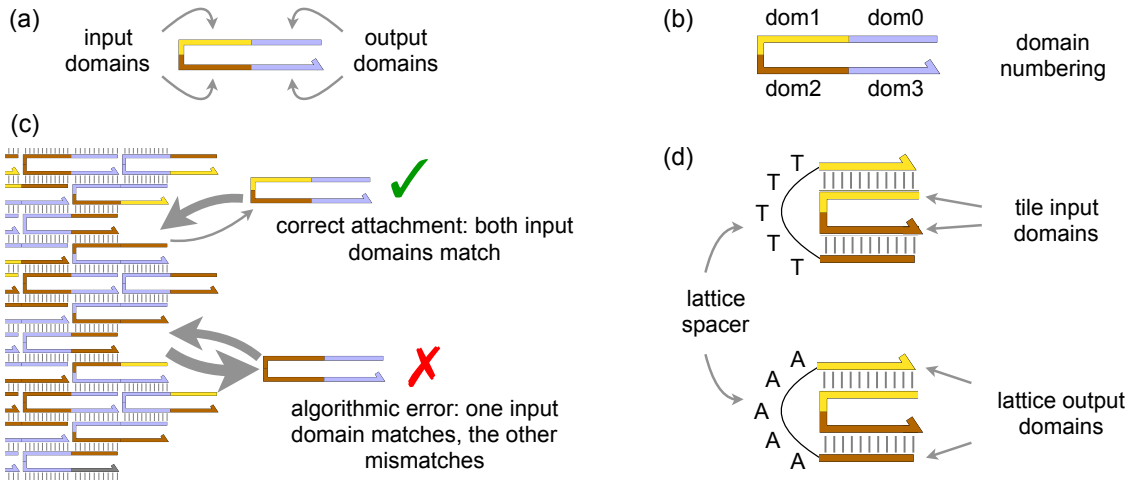


Figure S19: (a) A SST motif with its two input domains and two output domains. Each domain has an associated length 10 base, or length 11 base, DNA domain sequence. (b) The four domains of an SST are numbered 0 to 3 and named as shown. (c) A correctly grown lattice showing two tile binding events about to happen, a correct attachment and a tile attachment error. The correct attachment joins by two input domains that match two (output) domains on the lattice. The tile attachment error attaches by one matching domain and one mismatching domain. (d) Illustration of the intuition behind simulating a lattice tile-binding site with a sequence TTTTT or AAAAA between two domains, in order to simplify calculations of lattice-binding energies.

```
return max(binding(dom1+dom2,wc(dom2)+'TTTTT'+wc(dom1),temp),
           binding(dom1+dom2,wc(dom2)+'AAAAA'+wc(dom1),temp))
```

The function `binding` attempts to approximate the geometry of a lattice by a simple and computationally efficient string concatenation method, avoiding the need to consider complicated lattice geometries, possibly composed of pseudoknotted DNA structures, for which we can not efficiently compute binding energies – see Figure S19. The function computes the (partition function) free energy of association between two strands, via the intuition

$$\Delta\Delta G_{\text{reaction}} = \Delta G(\text{products}) - \Delta G(\text{reactants})$$

and using NUPACK's `pfunc` executable (version 3.0.4), as defined below. Note that we do two calculations with two sequences 'TTTTT' and 'AAAAA' as "spacers" between domains that should be colocated but not immediately adjacent, and take the least favourable (max) of the two results.

```
def binding(seq1,seq2,temp):
  return pfunc((seq1,seq2),temp) - (pfunc(seq1,temp) + pfunc(seq2,temp))
```

Using this method we insist that all tiles should bind with energy no more than (i.e. "no weaker than") -12.3 kcal/mol. I.e. for all tiles we have `eval_lattice_binding_energy` < -12.3.

(ii) **Minimize erroneous binding through minimizing mismatch binding energies:** A tile attachment error is where a tile binds to a growing lattice by its two input domains, but where one domain is correctly binding ('matching') and the other is incorrectly binding ('mismatching'); an example is shown in Figure S19(c). We sought to minimize tile attachment errors of two kinds.

- weaker than -1.6 kcal/mol: Threshold for allowed energy of interaction between algorithmically conflicting domains that might be co-located during a strength-1 binding event (one input domain matches and one mismatches) that is a "first" error to an otherwise correct lattice.
- weaker than -2.6 kcal/mol: Threshold for allowed energy of interaction between algorithmically conflicting domains that might be co-located during an arbitrary strength-1 binding event (on a correct lattice or a lattice with errors from other tiles binding with strength 1).

```
def eval_colocated_domain_pair(domain1, domain2, temp):
  return min(pfunc(domain1+domain2, temp),
         max(pfunc(domain1+'TTTT'+domain2, temp),
             pfunc(domain1+'AAAA'+domain2, temp)))
```

(iii) **Minimize erroneous binding through the use of proofreading tile sets:** See Section S2.3.

(iv) **Minimize off-lattice interactions, i.e. minimize binding of tiles to each other as dimers:** Unintended binding of tiles away from the lattice (e.g. as dimers) have dual unintended effects of reducing the concentration of free monomer tiles and of having unintended secondary structure between tiles in a dimer that needs to be undone before one of the tiles can bind to a lattice. Both of these negatively effect Point (i) above. Recall that tiles were designed so that any pair of tiles binds to each other by zero, or at most one, domain, giving two cases to consider:

- weaker than -5.4 kcal/mol: Threshold for tile pair secondary structure energy, for tile pairs that do not have a complementary domain. Calculated using ViennaRNA's `RNAduplex` (version 2.1.9) executable which computes the partition function energy for inter-molecular base pairs between a pair of strands, and ignores intra-molecular base pairs within the strands.
- weaker than -7.4 kcal/mol: Threshold for tile pair secondary structure energy, for tile pairs that have a complementary domain. Calculated using ViennaRNA's RNAduplex.

(v) **Minimizing near-lattice interactions (unintended interactions of strands at the lattice growth frontier):**

- weaker than -1.4 kcal/mol: Threshold for two energies: (a) Energy of a *lattice* secondary structure (from two co-located output domains) that must be broken up to allow the *correct* binding of a tile to those domains. (b) Energy by which the two *output domains* of each tile bind to each other (must be broken up for any tile binding to either one of these outputs). Calculated using the function `eval_colocated_domain_pair(domain1, domain2,temperature)` described above.

S4.2.2 Sequence design algorithm

The above criteria specify a multi-objective optimisation problem: the goal being to find DNA sequences, or more specifically a set of length 10 and 11 domain sequences assigned to the 355 tiles, that satisfy all criteria. This multi-objective sequence design problem was tackled using a stochastic local search algorithm (described below). Part of the process involved finding the tightest set of criteria above such that the algorithm halted in a reasonable amount of time, and such that the produced DNA sequences looked good according to the analysis carried out below in Section S4.3. The final run of the program took about a week (with some manual guidance) to produce 355 length-42 DNA SST sequences that met our stringent design criteria. The pseudocode in Figure S20 outlines the algorithm.

S4.3 Analysis of designed DNA sequences

As part of the design process, designed sequence sets were analysed for a variety of properties. These tests were (a) more extensive (but more computationally expensive) than tests carried out by the sequence designer itself and (b) provided feedback on how to modify and improve the designer to achieve a sequence set that best achieved intended goals. The results of this analysis for our 355 designed sequences are summarised here.

S4.3.1 Logical correctness check

A logical design check was performed where a Python program checked that, based on sequence information only, designed sequences do indeed form into the intended proofreading blocks, and that those blocks in turn bind to each other in ways that form intended nanotube structures. This can be considered as a kind of "program correctness" check for the class of 6-bit molecular self-assembly IBC programs, at the DNA sequence level.

```
list_of_domains = generate_all_domains()       // DNA sequences of length 10 & 11, in energy ranges (ia) above
tiles = assign_domain_sequences_to_tiles(list_of_domains)
list_of_domains = list_of_domains.remove(tiles.domains())          // remove domains already assigned to tiles
list_of_bad_domains := find_bad_domains()
  // The function find_bad_domains() checks the set of tiles (strands) for violations of the above sequence
  // design criteria, and returns a list containing one copy of each domain for every such violation, and
  // for a violation involving entire strand(s) all four domains on the strands are added to this list.
  // Domains that cause violations on multiple different tests are added multiple times, giving a weighting
  // on their badness.

num_bad_domains = length(list_of_bad_domains)
while num_bad_domains > 0:
  domain = choose_uniformly_at_random(list_of_bad_domains)           // choose a domain to 'fix'
  domain_current_sequence = domain.sequence()                        // get DNA sequence of domain
  domain_new_sequence = choose_uniformly_at_random(list_of_domains)  // choose a sequence, but do not yet
                                                                     // remove its domain from list_of_domains
  tiles.assign_domain_sequence_to_tiles(domain, domain_new_sequence) // attempt a change of domain sequence
  list_of_bad_domains := find_bad_domains()
  if num_bad_domains >= length(list_of_bad_domains):                 // keep the change
    num_bad_domains = length(list_of_bad_domains)
    remove domain_new_sequence from, and add domain_current_sequence back to, list_of_domains
  else:
    domain.assign(domain_current_sequence)                           // reverse the change
```

Figure S20: Pseudocode for DNA sequence designer.

S4.3.2 Individual domain-level and strand-level analysis of designed DNA sequences

Figure S21 (left) shows that each designed domain sequence (of length 10 or 11 bases) had relatively little secondary structure (roughly no more than a typical DNA stack). Figure S21 (right) shows energies of secondary structure for each tile/strand.

Point (ia) of our sequence design criteria strives to have each domain bind isoenergetically in a lattice. However, for computational efficiency in the sequence designer, as discussed in Point (ia), we asses this by a simple method that sums nearest-neighbour energies in pairs of perfect domain-duplexes. Figure S22 gives an analysis of secondary structure energies between pairs of domains (i.e. binding of domain pairs) computed using the function `binding()`, described in Section S4.2, that takes account of undoing any secondary structure that might exist within a domain. Ideally the red and blue curves would not overlap, but some overlap can be seen (max WC is more than min non-WC). Later figures using more careful, and we hypothesize more accurate, methods will show a clear separation between intended binding and unintended/erroneous/non-WC bindings of entire tiles to a lattice.

In the sequence designer, `RNAduplex` was used to check the strength of strand pair interactions, see Section S4.2, Point (iv). `RNAduplex` is fast, but looks only at inter-strand secondary structure, and we wanted to compare with a method that also takes into account both inter- and intra-strand secondary structure. In Figure S23 gives one such comparison: it shows energies for strand-pair interactions computed via `binding()` (uses NUPACK `pfunc` as described in Section S4.2) and `RNAduplex`. Since we have these two models available to us, and not knowing which maps closer to reality, we used plots like those in Figure S23 to asses our designed sequences with the aim of choosing a final set of sequences that performed well on both.

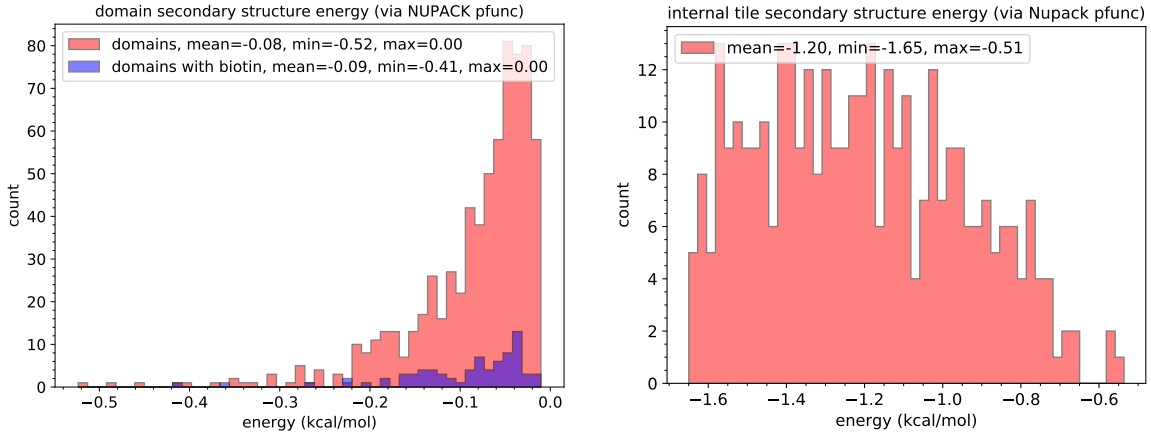


Figure S21: Histograms of secondary structure energies of individual domains (left), and individual tiles/strands (right), from our 355 designed DNA sequences measured using NUPACK `pfunc`, evaluated at 53 °C. Left: Domains with biotins and shown in red and domains without biotins are shown in blue. Domains with biotins (i.e. internally modified 'T' bases with a biotin attached) were treated as standard 'T' bases in the calculation. Right: As noted in Section S4.2(ib), a secondary structure energy that is greater than a threshold of −1.65 kcal/mol for structure within each of the 355 DNA strands was used in the design of the tile set. This threshold value is comparable to a single GC or CG stack at our design/analysis temperature (53 °C) [81], which implies a relatively small amount of secondary structure for a length 42 DNA strand.

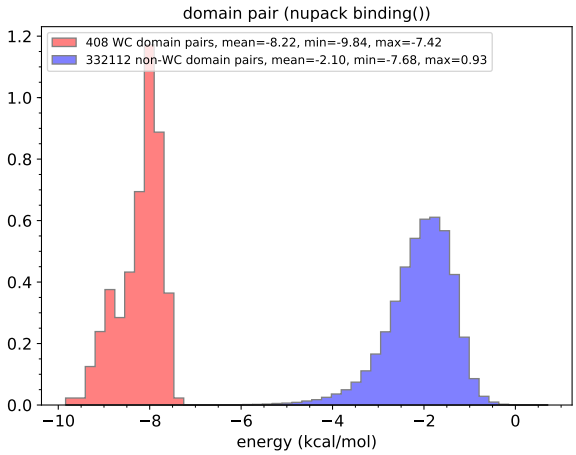


Figure S22: Histograms of energies for interactions between pairs of domains computed using the function `binding(dom1, dom2, 53.0)` defined in Section S4.2.1, Point (i), and where `dom1`, `dom2` are arbitrary non-equal domains from the complete 6-bit sequence set. Red histogram: binding energies for complementary Watson-Crick domain pairs (WC). Blue histogram: non-complementary domain pairs (non-WC). The curves are normalised so that both curves have equal area under the curve; without normalisation the red curve would not be visible to the eye due to the large count of non-WC domain bindings.

S4.3.3 Energies for tiles binding to a growing lattice

Roughly speaking, we would like binding for all correct tile attachments to a growing tile-lattice to be strictly more favorable than for incorrect attachments. For reasons of computational efficiency, the design code and the analysis code treat these attachment energies a little differently.

We analyse three kinds of tile attachment errors, based on whether, before the erroneous attachment the

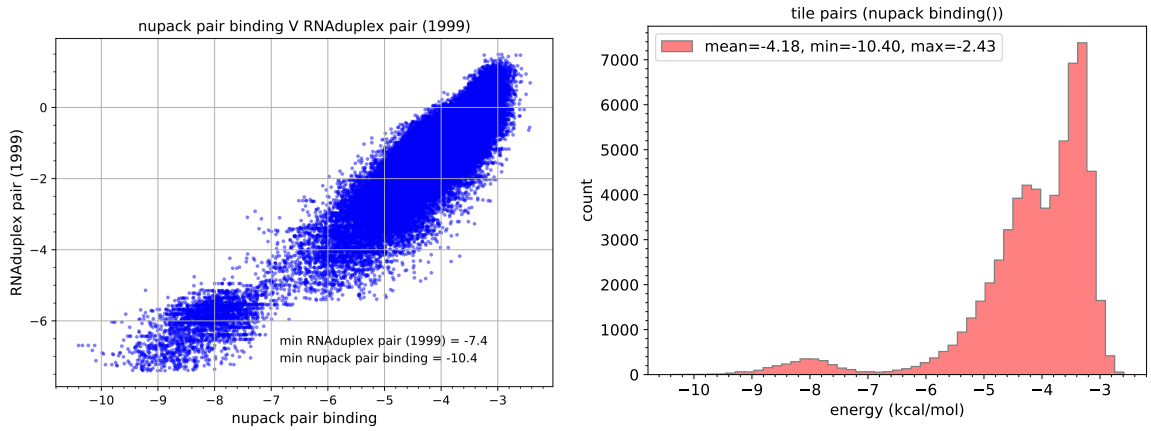


Figure S23: Left: Tile pair energies via `binding(tile1,tile2,53.0)` (horizontal axis; defined in Section S4.2.1, Point (i)) versus ViennaRNA `RNAduplex(tile1,tile2,53.0)` (vertical axis), for all `tile1` and `tile2` that are non-equal tile-strands from the complete 6-bit sequence set of 355 tile-strands. In the scatter plot data, one can observe the two thresholds for RNAduplex used in the design code: energies were greater than a threshold of -7.4 kcal/mol for tiles pairs that share a common glue, and -5.4 kcal/mol for those that do not. Right: Histogram of tile pair energies via `binding(tile1,tile2,53.0)`; not normalized.

lattice has no or already has some errors. Energies are computed using the function `lattice_binding_spacer` defined below using Python syntax.

1. Correct attachment, or tile attachment error attachment, to an otherwise valid lattice. Here we assume we have a correctly grown lattice, and a tile-strand binds to the lattice with its two input domains where one input domain matches and the other mismatches. An example is shown in Figure S19(c). The analysis is shown in Figure S24 (which is identical to Figure S18(b)).

2. In this point, and in Point 3 below, we analyse the energy of attachment of a second[36] error given that a first error has already occurred. This kind of analysis makes particular sense in the light of proofreading; one of the main properties of proofreading is that an incorrect attachment forces a second incorrect attachment. Having such second attachment errors be as energetically unfavourable as is possible should help to lower the overall error rate.

   In this point in our analysis, as first errors we include a reasonably wide class of errors that includes both tile attachment errors (match on one input domain, mismatch on the other), as well as some non-tile attachment errors (match on neither input domain). In aTAM terms, this includes all strength-1 binding errors (i.e. tile attachment errors and facet errors [24]) and as well as some strength-0 binding errors (binding with 0 matching sides, due to unintended interactions). For second errors, we assume we have a lattice that is locally perfect except that one of the tiles $t_{\text{incorrect}}$ is incorrectly bound via a first error; and $t_{\text{incorrect}}$ has the correct helix number and the correct position within its proofreading block (but is from the wrong proofreading block). Figure S25(b–d) gives examples.

   The blue histogram in Figure S25 gives energies of attachment of the first and second errors. For comparison, energies of correct bindings are shown in the red histogram in Figure S25.

3. The goal here is to investigate the energy of attachment of tiles to a lattice that already has arbitrary errors — if we are unlucky enough to see erroneous attachments we'd like to know whether they will propagate. Consider a lattice that has formed from the addition of arbitrary tiles (but respects the usual SST lattice geometry, i.e. growth of a lattice from a seed where tiles bind by their two input domains, but possibly mismatching). We analyse the binding of a tile to such a lattice where one of the binding-tile's input domains matches a domain on the lattice and the other domain is co-located

[36] We thank Constantine Evans for a fruitful discussion and thoughts on the idea of analysing and suppressing "second" errors.

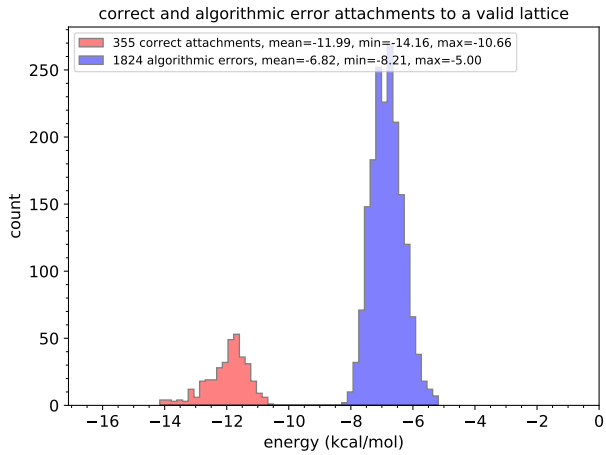


Figure S24: Energy of a tile binding to a perfect lattice either correctly (red) or as a tile attachment error binding event (blue, i.e. one input domain matches the other mismatches). Energy computed using the function `lattice_binding_spacer()` described below; with parameters defined as follows. The parameter `strand` iterates over all 355 tile-strands. The pair of parameters `lattice_bottom_domain` and `lattice_top_domain` iterate over all pairs of domains that exactly bind to the two input domains of `strand` (giving the red histogram), or with one binding correctly and the other mismatching so that it simulates a tile attachment error as shown in Figure S19(c) (giving the blue histogram). The histograms are not normalised in this figure.

with a mismatching domain. The resulting energies of attachment are shown in Figure S26, evaluated using two different metrics.

The following Python code defines the function `lattice_binding_spacer()` used throughout this section. The function gives an evaluation of the energy of binding of a tile to a lattice by two domains according to an implied model that takes into account lattice-tile secondary structure upon binding, as well as any secondary structure within or between the two lattice domains that the tile will bind to, and within the tile before it binds to those two lattice domains. Figure S19 explains the idea of how the code approximates a lattice location for a tile to bind.

```
def lattice_binding_spacer(strand, lattice_bottom_domain, lattice_top_domain, temperature):
    '''The parameter 'strand' is string with 4 single-whitespace delimited domains. A lattice location for tile
    attachment is treated as a single strand of the form lattice_bottom_domain+lattice_spacer+lattice_top_domain,
    where lattice_spacer is either 'TTTTT' or 'AAAAA'.'''
    tile1_domains = strand.split(' ')
    dG_sticky_ends_bound = max(binding(tile1_domains[1]+tile1_domains[2],
                                       lattice_bottom_domain + 'TTTTT' + lattice_top_domain, temperature),
                               binding(tile1_domains[1]+tile1_domains[2],
                                       lattice_bottom_domain + 'AAAAA' + lattice_top_domain, temperature))
    dG_tube_ends_after   = max(pfunc(tile1_domains[0] + 'TTTTT' + tile1_domains[3], temperature),
                               pfunc(tile1_domains[0] + 'AAAAA' + tile1_domains[3], temperature))
    dG_strand_before     = pfunc(strand, temperature)
    dG_tube_ends_before  = max(pfunc(lattice_bottom_domain + 'TTTTT' + lattice_top_domain, temperature),
                               pfunc(lattice_bottom_domain + 'AAAAA' + lattice_top_domain, temperature))
    dG = dG_sticky_ends_bound + dG_tube_ends_after - dG_strand_before - dG_tube_ends_before
    return dG
```

S4.3.4 Final choice of designed DNA sequence set

A number of sets of DNA sequences were designed, under various sequence design criteria, and then analysed using the methods described here in Section S4.3. In addition to examining energy plots, for all of these lattice-binding tests the sequences of the worst ten or so offending binding events were printed to a screen and examined by eye.

A final set was chosen that seemed to give a pareto-optimal solution under our analysis criteria. That particular sequence set was found at the end of a week-long run of the sequence designer, during which time the search was paused and the criteria slightly adjusted by hand to get out of presumed local minima.

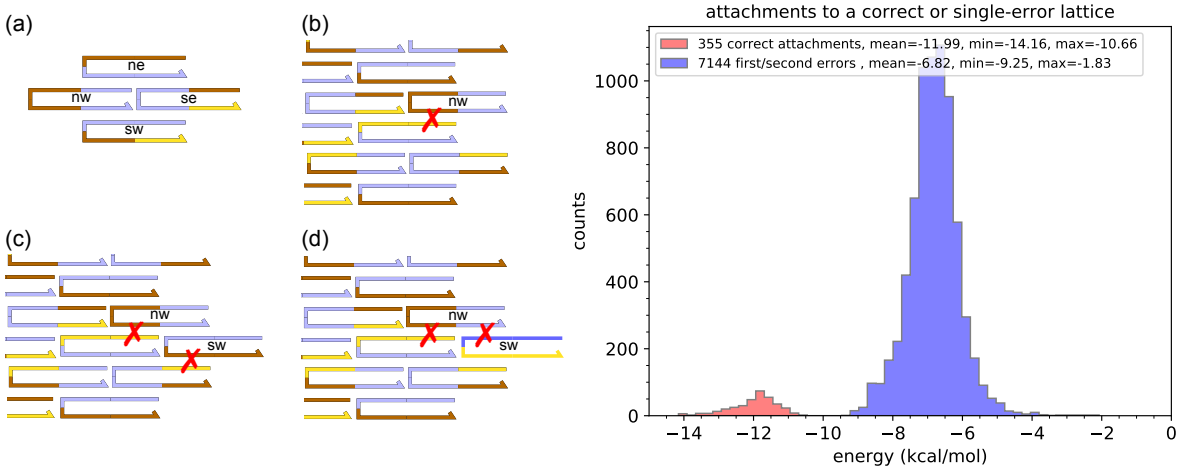


Figure S25: Energy of a tile binding to a lattice that has has zero or one errors. Left: (a) Naming convention (using cardinal directions) used for names of the four tiles in a proofreading block. (b) A tile has incorrectly bound by a tile attachment error (one mismatching domain); the tile has the correct position from within a proofreading block (nw in this case) and straddles the correct pair of helices. After the error in (b) has occurred, it can either fall away or, due to the proofreading property, force a second error. Two such example second errors are shown in (c) and (d). Right: The red histogram shows energies for correct binding (same as red histogram in Figure S24). The blue histogram shows energies of binding for tiles that are at the correct position within a block, and are on the correct helix, but are from the wrong proofreading block (and so mismatch on at least one domain), and are binding to a lattice that is correct or that already has a single such an error. Energies were computed using the function `lattice_binding_spacer()` specified in Section S4.3.3, Point 3. The histograms are not normalised in this figure.

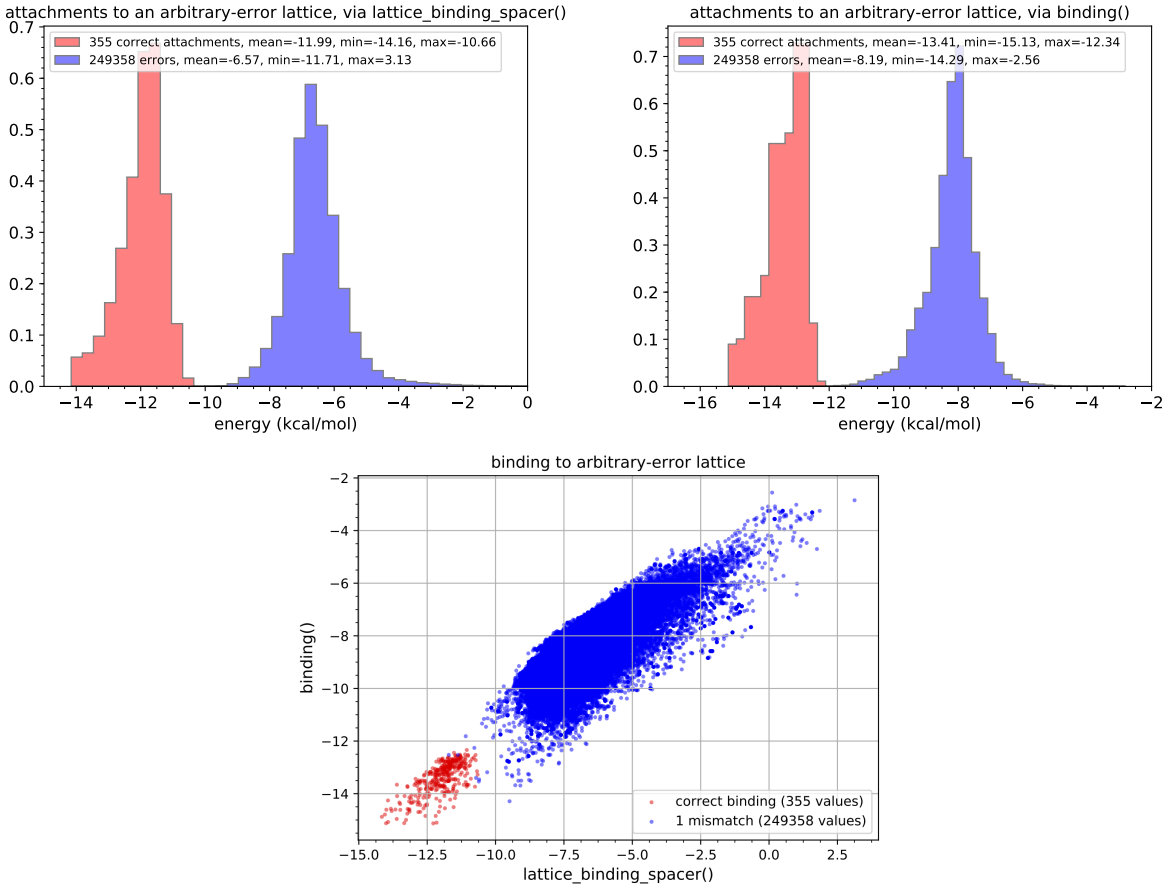


Figure S26: Overcoming adversarial errors: Energy of a tile binding to a lattice that may have arbitrary errors already present, i.e. we assume a preformed lattice that has a position available for a tile to bind, but that spot was created through arbitrary previous errors. We then compute binding energy for both perfect binding events (if there is a tile that matches, red histogram), or binding events that are tile attachment errors (attachments with one matching and one mismatching domain, blue histogram). Top-left: Energies were computed using the function `lattice_binding_spacer()`. Top-right: energies computed using the (simpler) function `binding()`. All histograms are normalised to have constant area under the curve. Bottom: The scatter plot compares the two: vertical axis shows analysis from top-left histograms, horizontal shows analysis from top-right histograms and shows that the strands perform better on the metric `lattice_binding_spacer()` which is more complicated, but presumably more accurate, than `binding()`.

S5 Justification for design decisions

S5.1 Seeded growth of SST nanotubes

Before settling on our implementation choice of SSTs grown on a cylindrical lattice, for algorithmic self-assembly we needed to know whether these tiles were capable of seeded growth. *Seeded growth* means self-assembly from a seed structure with no other off-seed assembly taking place in solution. This section explains our methodology and findings.

In general, seeded growth must take place below the melting temperature of the structure being grown. However, too low a growth temperature will result in assembly occurring away from the seed, an unintended assembly pathway that leads to undesired products. In previous work, *zig-zag* tile sets that self-assemble into flat ribbons were designed so that their self-assembly pathways implied a kinetic barrier to nucleation, shown both theoretically [23] and experimentally [7] using DNA double-crossover tiles. Zig-zag tiles are designed to grow in a zig-zag pattern from the seed; if growth away from seeds occurs then a full zig-zag pattern must complete in order to obtain an assembly of tiles where all tiles are bound by two domains/sides. The essence of the theoretical argument is that – much like standard models of nucleation in 2D crystals [24] – nucleation pathways that reach a point where continued favorable growth (by 2 binding domains at each step) is possible must first experience multiple unfavorable growth steps (by 1 binding domain each) and pass through a critical nucleus whose thermodynamic concentration limits the rate at which new supercritical assemblies are formed.

For macromolecular self-assembly of tubular structures, similar models have been used to explain nucleation barriers. For example, in the relevant regime, microtubule nucleation has been measured to depend on the seventh to twelfth power of monomer concentrations, consistent with a critical nucleus involving a comparable number of monomers [82, 83, 84], although the exact nucleation pathway remains controversial [85]. The strong dependence on concentration implies that nucleation can be dramatically reduced by lowering concentrations. In DNA nanotechnology, nanotubes self-assembled from double-crossover tiles have similarly been observed to exhibit a nucleation barrier sufficient to enable controlled nucleation using a DNA origami seed [86, 26], where a mechanism analogous to that proposed for the zig-zag tile set was assumed to be relevant. Data from previous work on periodic SST nanotubes [13] similarly indicated a kinetic barrier to nucleation in 4-helix and 6-helix nanotubes, and a critical nucleus was proposed; that data suggested the potential for seeded growth on the timescale of an hour or two, but is inconclusive for longer time-frames or whether the nucleation barrier increases for larger-circumference nanotubes. Indeed, as already noted, for algorithmic self-assembly we desired (a) seeded growth times on the order of a day and (b) for nanotubes with a number of helices significant larger than 6 (in order to implement a circuit with as many input bits as possible). There have been a number of studies that use computer simulation of coarse-grained self-assembly models to elucidate the kinetics of nucleation and growth of (uniquely-addressed) SST structures [33, 32, 87].

S5.1.1 SST nanotubes have a sufficiently large kinetic barrier to nucleation for seeded growth

We carried out a systematic survey of nanotubes of diameter 8, 10, 12, 14 and 16 helices. The examined nanotubes were periodic (i.e., not algorithmic) SST structures of the kind in [13], although here we study a larger number of helices and here we have one strand being 5′-labelled with a Cy3 or Alexa647 fluorophore for imaging via fluorescence microscopy. DNA sequences are given in Section S13 (Supplementary Information B). Our specific goal was to find a tile-type concentration high enough to allow self-assembly of well-formed tubes in a reasonable amount of time, but low enough to exhibit a sizable kinetic barrier to nucleation; enough to exhibit seeded growth conditions for roughly a day.

Data set I: Tile type concentration of 1 μM. We began with a tile-type concentration of 1 μM. The experimental protocol was as follows. For a h-helix nanotube with $h \in \{8, 10, 12, 14, 16\}$ (also called a circumference-h nanotube), a sample test tube (0.5 ml, Eppendorf LoBind) was prepared with each of the h tile types at a concentration of 1 μM in 50 μl of 1×TAE/12.5 mM Mg^{++}. For these long temperature-hold experiments, test tubes with low binding affinity for DNA were used. Samples were heated to 90 °C for 10 mins, then cooled to some target temperature T °C at a rate of 1 °C/min, and held at T °C for at approximately 24 hours. A droplet was removed and placed on a room-temperature, uncleaned, glass slide

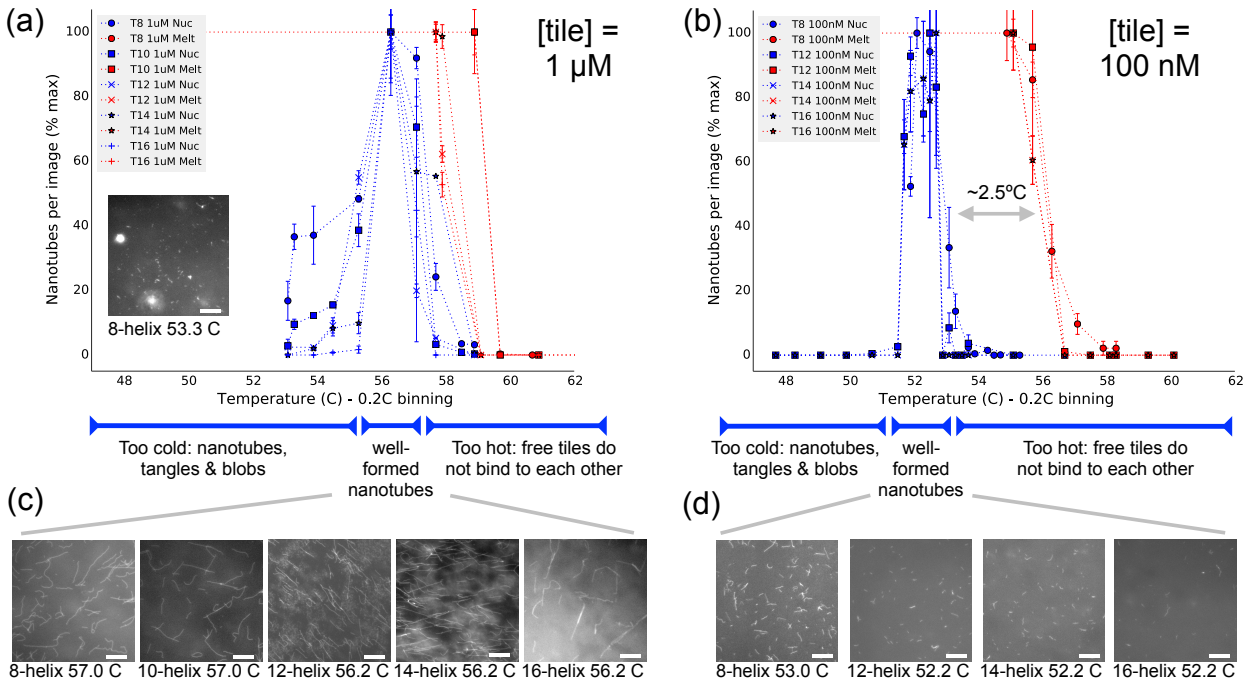


Figure S27: Experimental characterisation of nucleation and melting temperatures of periodic SST nanotubes over a 1-day time period. The protocol is described in the text (Section S5.1.1). Data points were binned into 0.2 °C intervals. Data shows (approximately) 24-hour temperature hold experiments for SSTs. (a) Tile type concentration of 1 μM. (b) Tile type concentration of 100 nM. (d) Representative fluorescence images of nanotubes formed in a temperature range where we consistently see well-formed nanotubes at 1 μM concentration per tile type, from the blue curve: 8-helix (T = 57.0 °C), 10-helix (T = 57.0 °C), 12-helix (T = 56.2 °C), 14-helix (T = 56.2 °C), 16-helix (T = 56.2 °C), (c) Representative fluorescence images for nanotubes that were formed in a temperature range where we consistently see well-formed nanotubes at 100 nM concentration per tile type, from the blue curve: 8-helix nanotubes (holding temperature of T = 53.0 °C), 12-helix nanotubes (T = 52.2°C), 14-helix nanotubes (T = 52.2°C), 16-helix nanotubes (T = 52.2 °C). (a, Inset): Representative fluorescence image for 8-helix (T = 53.3 °C) held at a temperature below the "well-formed" temperature range, on the blue curve — large blobs and tangles of nanotubes can be seen. Scale bars 10 μm.

and imaged using fluorescence microscopy. The number of nanotubes observed per 200 μm × 200 μm image were counted. That count was normalised to the % of the maximum count over all images for nanotubes with the same number of helices. The resulting percentage value was plotted as a blue datapoint Figure S27(a). After imaging, samples were stored at 4 °C. One might wonder if assembly could take place during or after deposition onto room-temperature glass; however, we observed that (i) the DNA rapidly stuck to the glass surface and (ii) for those samples that were held above the nanotube melting temperature prior to imaging, we observed only flat bright background, implying that no assembly of (intended nor unintended) structures was taking place during the imaging step of the protocol. Three distinct temperature regimes were observed: a high temperature range with no observable growth (i.e. images with a bright flat background), a mid range exhibiting growth of long-well formed nanotubes (Figure S27(c)), and a lower temperature range exhibiting assembly of nanotubes mixed with other non-specific aggregates (Figure S27(a), inset). Below, we use the term the "nucleation temperature" to refer to the approximate transition temperature between the mid and high range (which lies in the range 56.5–59 °C in Figure S27(a); depending on nanotube circumference).

The previous set of experiments gave us nanotube nucleation temperatures for a 1-day temperature hold; to find the corresponding nanotube *melting* temperature for a 1-day temperature hold the following procedure was followed. Samples (from the previous step) that were found to contain a dense population of well-formed nanotubes were heated from room-temperature to a target temperature of T °C at a rate of 1 °C/min, and held at T °C for at approximately 24 hours. A droplet was deposited on a glass slide, imaged by fluorescence microscopy, nanotubes counted and converted to % of max for each value of h and the results were plotted

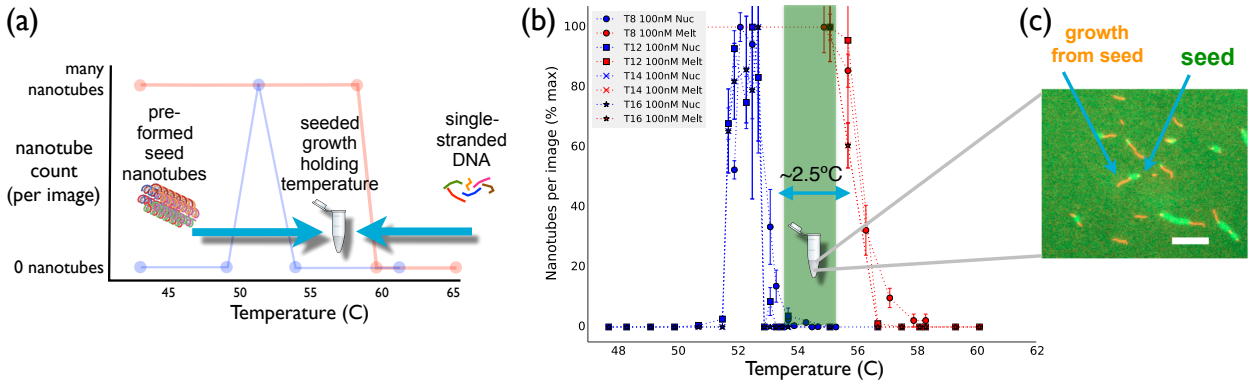


Figure S28: Exploiting a kinetic barrier to nucleation for seeded growth (a) Schematic: SSTs are cooled from 90 °C down to a temperature T, that lies between the nucleation and melting temperature (determined from previous 24-hour temperature-hold experiments). Pre-formed nanotubes are added at temperature T. T is too hot for spontaneous nucleation to occur (away from the seeds) and too cold for nanotubes to melt, also since T is below the '24-hour melting temperature', the seeds grow. (b) Data from Figure S27(b), illustrating a temperature range of approximately 2.5 °C where seeded growth may occur according to the schematic in (a). (c) Example experiment demonstrating seeded growth. Seeds are pre-formed 8-helix SST nanotubes with one tile type having a 5′ cy3 modification for visualisation via fluoresce microscopy. At 100 nM concentration of each tile type, and in the presence of seeds, nanotubes grow. Without seeds, nanotubes do no form (data not shown).

as red datapoints in Figure S27(b).

Two distinct temperature regimes were observed; a high temperature range where no structures were visible and thus we concluded that the nanotubes had melted, and a low temperature range were nanotubes were visible (and thus had not fully melted). Below, we use the term "melting temperature" to refer to the approximate transition temperature between these two ranges for a given nanotube circumference (which lies in the range 58–60 °C in Figure S27(a); depending on nanotube circumference).

For a given circumference, a larger gap between the nucleation temperature and melting temperature is consistent with a larger kinetic barrier to nucleation according to standard nucleation theory (see, for example, [23, 7, 24]). Although some circumferences showed a measurable gap between their nucleation and melting temperature at 1 μM, we deemed the gap too small for seeded growth (technique described below).

We hypothesised that a lower tile type concentration might result in a larger gap between nucleation and melting temperatures for SST nanotubes. This hypothesis comes from the observation that standard theories of nucleation and elongation predict that the rate of nucleation scales as the k^{th} power of the monomer concentration, where the critical nucleus contains k monomers, while elongation rates scale proportionally with the monomer concentration. Initial experiments showed that at a tile type concentration of 100 nM SST nanotubes formed well, but at significantly lower concentration yields were low and nanotubes were short (and thus difficult to image). Hence in the next step we choose a monomer concentration of 100 nM for each tile type.

Data set II: Tile type concentration of 100 nM. We repeated the experiments for $h \in \{8, 12, 14, 16\}$ but at a lower tile type concentration of 100 nM (instead of 1 μM). Figure S27(b) shows the data. There were two clear differences between the 100 nM and 1 μM data sets. First, the nucleation and melting temperature for the the 100 nM is significantly lower than that of the 1 μM data set. Second, the 100 nM data set has a wider gap between nucleation and melting temperatures than 1 μM. This gap, of roughy 2.5 °C between nucleation and melting temperatures, was deemed sufficiently large for our intended seeded growth experiments to implement IBCs.

The largest circumference tested was 16-helix, hence we used a 16-helix DNA nanotube implementation of IBC, which via our abstraction corresponded to a 6-bit IBC.

S5.1.2 Two experimental protocols for seeded growth with single-stranded tiles

The data in Figure S27(b) can be exploited for seeded growth of SSTs in the following two ways, the second of which was used in all seeded growth experiments for algorithmic self-assemby in this paper.

(i) Growth of SST nanotubes from pre-formed seeds. Here we have pre-formed seeds that are nanotube SST structures that have been formed in a previous anneal or temperature hold. Seeds are diluted to a low concentration (e.g. picomolar per tile type). The caption of Figure S28 explains the idea and shows data where SST nanotubes were used as seeds.

(ii) One-pot seed and SST nanotube growth. Here DNA strands are designed to form a seed structure at, or slightly above, the growth a temperature of the SST nanotubes. Figure 2a–d of the main text explains the idea: in a one-pot experiment strands are heated to 90 °C, then cooled quickly (e.g. 1 °C per minute) to a target temperature T where they are held for a day. Using the 16-helix DNA origami seed described in Section S3.2, the data in Figure S34 shows a temperature range such that in the absence of seeds, no or little off-seed nucleation occurs, but in the presence of seeds, on-seed self-assembly occurs. Also, for our complete 6-bit tile set, the data in Figures 2–4 in the main text and in Sections S8.1–S8.21, show that intended seeded growth occurs (i.e. the fact that we overwhelmingly saw the intended computations from the intended inputs, and that nanotubes with other unintended streptavidin patterns were extremely rare).

S5.2 4-bit copying tile set

We designed and tested a preliminary tile set similar to the subset of tiles implementing the circuit Copy (Section S8.9), but which copies only 4 bits instead of 6. Many of the basic design principles and growth protocol steps were worked out using this tile set, including growth from a DNA origami seed, unzippers, guards, biotin labeling, and some aspects of DNA sequence design.

The tile set is shown in Figure S29. Unlike the Copy tiles from the complete 6-bit IBC tile set, many tiles encode no bit (all their glues are $-_p$), and every tile that *does* encode a bit encodes only a single bit.

The abstract tile set in Figure S29 was implemented by two different sets of DNA sequences. In the first set of sequences, tiles representing a logical 1 had a 5′ biotin attached and the sequences were designed to use an equal target binding energy for all domains, intentionally not accounting for any energetic penalty due to the presence of a 5′ biotin. (The approach stands in contrast to the complete 6-bit IBC tile set, which used internal biotins and used stronger target binding strengths for domains with a biotin.) Also note that no proofreading is used here.

Figure S30 shows AFM images of an experiment on several different 4-bit inputs using the first set of DNA sequences. In this case, the seed is labeled with a σ as in Figure S14. The interpretation is similar to the Copy images in Section S8.9: successful copying of an input bit string results in stripes of streptavidin in the positions corresponding to the 1's.

At the empirically best growth temperature, we found that the only inputs that were successfully copied were 0000, 0100, and 0010. (Only the first two are shown in Figure S30.) Inputs containing more 1s grew only at lower temperatures (data not shown), which would be untenable for algorithmic self-assembly where the resulting bit pattern varies with the algorithm and the input. We hypothesized that the biotins negatively affected the binding energy. This hypothesis was confirmed experimentally (Figure S34), and a second set of DNA sequences was designed to implement the tile set of Figure S29, but now using sequences with larger predicted binding strength for domains that carry a biotin, to counteract the negative effect of the biotin. Figure S31 shows the result of this experiment, where many different sequences were successfully copied.

S5.3 Unzippers

The DNA origami seed and SST nanotube that grows from it are each topological cylinders, and to enable AFM imaging, each must be "unzipped": cut along the length of the tube to transform it into a topological rectangle (a.k.a., ribbon) able to lay flat on mica. The origami seed and the SST nanotube are both unzipped by DNA strand displacement, but the respective designs are different.

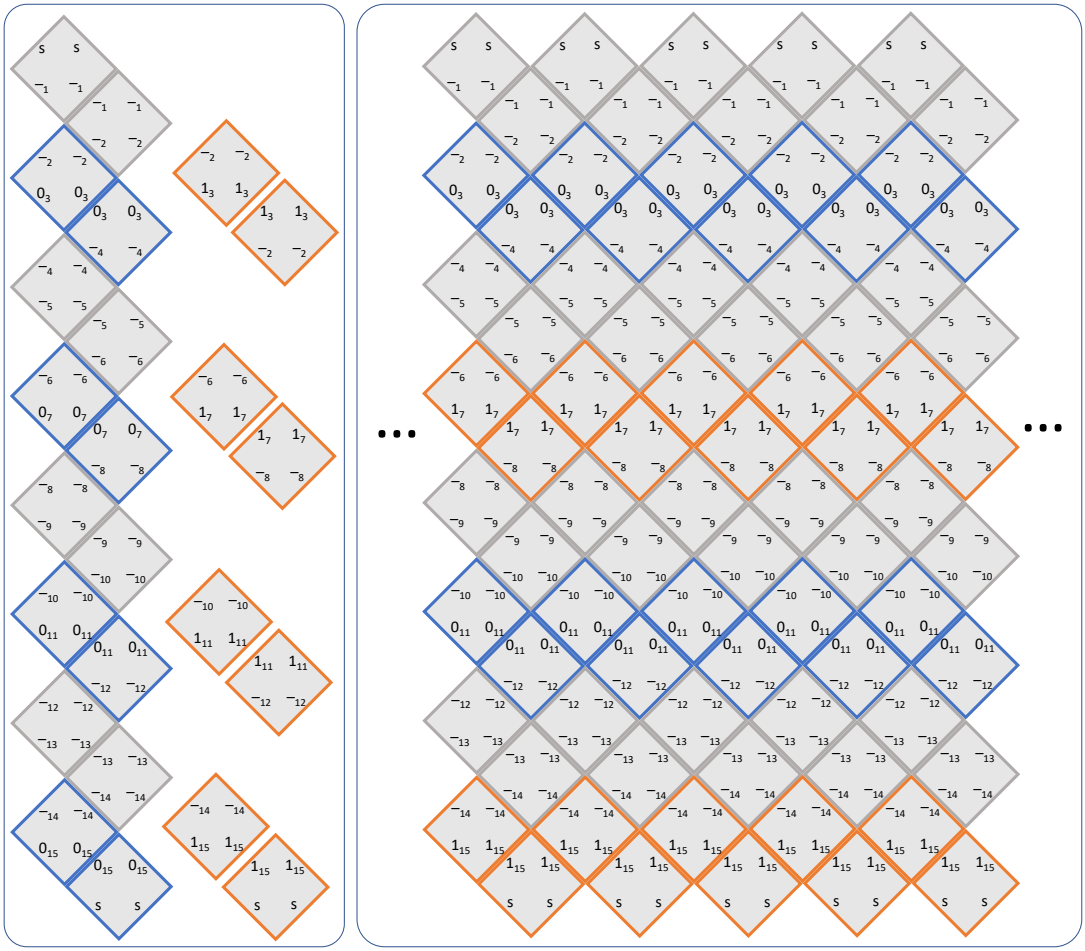


Figure S29: **left:** Tile set designed to copy 4 bits. This tile set was implemented as SST without employing a proofreading tile set transformation. Shown are one "layer" of tiles to represent the bit string 0000 along with the tiles representing "1" in each of the four bit-carrying positions. Each glue's subscript represents its vertical position. In four vertical positions $p \in \{3, 7, 11, 15\}$ on the layer, there is a glue 0_p, binding a pair of tiles, to represent logical 0 in position p, and another glue 1_p to represent logical 1. The seam glue is s. **right:** Portion of assembly showing correct copying of input bits 0101. Note that as in Fig.1 of the main text, the molecular assembly is a tube, so the bottom seam glue connects to the top seam glue.

S5.3.1 Origami seed unzippers

The seed unzipping strategy is the simpler of the two. Every staple crossing between helices 2 and 17 is bound by 16 bases on one of these helices and 8 bases on the other, with the latter having the 5′ end of the staple. To each of these 5′ ends, a unique extension sequence called a *toehold*, of length 8 bases, was added. This toehold remains single-stranded even after the DNA origami forms and is intended to help initiate DNA strand displacement of the unzipper strand once that strand is added. The toehold sequence was checked using NUPACK [28] to ensure that it did not introduce secondary structure into the staple.

Since it is not necessary to remove the entire staple to unroll the barrel, but merely to remove the region containing this crossover, unzipper strands consist of the Watson-Crick complement of the first 16 bases of the modified staple: the 8 bases complementary to the toehold, followed by the 8 bases complementary to the length-8 domain (or possibly length 7 or 9 if there is a deletion or insertion) of the staple closest to the 5′ end (i.e., the unzipper strand is just long enough to remove the staple up until the point of the first crossover). For example, for the bottom-left staple in Figure S11, whose toehold region is 5′-GAAGGTAT-3′, and which is bound to helix 2 by the domain 5′-ATGAGGA-3′, its associated unzipper

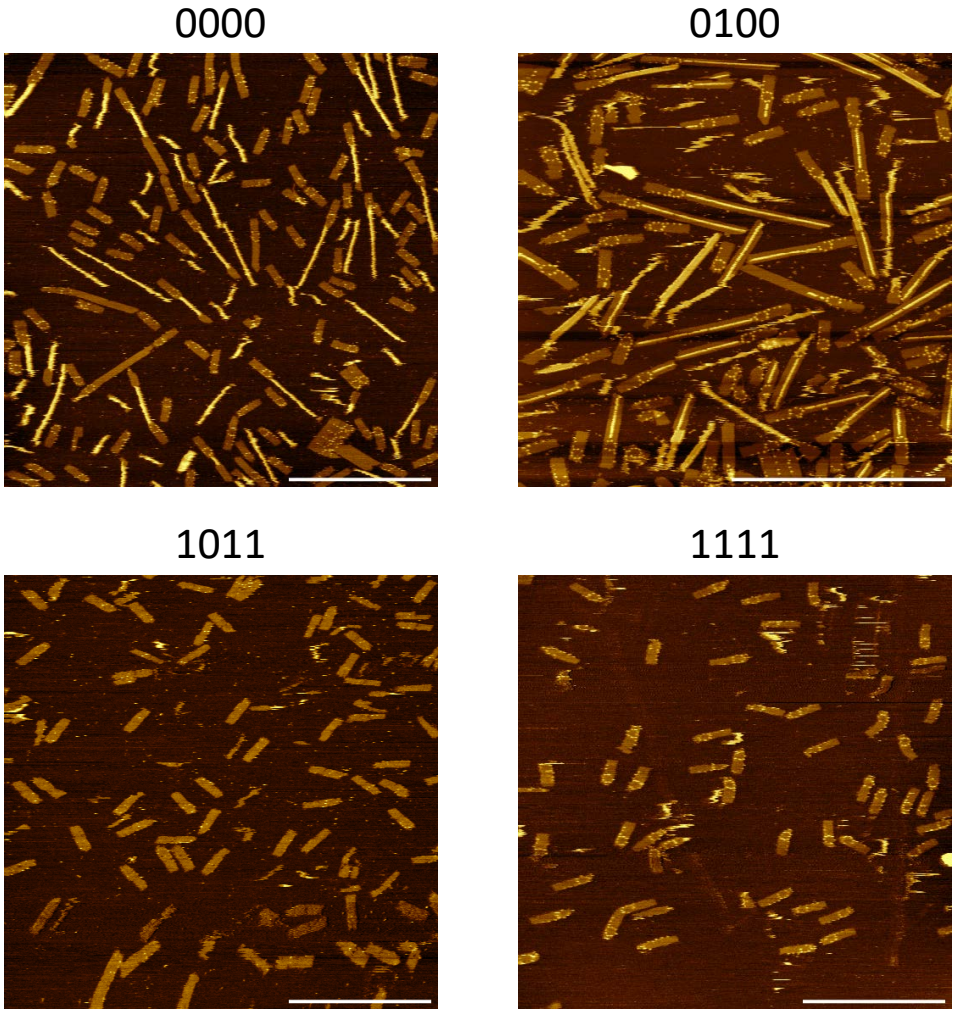


Figure S30: Results of initial implementation of 4-bit copying tile set from Figure S29 with DNA sequences that did not use stronger domains near biotins. Growth temperature is 53 °C; tile concentration is 100 nM; seed concentration is 1 nM. **Top:** Two inputs with ≤ 1 biotin per layer were successfully copied. **Bottom:** Two inputs with > 1 biotin per layer did not grow. DNA origami seeds have σ as in Figure S14. Scale bars: 1μm.

strand is 5′-TCCTCAT-ATACCTTC-3′.

S5.3.2 Tile unzippers

Wei, Ong, Chen, Jaffe, and Yin [30] showed that toehold mediated DNA strand displacement can be used to cut 2D SST rectangular lattices into multiple pieces. We apply a variation of their technique to unzip SST nanotubes.

The obvious adaptation of the unzipping strategy for the origami seed to the tile nanotube is not straightforward. There are two challenges present in SSTs that are not present in origami staples:

toehold extensions affect binding energy of SSTs more than staples: As discussed in Section S5.5.1, we found that the effect of a single biotin with a C6 linker (about 12 carbon bond lengths) on the 5′ end of a tile sufficiently alters the binding energetics to inhibit growth at the optimal growth temperature for biotin-less strands. A multi-nucleotide toehold (extension to a strand), being much larger in size than a biotin with a linker, would presumably affect binding energies even more. Experiments showed that DNA origami formation occurs significantly above the growth temperature of our tiles (at least

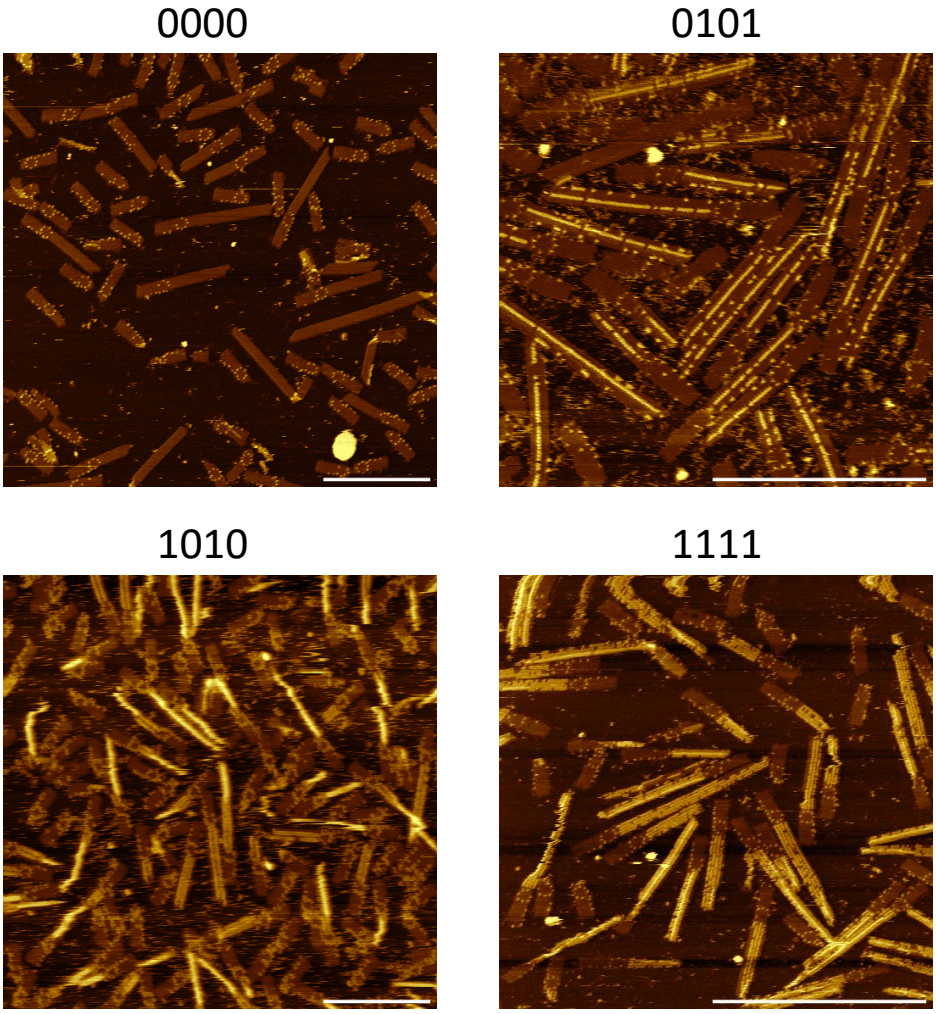


Figure S31: Results of redesigned implementation of 4-bit copying tile set from Figure S29, in which domains that use biotins were implemented with DNA sequences that bind more strongly and with internal biotins rather than 5′ biotins. Inputs 0101 and 1010 had some imaging problems, but it is clear that growth occurred. DNA origami seeds have σ as in Figure S14. Growth temperature is 53 °C; tile concentration is 100 nM; seed concentration is 1 nM. Scale bars: 500 nm.

up to 60 °C), so this weakening of staple binding energy does not significantly affect the stability of the origami. However, we wanted to avoid toehold extensions on tiles due to the tiles' greater sensitivity to binding energies. If we had included toeholds, we would have had to augment our design pipeline to account for their energetic cost during tile binding.

In thinking about how to get rid of toeholds on strands that must be "unzipped" via strand displacement, it is worth considering that toeholds aid in DNA strand displacement in two ways:

thermodynamic: If a toehold-mediated strand displacement results in more base pairs bound after the displacement than before (due to the toehold becoming double-stranded), then it is energetically more favorable to do the displacement.

kinetic: The toehold helps to co-locate the displacing strand with the double-stranded complex, speeding up the initialization of displacement.

Both purposes can still be achieved in the absence of toeholds by adding the displacing unzipper strands

in large concentration compared to nanotubes and free tiles. Kinetically, each double-stranded complex frequently encounters unzippers, boosting the kinetic rate of displacement. Thermodynamically, the entropy of mixing more strongly favors displacement by unzippers at higher unzipper concentrations.

However, we discovered experimentally that a large concentration of unzippers is not sufficient for unzipping to proceed at room temperature. We hypothesize that this is due to the second challenge:

a longer domain must be displaced in SSTs than staples: This problem is not so easily solved by increasing unzipper concentration. An unzipper strand on the DNA origami seed needs only to displace 8 bases before reaching the staple crossover between the helices. However, since each tile crossing over between two helices is bound to each helix by 21 bases, a tile unzipper strand needs to displace almost 3 times as many bases as an origami unzipper. However, as explained below, this is not merely a problem of slowing down by factor 3 (or even factor 9 since the displacement is modeled as an unbiased random walk, which takes n^2 expected time to move n steps).

Many uses of DNA strand displacement in the DNA nanotechnology literature displace a strand from a double helix that is physically separated from other double helices. A typical complex consists of a single double helix, possibly with some single-stranded extensions at nicks along the helix. That is to say, the only thing physically blocking the displacing strand from moving in place to bind to the complement is the other strand it is displacing.

In contrast, in the DNA nanotube design (as in SST canvases [15, 30]), the helix constituting the seam of the tube is flanked on two sides by two other parallel helices. The displacing strand must rotate around the middle helix as it displaces the other strand, threading itself between a pair of adjacent helices every 5.25 bases, which are repelling it electrostatically the whole time. When displacing only 8 bases on the DNA origami seed, this threading must happen only once (and it is possible that only a few out of 8 bases must be displaced before the remaining base pairs spontaneously break, releasing the crossover). But on the tile lattice, displacing 21 bases requires up to 4 of these threading events.

This intuition is supported by the following experimental observation: although unzipping of the DNA origami barrel can happen in a few minutes by simply adding a large excess of unzipper strands at room temperature, the same procedure failed to unzip any tile nanotubes by any detectable amount even after waiting 24 hours.

A solution to the latter problem was demonstrated by Wei, Ong, Chen, Jaffe, and Yin [30] and was independently suggested to the authors by Yannick Rondelez. Wei et al. [30] found that a rectangular single-stranded tile canvas could be cut into two pieces through DNA strand displacement, but only at higher than room temperature (they used 40° C).

Since we already hold above 50° C to grow the tubes, our unzipping procedure is simple: after the growth period (done in a thermal cycler), add unzipper strands directly into test tubes while they remain in the running thermal cycler and wait several more hours at the same temperature (24 hours sufficed to unzip most tubes). It would be risky to remove the test tubes from the thermal cycler for this step, lest they cool and encourage aggregation of free tiles and erroneous binding of tiles to existing nanotubes.

Several specific DNA strand designs implementing this scheme sound reasonable, but for us, some did not work, and others worked with one set of tile sequences but failed when the sequences were changed. Figure S32 illustrates the unzipper design that ended up working for the final sequences shown in most parts of this paper (including for the complete 6-bit IBC tile set).

There are a few key design features that appeared to be necessary, in the sense that other designs lacking one or more of these features either did not produce the desired result, or produced the desired result in initial experiments but did not work when new tiles with new sequences were designed:

use toeholds: (This means toeholds that are subdomains of existing binding domains, not single-stranded extensions as with the origami seed; see next design feature for more discussion.) At sufficiently large concentration, in principle the unzipper strands should be able to displace the tiles without a toehold. In practice, even at very large concentrations (100 μM of each unzipper, a 1000x excess over the tiles), unzipping is slowed or stopped without toeholds. For instance, no unzipping occurs if unzipper strands were designed so that the nicks between them coincide with the nicks between seam tiles (corresponding to shifting all the blue and orange unzipper strands in Figure S32 to the left by 5 base pairs).

In most partially zipped tubes, the zipped region is continuous and adjacent to the seed. See Figure S38(a), which shows 3 of 4 partially unzipped tubes in this configuration; in practice nearly all partially unzipped tubes had this structure, and the fourth (with seed label 231) is a rare counterexample showing that the phenomenon is not totally uniform. This gives evidence for the hypothesis that the unzipping usually begins at the opposite end of the nanotube from the seed and proceeds toward the seed, using the toeholds freed by the previous tile displacement to initiate the next. Other hypotheses rejected by this data are that the unzippers work in parallel on a single tube, which would distribute remaining zipped regions arbitrarily along the length, or that unzippers start near the seed and displacement proceeds in the same direction as tile growth, which would concentrate remaining zipped regions away from the seed.

toeholds are subdomains of existing binding domains, not single-stranded extensions of strands: As explained above, the DNA origami unzippers attach to toeholds that are simply single-stranded extensions of staples. However, this would have likely reduced the stability of attachment during the growth phase at least as much as the addition of biotins to the $5'$ ends of some tiles, which was experimentally demonstrated to disrupt growth. (See Section S7.5.)

each unzipper is eventually bound by $\gg 20$ base pairs: A preliminary design was similar to Figure S32, but each unzipper strand was only 21 bases instead of 42. In other words, each pair of unzippers cooperated to displace only a single tile rather than two tiles. However, at the growth temperature, 21 base pairs is only slightly favorable to attach rather than detach, so fully bound unzippers would be expected to detach, leaving tiles that then would themselves attached by only 21 base pairs. Further, there is an entropic penalty due to two strands becoming co-localized with the tube while only one strand is released into solution. Indeed, we observed in this shorter unzipper design that the long unzipping period resulted in frayed top and bottom edges of ribbons. (Images not shown.)

toeholds of length 5 bases: This is long enough to speed up the rate of strand displacement compared to no toeholds [88] yet short enough that a tile attached by two domains of length 5 will detach spontaneously at the unzipping temperature. The latter holds because the unzipping temperature is equal to the growth temperature, which is close to the melting temperature at which the off-rate of tiles binding by 20–22 total bases (two domains of length 10–11 each) equals the on-rate. The off-rate of tiles bound by merely 10 total bases is then significantly higher than this, so we expect a tile bound by only two toeholds to detach spontaneously.

unzipper strands bind to the nanotube rather than the seam tile: This helps to stabilize the structure as in the previous feature. If instead the unzipper strands bound to the seam tiles and carried them away from the nanotube, the resulting ribbon would then have all the tiles adjacent to the former seam bound only by two domains (21 bases), so one would expect the top and bottom edges of the ribbon to fray.

displacement at high temperature ($\approx 50^\circ$ C): Waiting for a full day at room temperature resulted in unzipping of DNA origami seeds, but no unzipping of tile nanotubes. Also, unzippers were heated before adding to samples, as described in Section S6.3.1, to prevent lowering of sample temperature and subsequent spurious nucleation of tiles.

waiting several hours: Some unzipping was observed after only 3-4 hours, but some variability in this was observed from one experiment to the next. Longer nanotubes require a longer unzipping period, which is unsurprising since unzipping is hypothesized to proceed sequentially from one end of the nanotube. For best results, unzipping required at least 24 hours (data not shown).

adding a small volume: Adding unzippers increases the sample volume, decreasing the concentration of tiles. This has the predicted effect of decreasing the on-rate, possibly below the off-rate so that some net dissociation of tiles from nanotubes might occur over the long unzipping period. To minimize this effect, the volume added was about 10 % of sample volume (i.e. 6 μl of unzippers added to 55 μl of sample, see Section S6.3.1 for details).

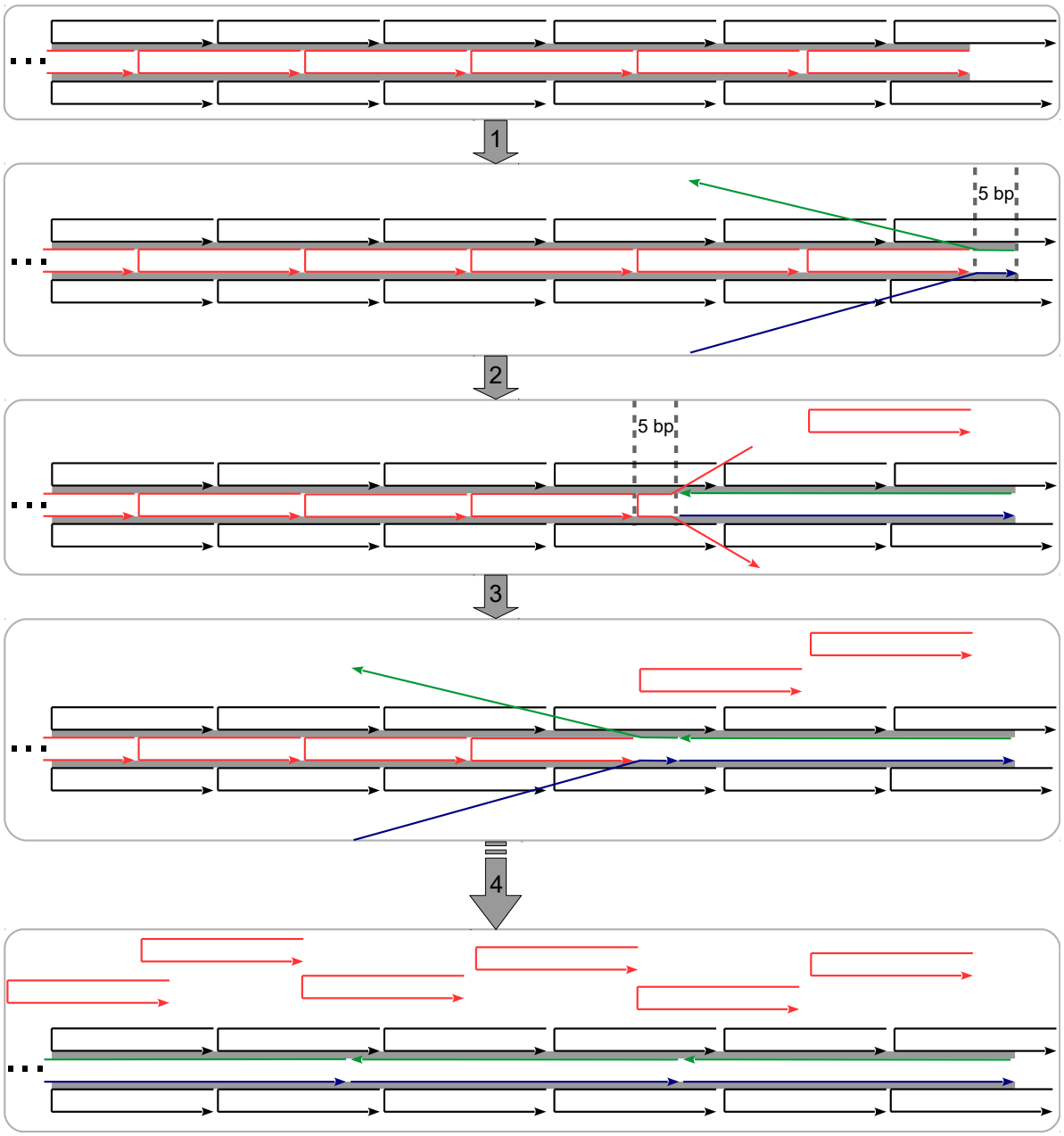


Figure S32: Unzipper design and intended strand displacement pathway for SSTs. For simplicity, only two helices are shown of the 16-helix nanotube. The seed and remainder of the tile nanotube are to the left and not pictured. There are two unzipper strands (blue and orange), one that binds to the "top" helix (helix 2 in the seed in Figure S11) and the other that binds to the "bottom" helix (helix 17 in the seed in Figure S11), to displace tiles along the seam (red). The unzipper strands are designed to **1)** bind to a length-5 toehold region adjacent to the seam tile to be displaced, which will be exposed at the growing end of the nanotube, then **2)** cooperate with the other unzipper strand to displace one seam tile completely, and a second seam tile almost completely, leaving 5 base pairs bound on two domains of the second seam tile. This tile is now only bound by 10 base pairs total, at a temperature ($\approx 50^\circ$ C) at which 20 base pairs are required for stable binding, so then **3)** the tile spontaneously detaches, opening toeholds for two new unzipper strands to bind. **4)** This repeats until the seam tiles have been removed and replaced by unzippers on the former tube (now topologically a rectangle). Each unzipper, being the length of four tile domains, is now bound by 42 base pairs (stable at the growth temperature of $\approx 50^\circ$ C). For entropic considerations, note that for each pair of unzipper strands that incorporate into the nanotube, a pair of seam strands is released. (If the unzipper strands were shorter, such that each pair triggers the release of just one seam strand, the reaction would have involved an entropic penalty.)

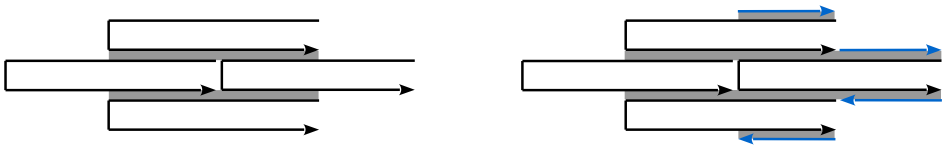


Figure S33: Guard strand design. **(left)** A proofreading block consisting of four tiles. **(right)** Guard strands (blue) binding to output domains of tiles that are external to the proofreading block.

S5.4 Guard strands

In prior work on self-assembly with double-crossover tiles, Schulman, Yurke and Winfree [9] used a technique of adding *guard strands* to shut down the process of self-assembly. There, a guard strand was a short strand that removed the sticky ends from the double-crossover tile (using toehold-mediated DNA strand displacement). Here, as guard strands, we use the complements of the four tile output domains that are "external" to a proofreading block, as illustrated in Figure S33. For experimental simplicity, in experiments on the complete 6-bit IBC tile set, rather than using a different guard set for each circuit, we used a "universal" mix of merely 54 guard strands that can be used for any 6-bit IBC implemented using our 355 tiles. The various design choices for guards are explained in this section.

Recall that the growth temperature for tiles is chosen slightly below the melting temperature of the tubes, where the melting temperature is defined to be the temperature at which the on-rate of tiles equals the off-rate of tiles bound by exactly two domains, each of length 10 or 11. Thus, at the growth temperature, binding by two domains is only slightly favorable, so binding by only a single domain is significantly unfavorable. However, at room temperature, binding by a single domain is favorable.

Also, most tiles are not incorporated into nanotubes. This is expected, because the on-rate—therefore also the melting temperature —depends on free tile concentration, so the melting temperature drops as free tiles are depleted to grow the tubes. Once the melting temperature reaches the (constant) growth temperature, the on-rate equals the off-rate. Although tiles continue to attach and detach from the ends of nanotubes, *net* incorporation of tiles is halted. Since free tiles remain in solution after the growth period, cooling the sample to room temperature makes it favorable for these tiles to bind to each other or to existing ribbons by a single bond, resulting in many unintended structures that occupy most of the mass of DNA observed in AFM images. Furthermore, binding to the end of a nanotube by only a single domain would result in tile attachment errors, also undesirable.

One solution would be to attempt to collect AFM images at the growth temperature, but this presents serious technical challenges (specifically, ensuring constant temperature throughout sample preparation for AFM imaging and during AFM imaging itself). We adapt a simpler method to solve this problem known as guard strands, and due to Schulman, Yurke and Winfree [9]. The purpose of guard strands is to deactivate unbound tile domains so as to prevent free tiles from binding to each other and to the ends of nanotubes.

Recall that each single-stranded tile consists of four binding domains, two used as "input" to bind the tile to the nanotube, and two used later as "output" to be used for binding to the inputs of two other tiles. Here, our guard strands are the complements of output domains that are "external" to a proofreading block, in principle limiting tile aggregation to at most 4-tile complexes.See Figure S33.

After the unzipping period, guard strands are added at 10× the concentration of tiles (1μ M per guard versus 100 nM per tile type), while the sample is still at the growth temperature, and allowed to mix for at least 5 minutes. Then the sample is rapidly cooled by removing from the thermal cycler and leaving at room temperature or in a fridge at 4 °C.

The idea is that upon cooling, when one binding domain becomes favorable, each free output domain is 10× more likely to encounter a guard strand than another tile. Thus, nanotube ends are "capped" largely by guard strands to prevent binding of new tiles, and most free tiles bind to guards rather than other tiles. Under standard imaging conditions, individual tiles do not bind strongly to the mica surface compared to large structures such as DNA origami seeds and seeded nanotubes. This holds also for complexes consisting of a few tiles, as well as guards, unzippers, and excess staples. Thus, a "buffer wash" suffices to clear away this excess DNA, which is done by repeatedly pipetting buffer solution onto the mica and then back off again. See Section S6.4.1 for details.

Mica stickiness (favorability for attracting DNA structures) varies significantly. Sometimes guard strands

alone suffice to create a clean background for imaging, and sometimes a buffer wash alone suffices. In rare cases the mica was insufficiently sticky for large structures and a buffer wash removed most of the DNA (origami and nanotubes as well).

Below we discuss some of these design decisions:

guards hybridize to individual single-stranded domains, with no intended strand displacement: This design is slightly different from the original guard strand design of Schulman, Yurke and Winfree [9]. That experiment used double-crossover tiles, composed of many DNA strands. One strand has a portion bound to the interior of the tile, and its two ends are the output sticky ends. Another strand has a portion bound to the interior of the tile, and its two ends are the input sticky ends. In this case the guard strands for a tile are the complements of these two strands. The guards remove the two strands from the tile by strand displacement, leaving most of the tile "interior core" intact but removing its domains for binding to other tiles. However, SSTs have no interior core, and the closest analog would be for each guard strand to be the complement of a whole tile strand. However, we feared this would result in strand displacement removing tiles from well-formed nanotubes. Furthermore, this would result in significant sequence complementarity between guard strands, so guards would bind to each other as well.

guards are complements of only output domains, not inputs: Adding guards for both output and input domains would imply that guards bind to each other, hence we added for guards for output domains only.

a small set of 54 "universal" guards were used for all IBC experiments: We wished to avoid the following two issues. (a) Having to prepare a custom guard strand mix for each circuit, since this is time-consuming and boring, or (b) preparing a mix of guard strands for output domains of *all* 355 tiles and adding that to each experiment. There are enough domains in total (calculation omitted) to imply that (b1) by using our protocols the guard mix needs to be prepared at a very high concentration of roughly 4 mM and (b2) that there will be lots of unneeded strands added to each experiment complicating any experimental debugging that might need to be done. We used the following strategy: we chose a subset of guard strands that are complementary to the output domains that are external to each proofreading block (including the non-proofreading 2×2 seam block), this gives a set of 54 guard strands.[37] Experimentally this gave similar results as having a set of guard strands complementary to all output domains of all tiles for the given circuit (tested on COPY, SORTING, LAZYSORTING; data not shown). The DNA sequences for these 54 guard strands are given in Section S11.

sample is rapidly cooled after guards are mixed: We also attempted to cool the sample slowly over several hours following mixing of guard strands. This resulted in greater aggregation of tiles. (Images not shown.) The reduced effectiveness of guards under slow cooling may be due to the thermodynamic favorability of tiles to bind to each other even in the presence of guards, as explained in the next design decision. Rapid cooling, by contrast, kinetically traps tiles in a meta-stable state of being protected by guards. Slow cooling was tested on a non-algorithmic tile set, so it is not known whether tile attachment errors would have increased as well under slow cooling.

guards are added only at end: There is a plausible mechanism by which tile growth could occur even with guards present. At the growth temperature, binding by a single domain is unfavorable, so we expect most tiles not to have a guard strand attached. Furthermore, even if a tile has guard strands attached to both output domains, this does not prevent it binding, since the *input* domains are not protected by guards. The output domains extending from the nanotube to which the tile binds could be bound to guards. However, the binding event, which would release the guards by (zero-toehold) strand displacement, replacing them with the input domains of the binding tile, is thermodynamically favorable at *any* temperature. This holds because number of bound base pairs is equal before and after, so binding is enthalpically neutral, but binding pulls one complex (the tile) out of solution and releases two (the guards), so binding is entropically favorable. Despite the plausibility of this hypothesis,

[37] $54 = 40 + 12 + 2 =$ [5 dual-output proofreading blocks $\times$ 4 output domains external to those blocks $\times$ 2 bits] + [2 single-output proofreading blocks $\times$ ((2 output domains external to those blocks $\times$ 2 bits) + 2 blank output domains)] + [1 seam block x 2 blank output domains (i.e. we did not add guard strands for the 2 output domains of the seam)].

experiments showed that the presence of guards prevents tile nanotubes from growing, which is why they are added only after the growth stage.

S5.5 Biotin-streptavidin labelling

In order to read out the process and results of computation by self-assembly we desired a method to distinguish tiles representing the bit 0 from those representing the bit 1. We chose to use biotin-streptavidin labelling to label 1 bits, and to leave 0 bits unlabelled, and to read bits by atomic force microscopy (AFM). We hypothesised a key advantage of biotin-streptavidin labelling over labelling by single-stranded or double-stranded DNA extensions (e.g., as seen in [14]) was that modifying a strand with a small molecule like biotin should incur less of an energetic penalty for hybridisation than a strand extension. We also hypothesised that the yield of streptavidin labels should be high when viewed by AFM. This scheme presented some design and implementation challenges that are discussed in this section.

S5.5.1 Biotin labels on tiles reduce binding strength

The first issue is that although biotin-streptavidin labelling has been used in previous experimental demonstrations of algorithmic self-assembly [9, 10], that work used multi-stranded double-crossover tiles where the biotin labels were on a central "core" of the tile and not any the sticky-end/glue domain of the tile. Here, since we are using SSTs, all bases of which are part of a binding-domain and we must account for, and then design against, any energetic penalty or gain associated with having a biotin.

Figure S34 shows an effect of biotins on the self-assembly of SSTs: The presence of biotins (either on the 5′ end of a strand or an internal dT modification in the middle of a binding domain) inhibits growth of tiles in a certain temperature range.

The experiments shown in Figure S34 use a single set of DNA sequences, designed to implement the 4-bit copying tile set in Section S5.2, but setting the seed input-adapter strands to 1111 and leaving out the tiles encoding the bit 0. Thus, this is a *non-algorithmic* set of SSTs that grow a 16-helix DNA nanotube. Around the circumference of this tube, there are four tiles that have a potential biotin. The tiles in the left column (5′ *biotin*) have a 5′ biotin attached to all four of these tiles. The tiles in the second column (*internal biotin*) have an internal biotin attached to all four of these tiles, see Figure S10. The tiles in the third column (*no biotin*) have no biotin attached to any tile. The final column (*no seed or biotin*) is another control with the same tiles as the previous column, but no DNA origami seeds.

The left two columns in the bottom row (54.1 °C) show what happens when there is no growth in the presence of DNA origami seeds: several DNA origami-length rectangles. In the other images in the left three columns, as the temperature is decreased, more nanotube growth is observed from these seeds. However, growth is less favoured in the presence of biotins than in their absence.

To compensate for this effect, at the sequence design level of abstraction (Section S4), we model the addition of biotin to a binding domain as subtracting 1.1 kcal/mol from its overall binding strength, so that to have roughly equal binding strengths, a domain with a biotin should have a nearest-neighbor duplex energy $\approx$ 1.1 kcal/mol greater than the nearest-neighbor duplex energy of a domain without a biotin. We chose internal biotins rather than 5′ biotins because an internal biotin can be placed in the middle of a binding domain, whereas 5′ biotins occur at the end of a binding domain, immediately adjacent to another binding domain. We conjectured that the effect of stacking interactions between adjacent binding domains could imply that *both* binding domains would have their interaction strength affected by the presence of a biotin. Therefore it would perhaps be necessary to model both domains as having reduced binding strength. However, the neighboring binding domain, depending on the algorithmic data being processed, could be sometimes adjacent to a biotin (when the neighboring domain represents logical bit 1) and sometimes not. The modified interaction strength would then introduce problems when a biotin is *not* present in the adjacent binding domain. However, with an internal biotin placed on a dT modified base in the middle of a binding domain, we could restrict the effect of the biotin to a single binding domain, and model the effect of biotin on only that domain.

The modification is available only for T bases (with the synthesis company IDT), which added a further constraint at the sequence design step. Furthermore, since IDT documents that the internal biotin resides in the major groove of the DNA helix, we chose to place the biotin on a base such that the major groove

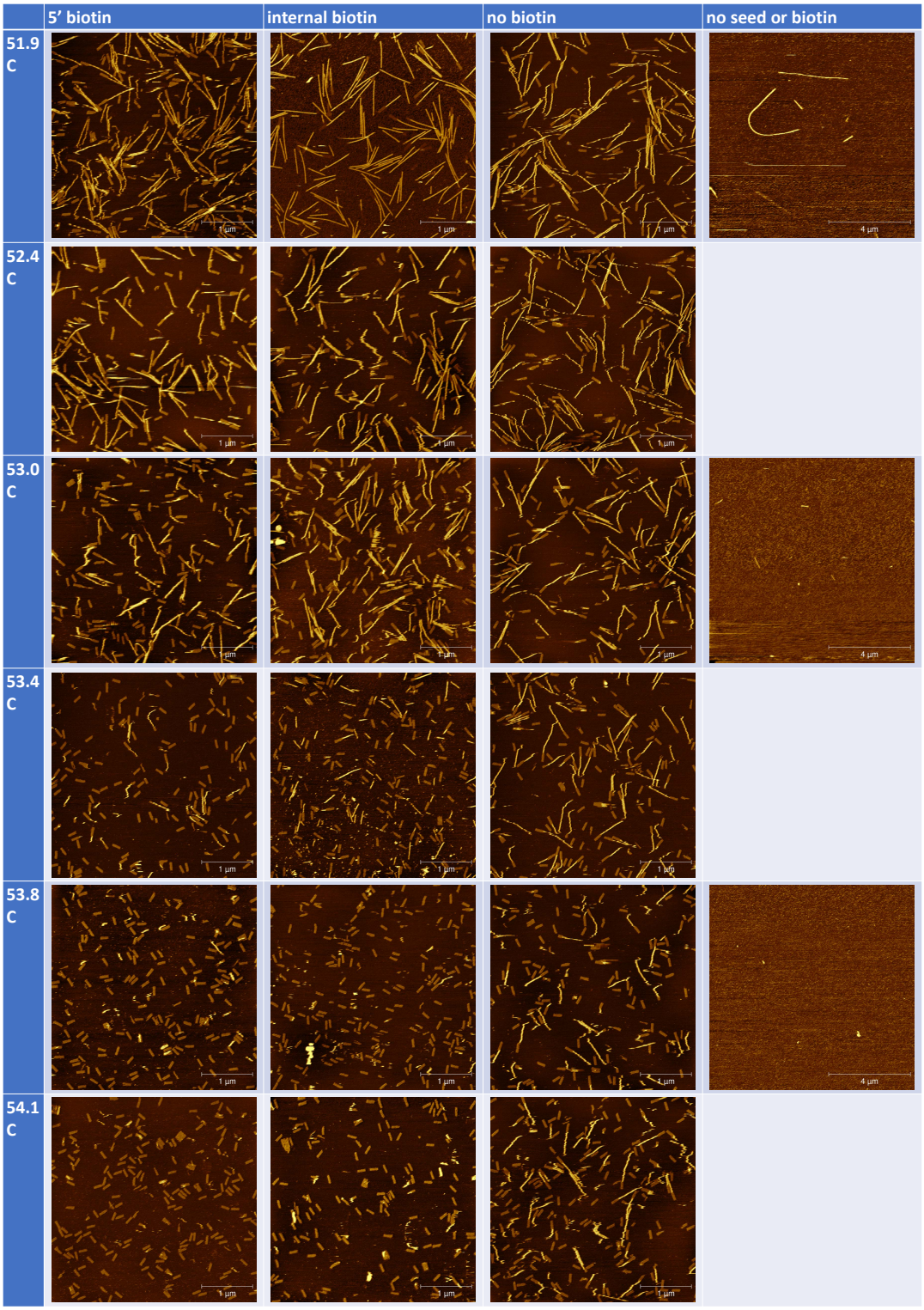


Figure S34: Growth at various temperatures to show two effects: (1) biotins inhibit tile growth, whether they are on the 5′ end or on an internal dT base, and (2) tiles do not spontaneously nucleate at temperatures favorable for seeded growth. Samples containing seeds and (hardcoded) SSTs were heated to 90 °C, then cooled to a chosen target temperature and held there for about a day, after which guard strands were added. Unzipping strands were not added (despite this origami seeds seem to readily unzip, presumably due to AFM tip and/or mica interaction).

was facing roughly toward the inside of the nanotube (which faces *up* when the nanotube is unzipped and deposited on mica, and where 'up' means 'away from the mica').

S5.5.2 Obtaining a high yield of streptavidin labelling

In initial experiments on the complete 6-bit tile it was found that streptavidin labeling of SSTs was significantly lower than that on the DNA origami seeds. Here, we detail our attempts at increasing streptavidin labelling efficiency. In the literature, streptadvidin-biotin binding yield has been studied in the context of attaching quantum dots to DNA origami [89, 69] and in the context of labeling DNA double-crossover tiles, where two biotins per tile (placed such that both could bind to a single streptavidin molecule) increased the labeling efficiency considerably [9].

Our protocol for streptavidin-labelling of nanostructures is described in Section S6.4.2, and can be summarised as slowly increasing streptavidin protein concentration after samples have been deposited on mica, until a high enough labelling efficiency is reached.

As highlighted in Figure S38(f) streptavidin binding to SST nanotubes was at a lower yield than binding to DNA origami seeds. Two key differences between the origami seeds and tile-lattice were: (i) biotins on DNA origami seeds were at the $5'$ end of staples, while biotins on SST nanotubes were on internally modified strands and (ii) DNA helices in the SST lattice are more tightly packed than in the DNA origami lattice. We hypothesise that some combination of (i) and/or (ii) led to the discrepancy in streptadvidin-biotin binding yield for the two kinds of nanostructure.

As highlighted in Figure S39, streptavidin binding to SST nanotubes varied slightly over several AFM scans of the same area. Section S7.4.1 further discusses incomplete streptavidin labeling as seen in our data.

We attempted to improve streptadvidin-biotin binding yield with a number of protocols. In our hands, we found that the best protocol was to slowly ramp up streptavidin concentration on the mica, as described in Section S6.4.2. Other protocols that were tried and that gave no improvement (and in some cases made the background—streptavidin bound to mica—worse) are summarised as follows. We tried various incubation temperatures (4 °C, room temperature, 35 °C) with streptavidin and DNA nanostructures on mica; various incubation times (from minutes to a day), mica washing steps, with either/both of sodium-magnesium or magnesium buffers; using nickel to fix DNA nanostructure to the mica surface, then adding streptavidin at a high concentration and then undergoing vigorous sodium-magnesium washes to clear the background; trying NeutrAvidin protein instead of streptavidin; adding streptavidin at a high concentration in a single step.

For future work one solution to increase the yield of the streptavidin labels would be to use multiple co-located biotin modifications, as has been done for multi-stranded double-crossover tile motifs [9].

S5.6 Repeating tile types along circumference

A key feature of *algorithmic* self-assembly distinguishing it from "hard-coded" or "uniquely-addressed" self-assembly [14, 15, 80, 17] is the reuse of the same type of DNA strand in multiple locations throughout the assembly. Our tile set indeed reuses tile types along the length of the nanotube: for any given row r and pair of inputs $i_1, i_2 \in \{0,1\}$, the proofreading block (set of four tile types) corresponding to gate g_r on inputs (i_1, i_2) is the same for any layer. However, tile types do not repeat in the orthogonal direction, along the circumference of the tube. If we were to design a new DNA origami seed with more than 16 helices, specifying an input $x \in \{0,1\}^n$ for $n > 6$, we would also need to design new tile types to implement the gates in the added rows.

However, for circuits in which the gates are the same in each row (e.g., sorting), reusing the same tile types for these gates is possible in principle. Such a circuit is a type of cellular automaton, which computes the same transition function in every cell. This would allow a *single* fixed tile set to be used to compute such a circuit on *any* size input. In this spirit, we designed an abstract tile set intended to simulate a Turing machine, wherein the tiles implementing Turing machine head state transitions and tape copying operations did not have a hard-coded row. Unfortunately, a preliminary experiment in this direction showed that some experimental challenges would need to be overcome.

Figure S35 shows a non-algorithmic tile set[38] that was designed to reuse tile types along the circumference

[38]One could think of this tile set as logically equivalent (ignoring the repeating of tile types along the circumference) to the Copy tile set of Section S8.9, but without the presence of any tiles representing 1, if one thinks of glues a, a', b, b' as both

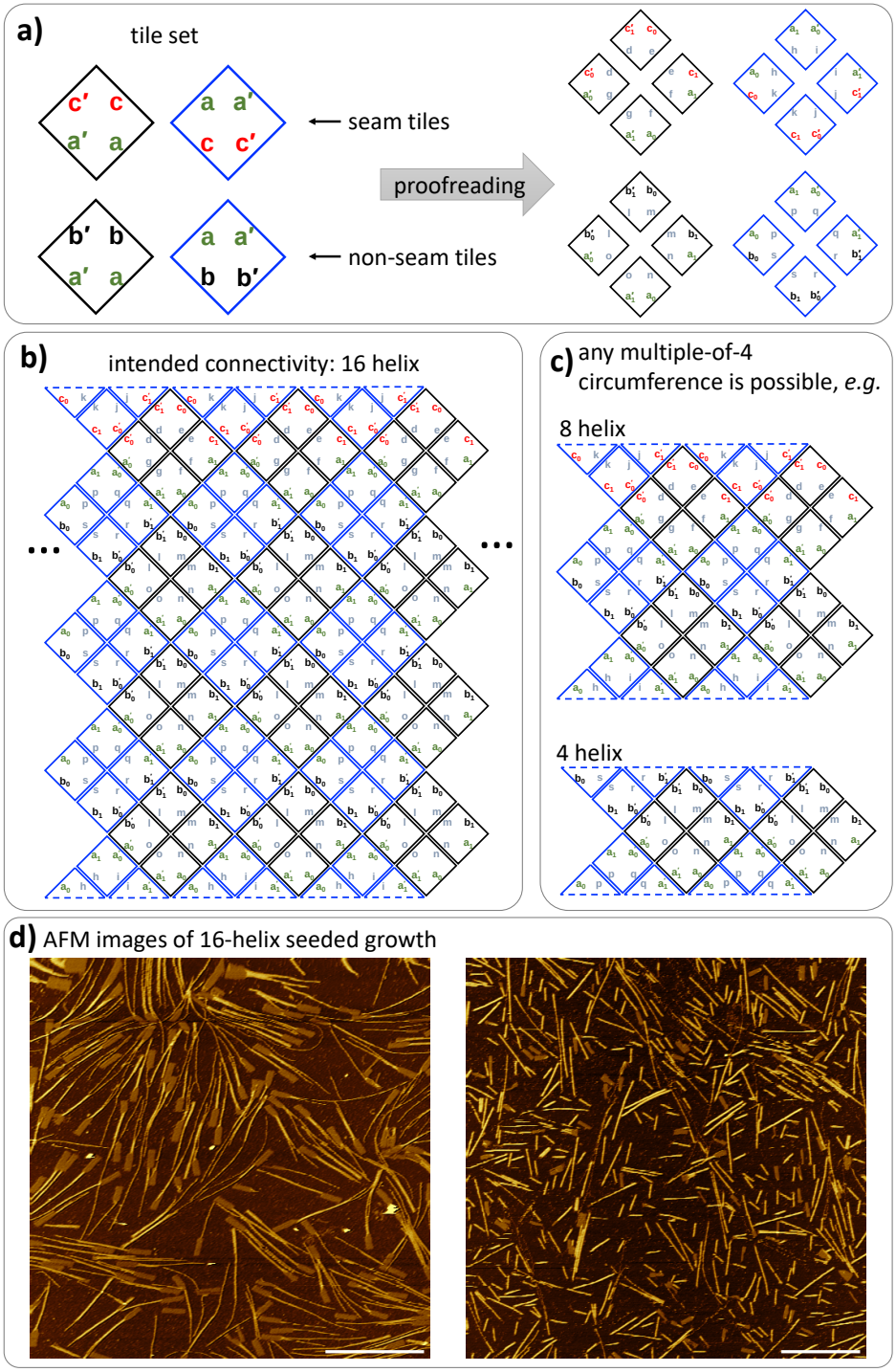


Figure S35: Tile set designed to reuse tile types along the circumference of a nanotube. **a)** Pre-proofreading tile set on left; post-proofreading tile set on right. **b)** The seed has 16 helices, with input-adapter strands encoding one c glue (the "seam" of the tube) and a/b glues elsewhere. **c)** Unlike our main tile set, which can only logically form tubes whose circumference is a multiple of 16, this tile set can form tubes with any circumference that is a multiple of 4. It is also possible to form such a tube without the seam, or with multiple seams. **d)** Experiments with a 16-helix seed show that tubes grow from the seed, but many shrink in their circumference, and some split into multiple smaller-circumference tubes. **scale bars:** 1 μm

representing 0.

of a 16-helix nanotube, along with AFM images showing the results.

The AFM images are not totally conclusive. However, it would appear that the tubes successfully grow from the origami seeds, but occasionally shrink in diameter as the tube grows, and in a few cases the tubes appear to split into two. We did not carefully test this hypothesis, but the images suggest that single-stranded tiles are possibly too floppy to prevent such lattice errors. It is interesting to note that, in addition to the approach used in this paper wherein each row is hard-coded by sequence, general theoretical methods for ensuring correct self-algorithmic self-assembly in the limit of "arbitrarily floppy" tiles have been explored [90].

S5.7 Comparison of DX, TX, and SST motifs for algorithmic self-assembly

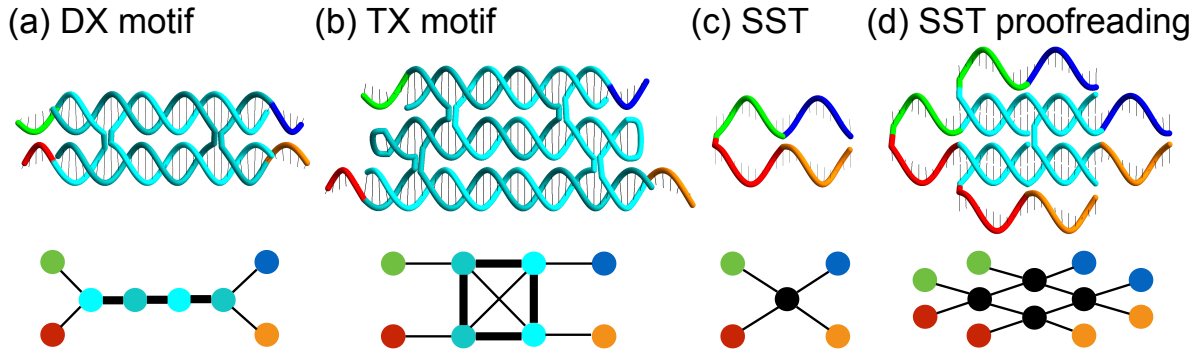


Figure S36: Comparison of tile motifs used in algorithmic self-assembly. The top diagram in each panel shows the tile structure, with the binding domains used for tile-tile attachments colored green, red, blue, and orange, and the tile core colored cyan. For the four SST shown in a proofreading motif, the uniquely addressed binding domains specific to that proofreading block are shown in cyan. The bottom diagram in each panel displays a connectivity graph for the structure. The green, red, blue, and orange dots represent strands on a hypothetical neighboring tile. For DX and TX motifs, each (light or dark) cyan dot represents a strand of the motif. For the SST motif and the SST proofreading block, each black dot represents a single strand. Edges between dots indicate hybridization to form a double-helix, either within the tile or involving binding domains attaching a tile to a neighbor. Thin edges indicate the strength of a tile-tile interactions; thick edges indicate intra-tile binding that is twice as strong or more, in terms of the number of base pairs formed.

To date, three general DNA tile motifs have been used for algorithmic self-assembly. The DX tile motif was introduced in ref. [76], shown to form periodic two-dimensional (2D) lattices [12] and nanotubes [86, 91], used in finite uniquely-addressed arrays [16], and exploited for 2D algorithmic self-assembly [6, 92, 93, 7, 8, 9, 10, 11, 94]. The DAO variant of the DX tile motif is shown in Figure S36(a). The TX tile motif was introduced in ref. [95] and used to make 2D lattices in the same work; it has also been shown capable of forming nanotubes [96] and exploited for one-dimensional (1D) algorithmic self-assembly [5]. A variant of the TX tile motif is shown in Figure S36(b). The SST motif was introduced in ref. [13] and used to make nanotubes in the same work; it has been used in finite uniquely-addressed arrays [15] and in simple 2D algorithmic self-assembly [97]. The SST variant used in this work is shown in Figure S36(c). While periodic three-dimensional (3D) crystals have been designed using a DNA tensegrity triangle tile [98] and finite uniquely addressed 3D structures have been created using SST [80, 17], algorithmic self-assembly in 3D has not yet been demonstrated.

The self-assembly of DX and TX tiles share several notable features that are in contrast to SST. Both DX and TX tiles are designed to have a substantial "rigid" core that, during an anneal from a high temperature to a low temperature, assembles from the tile's constituent strands (four of them, for the variants shown in Figure S36) before the tiles have significant interactions with each other to form lattices, arrays, or nanotubes. The separation of tile formation and array formation at different temperatures, and therefore different times, is due to the design containing many base-pairing interactions within the core (64 and 108 respectively for the DX and TX tiles shown in Figure S36) while the sticky-end binding domains mediating interactions between tiles are kept short (5 and 6 respectively). This separation was argued to be critical for self-assembly (e.g. as

argued in ref. [12]) because it was assumed that crystallization required rigid monomers, as supported by the observation that intrinsically disordered (i.e., floppy) proteins are difficult if not impossible to crystallize [99]. In further support of well-defined interactions between the rigid monomers, the sticky-end binding domains were designed to interact so as to form an intact double-helix with nicks. This was shown to keep the helix straight so that the relative orientation of the two tiles could be relied upon [100, 101, 102]. These features were also built into the standard models for algorithmic self-assembly, the aTAM and the kTAM, which consider rigid tiles whose interactions can be assumed to remain on a regular lattice [2]. The SST motif is profoundly different: due to the fact that single-stranded DNA is "floppy", unlike double-stranded DNA, individual tiles are intrinsically disordered prior to incorporation into a lattice, and small assemblies of two or three tiles have no well-defined local geometry relative to one another.

How these differences impact algorithmic self-assembly remains to be fully understood. It stands to reason that spontaneous nucleation of lattices, in the absence of a seed, have substantially more potential for off-lattice interactions with SST than with DX and TX tiles. This was considered in Section S5.6 as an explanation for our lack of success with a tile set that did not have uniquely-addressed rows. However, consistent with observations of lattice defects in previous algorithmic self-assembly with DX tiles [6, 93, 8], recent studies of putative nucleation intermediates and elementary tile attachment steps in DX self-assembly [103] have confirmed that prior to forming cycles[39], chains of DX tiles also have some flexibility and deformations that must be constrained during the attachment of additional tiles to an assembly. Consequently, differences in the potential for off-lattice interactions are a matter of degree, rather than absolute. That is, off-lattice interactions may be limited in their reach, at least after an assembly has nucleated or been seeded. It is therefore possible that SST tile sets for cellular automata and Turing machines of arbitrary width (therefore ruling out uniquely-addressed rows) could be successfully developed so long as rows are uniquely addressed *within a certain distance*, as would be the case for $k \times k$ proofreading [25, 65, 104, 67, 66].

Differences between the three tile motifs may also be seen in the light of proofreading. In this work, we used 2×2 proofreading with SSTs. The local geometry for a 2×2 proofreading block is shown in Figure S36(d). It is interesting to compare the four strands of an SST proofreading block with the four strands of a DX tile or of a TX tile. The glues inside each proofreading block (i.e. the four cyan binding domains in Figure S36(d)) are distinct and unique to that block, i.e. they are uniquely addressed. Similarly, the sequence design for the core of each DX or TX tile attempts to ensure that targeted binding interactions are highly specific [105] – i.e. they are also uniquely addressed. So each design involves four strands, four types of binding domains on the "outside", and uniquely addressed sequence interactions on the "inside", yet despite these similarities, the proofreading system can be expected to have a much lower error rate when implementing the same abstract tile set. The most crucial difference is that while the strands for a DX or TX tile come together at a higher temperature prior to tile self-assembly, proofreading blocks do not form in the absence of an existing assembly or seed. This is because the domains binding together tiles in the proofreading block have the same strength as the domains responsible for binding together tiles between proofreading blocks, as visualized in the strand connectivity graphs for these structures (bottom diagrams in Figure S36) Thus each DX or TX tile attachment occurs as a monolithic all-or-nothing event with some per-tile error rate ϵ, whereas a SST proofreading block assembles piecemeal, with success only when all four strands cooperatively find favorable reactions, requiring two matches on each side and enhancing the specificity of assembly, thereby achieving an error rate of roughly ϵ^2 [25]. Noting that each DX tile is roughly 6×14 nm while each SST is 3×7 nm, the net result is that SST with proofreading uses the same space and same number of strands as DX tiles without proofreading.

[39] We are here referring to cycles in the tile connectivity graph, with vertices representing tiles (as opposed to strands, as in Figure S36). For all three tile motifs, this graph is the intended 2D square lattice – perhaps finite for uniquely addressed structures, or wrapped into a tube for nanotubes. There are many cycles in this graph. However, during spontaneous nucleation, the graphs for some initial intermediate assemblies will be trees, as some initial steps will involve tile attachment by a single binding domain. A tile attachment event that binds by two binding domains will form a new cycle, which necessitates paying a loop-closure energy penalty due to the resultant reduced flexibility of the assembly.

S6 Algorithmic self-assembly experiments

S6.1 Preparation of DNA strand stocks and reagent stocks

Unless otherwise stated all DNA strands were ordered lyophilized and unpurified (termed Standard Desalting) from Integrated DNA Technologies (IDT). Tile strands with biotin modifications[40] were ordered HPLC purified. The ability to obtain low-error-rate algorithmic self-assembly using largely unpurified DNA strands dramatically simplified the experimental procedures (relative to prior experimental demonstrations of algorithmic self-assembly, all of which used purified DNA strands) and was instrumental to making the scale-up of tile set complexity feasible.

Preparation of tile strands. Tile strands were brought to a target concentration of approximately 100 μM as follows[41]: A volume v of ultrapure H_2O was added to the screw-top tube containing the lyophilized tile strand to target 150 μM, where v is calculated using the claimed number of nanomoles from IDT. Absorption of 260 nm light was carefully measured twice in a UV-Vis Spectrophotometer (NanoDrop 2000c, with a 2 μl droplet placed on detector, cleaned with a Kimwipe, repeat with another droplet). Implied concentration was calculated using the extinction coefficient supplied by IDT, and a volume of ultrapure H_2O was added to target 100 μM. Typically, this two-step procedure resulted in a concentration ≥ 100 μM and < 104 μM, but in those cases where the result was ≥ 104 μM (tile strand concentrations were never found to never be < 100 μM at this step), then a volume of ultrapure H_2O was added to again target 100 μM. Until concentration was in the desired range of ≥ 100 μM and < 104 μM, combinations of this dilution step and/or a (concentration-increasing) step using evaporation in a 35 °C heated Vacufuge (Eppendorf), were repeated as necessary. This procedure resulted in 356 individual tile strand stocks[42] with concentrations in the range 100 to 104 μM (except for two tile strands accidentally left at 104.6 μM and 105.3 μM respectively), distributed with mean 101.9 μM and standard deviation 0.8 μM. Tile strands were aliquoted into proofreading blocks (see next paragraph) and placed in the fridge. The 356 individual tile strand test tubes were placed in long-term storage stored at -20 °C; and were removed only if a proofreading block aliquot was used up in future experiments.

Preparation of proofreading block mixes. For each 2×2 proofreading block, the 4 tile strands for that block were mixed in equal volume to form a *proofreading block mix*. This gave a total of 89 proofreading block mixes with each such mix having a target individual tile type concentration of 25 μM. Proofreading block mixes were typically stored at 4 °C (i.e. unfrozen in order to avoid frequent freeze-thaw cycles).

Preparation of input-adapter strands. Input-adapter strands were ordered lyophilized and unpurified (Standard Desalting) from IDT and were targeted to 90 μM using a two-step procedure analogous to that for the preparation of tile strands: first targeting 150 μM and then 90 μM (possibly adding water to decrease strand concentration, or placing in a 35 °C heated Vacufuge to increase concentration), with the resulting strands measured on a NanoDrop 2000c to have concentrations in the range 88.38 to 92.75 μM, distributed with mean 91.2 μM and standard deviation 1.0 μM. Input-adapter strands were typically stored at 4 °C (i.e. unfrozen in order to avoid frequent freeze-thaw cycles).

Preparation of origami seed strands, guard strands and unzipper strands. Single-stranded M13mp18 DNA (scaffold strand) was purchased from New England Biolabs (Catalogue #N4040S), Bayou Biolabs (Catalogue #P-107) and Tilibit Nanosystems (type p7249), all of which claim to supply the same sequence. To write digits on DNA origami seeds, some staples had two versions: an unmodified strand and a 5′-end biotin-modified strand. Staple strands were ordered in plates and unpurified (standard desalting) in water from IDT (including biotinylated staples).

[40] We used biotin modifications internal to the tile strand and attached to a T base, written 'Int Biotin dT', or '`/iBiodT/`', in IDT's nomenclature.

[41] In hindsight, it would have been better to store all our stocks in TE buffer rather than pure water, so as to reduce DNA degradation by hydrolysis.

[42] Throughout this paper we say that our complete 6-bit IBC tile set had 355 tile types. Note that the seam tile is "repeated" in the sense that it occurs in two positions within its 2×2 block. For convenience during experiment protocols we had two test tubes with the same seam tile (we ordered the seam tile twice), hence there were 356 tile strand stocks in total.

Guard strands (short length-10 or length-11 DNA strands) that are the complements of "output" tile strand domains (the domains at the two ends of the strand) were ordered in plates and unpurified (standard desalting) in water at a high concentration of 500 μM (since many will be mixed together and we want guards to be in high excess over tile domains). To simplify experiment preparation we used a single "universal" set of guard strands for every experiment. The set of guard strands consisted of the complements of the domains of the four output domains of every proofreading block over the *full* set of 355 tile strands (experiments showed that this set of guards, which bind to around merely 1/4 of the tile domains in any given experiment was sufficient to prevent unwanted self-assembly while imaging by AFM). Recall that there are 4 output domains for each proof-reading block, and each of these domains encodes a (row, bit) pair, this gives 8 distinct domains for each of 5 rows (i.e. 40 guards so far) that encode 2 bits, 6 distinct domains for each of 2 rows (+12 guards) that encode 1 bit and for the special (non-proofreading) block U1 we include only those 2 output domains that are not complement to the tile C1 (+2 guards) (in order not to have guards compete with unzipping strands) giving a total of 54 guard strands. This "universal" set of 54 guard strands was used for all 6-bit experiments with the final concentration of each guard being 10-fold (1 μM) over each tile strand (100 nM).

The 2 tile unzipper strands and the 26 DNA origami seed unzipper strands were ordered in water typically in the concentration range of 200 μM to 500 μM. In an experiment tile unzippers were added at 20-fold excess concentration (i.e. 2 μM) over tile strands (i.e. 100 nM), which in turn is a 10-fold excess over the seam strand (there are two instances of the seam strand, each at concentration 100 nM) and origami unzippers were added at 733-fold excess concentration (i.e. 733 nM) over scaffold (i.e. 1 nM).

Preparation of streptavidin protein. Streptavidin was ordered from Rockland Immunochemicals (catalogue #S000-01) in a septum-sealed bottle with 5 mg of lyophilized protein. A 1 mg/ml (18.2 μM) solution of streptavidin was prepared by adding 5 ml of ultrapure water, then gently mixing by hand and incubating for 5 minutes at room temperature. The ultrapure water was added using a syringe through the septum to prevent loss of protein flakes. Assuming a molecular weight of 55,000 Da for streptavidin, the of 5 ml volume of water targeted a protein concentration of 18.2 μM. The bottle was then very gently mixed by rolling around by hand for one minute, leaving to sit for 5 minutes and repeating. The septum was removed and the streptavidin solution was put into 20 aliquots of 100 μl each which were frozen at -20 °C. The remaining (non-aliquoted) streptavidin solution was discarded. For use in an experiment, an aliquot of the frozen streptavidin solution was thawed and spun for a few seconds in a bench centrifuge. A few microlitres of the thawed aliquot was diluted to 5 μM in ultrapure H_2O and stored in the fridge, or at room temperature (to be added directly to the sample in experiments as described below). Streptavidin protein may aggregate which is seen as large blobs under AFM and in this case the solution was discarded and another aliquot thawed and prepared. We note that in our hands, best results were found with (a) very gentle mixing of streptavidin solutions with a pipettor (up and down at most once), and (b) thawing exactly once as above (i.e. avoid freeze-thaw cycles). Other variations on this protocol were tried, none gave cleaner AFM imaging than this.

Buffer. Samples were annealed in 1x TAE/Mg^{++}, a shorthand for a mixture of Tris-Acetate-EDTA buffer at pH 8.0 and 12.5 mM Mg^{++}. Samples were stored in water up until TAE/Mg^{++} was added.

S6.2 Preparation of samples for execution of a 6-bit circuit computation

The experimental protocol for each 6-bit experiment was automatically generated using a custom Python script. The advantages of this approach over manual protocol generation include (a) convenience of specifying the large combinatorial set of DNA strands for a given experiment (tile strands, input-adapter strands, origami seed strands), (b) protocol consistency between experiments, and (c) reduction in the likelihood of human error while generating and executing a protocol. We feel that carrying out so many experiments with so many DNA strands would have been an infeasible task in a reasonable time-frame without a significant amount of automation.

6-bit circuits were specified in the Python programming language syntax. For a two-input gate position, the gates are specified as a list of (8-bit string, probability) pairs, or more precisely in the format

$$[(s_1, p_1), (s_2, p_2), ...(s_k, p_k)]$$

where $s_i \in \{0,1\}^8$ for $i \in \{1,2,\ldots k\}$ and $\sum_{i=1}^{k} p_i = 1$ and for single-input gates $s_i \in \{0,1\}^2$. For deterministic circuits, each gate consists of a single such $[(s_1, p_1)]$ pair, for example for the rule 110 6-bit circuit:

```
layer_description = 'Rule 110'
circuit_inputs = ['000001', '010100', '110001']
seed_patterns =  ['001',    '110',    '301'   ]
layer = [
            [('00',1)],
            [('00111111',1)],              # OR; OR
            [('00111000',1)],              # b XOR c; (NOT(c) AND b)
            [('00111111',1)],              # OR; OR
            [('00111000',1)],              # b XOR c; (NOT(c) AND b)
            [('00111111',1)],              # OR; OR
            [('01',1)]
        ]
```

A Python script takes such a circuit specification as input and then outputs (a) computer simulations (drawn to PDF using a combination of Python and PGF/Ti*k*Z [see Figures 2–4 in the main text]), (b) detailed experimental protocol for preparation and mixing of all DNA strands and reagents (see example immediately below) and (c) test tube names and labels for each test tube to be used in an experiment. A LaTeX program then geometrically lays out, on a letter-sized PDF page, these test tube labels for printing onto sticky labels that are thermocycler/PCR machine heat-proof and placed on each test tube.

Typically several samples, from several different circuits, were prepared at one time (up to 32, depending on the experiment). An example protocol is shown next, for the RULE110 circuit on inputs 000001, 010100 and 110001.

```
**********************************************************************
Generating experimental protocol for the following circuit:

                    Rule 110

**********************************************************************
A. Advance solutions to prepare:

Prepare the following: M13 (at 10nM), MQ H2O, 10xTAE/Mg++.
Mixes to prepare: tiles, staples, input-adapter strands.

1. Tiles:
Prepare a 0.5ml test tube with label: 'r110 tiles'.
Add 2ul of each of the following proofreading blocks:
  U1;__->__
  U2;_0->_0  U2;_1->_0
  U3;00->00  U3;01->11  U3;10->11  U3;11->11
  U4;00->00  U4;01->11  U4;10->10  U4;11->00
  U5;00->00  U5;01->11  U5;10->11  U5;11->11
  U6;00->00  U6;01->11  U6;10->10  U6;11->00
  U7;00->00  U7;01->11  U7;10->11  U7;11->11
  U8;0_->0_  U8;1_->1_
This is a deterministic circuit, so mixing 2uL of each
proofreading block gives 50uL volume at 1uM per tile type.

2. Origami seed staple mixes.
For each seed mix do the following:
  To make 100uL vol and 100nM conc of the staple mixes for the seed 001:
    Put label on test tube: stap 001
    Add 90 uL of MQ H2O
    Add 5.5uL of noncoding staple mix
    Add for the following 3 amounts of coding staples:
      1.5uL of p1=0 staple mix,
      1.5uL of p2=0 staple mix,
      1.5uL of p3=1 staple mix.
Do the same for the other 2 seeds: 110, 301.

3. Input-adapter strands:
There are 8 input-adapter strands used in each deterministic experiment.
```

```
Input-adapter strands are stored  at a concentration of approximately 90uM.
The following subset of gates will be used to choose input-adapter strands:
  00 00111111 00111000 00111111 00111000 00111111 01.
There are 3 input strings: 000001 010100 110001.

To get 180uL with each input-adapter at 1uM:
  * Place 164uL of MQ H2O into a 0.5ml test tube with the input-adapter label.
  * Add 2uL of each of the 8 input-adapter strands.

Prepare the following 3 input-adapter mixes:
  Mix name: r110 000001.       Input: b1=0, b2=0, b3=0, b4=0, b5=0, b6=1    Seed label: 001
    adpU1;__->__
    adpU2;_;b1=0;_0->_0
    adpU3;b1=0;b2=0
    adpU4;b2=0;b3=0;00->00
    adpU5;b3=0;b4=0
    adpU6;b4=0;b5=0;00->00
    adpU7;b5=0;b6=1
    adpU8;b6=1;_;1_->1_

  Mix name: r110 010100.       Input: b1=0, b2=1, b3=0, b4=1, b5=0, b6=0    Seed label: 110
    adpU1;__->__
    adpU2;_;b1=0;_0->_0
    adpU3;b1=0;b2=1
    adpU4;b2=1;b3=0;10->10
    adpU5;b3=0;b4=1
    adpU6;b4=1;b5=0;10->10
    adpU7;b5=0;b6=0
    adpU8;b6=0;_;0_->0_

  Mix name: r110 110001.       Input: b1=1, b2=1, b3=0, b4=0, b5=0, b6=1    Seed label: 301
    adpU1;__->__
    adpU2;_;b1=0;_1->_0
    adpU3;b1=1;b2=1
    adpU4;b2=1;b3=0;10->10
    adpU5;b3=0;b4=0
    adpU6;b4=0;b5=0;00->00
    adpU7;b5=0;b6=1
    adpU8;b6=1;_;1_->1_

Circuit and input test tube labels for this experiment  (& is a separator symbol):
  r110 001 & r110 110 & r110 301
DNA origami seed staple mix test tube labels for this experiment:
  stap 001 & stap 110 & stap 301
Input-adapter test tube labels for this experiment:
  r110 000001 & r110 010100 & r110 110001
```

Next we prepare a test tube for our running example of the RULE110 circuit, with origami seed label 001 (see Section S3.2.3) and input 000001. The previously prepared solutions are mixed into a 0.5 μl Eppendorf LoBind Tube (CAT No. 0030108035)[43] according to the following table to give 50 μl sample. Note that in the table, the stock concentration for tiles refers to the concentration per tile type (i.e. from the above protocol we have prepared a "stock" of 100 tile types with each tile type at 1 μM). Likewise for M13, staples, and input-adapter strands.

S6.3 Executing a 6-bit circuit computation

The previously prepared sample (Table 1) is placed in a thermocycler (PCR machine) and the following program is run:

1. Set lid of thermocycler to 105 °C for entire program
2. Hold at 90 °C for 10 minutes

[43] DNA LoBind tubes were chosen to lower the likelihood of DNA sticking to the tube on long, several-day, temperature holds.

	Stock concentration	Volume to take	Target concentration
ultrapure H_2O		25 μl	
M13mp18 scaffold	10 nM	5 μl	1 nM
staple strands (001)	100 nM	5 μl	10 nM
input-adapter strands (000001)	1 μM	5 μl	100 nM
tile strands (RULE110)	1 μM	5 μl	100 nM
10x TAE/Mg^{++}	10x	5 μl	1x
Total		50 μl	

Table 1: Six stocks that when mixed in the volumes shown to give a solution containing a tile set for a 6-bit circuit and an input. Stock and target concentrations are given *per strand type*, e.g. per staple type, per tile type, etc. In this case the circuit is RULE110, with 6-bit input 000001 and seed label 001. The same volumes works for any circuit and input. Each such 50 μl sample was placed in a 0.5 μl Eppendorf LoBind Tube in preparation for placing in the thermocycler.

3. Drop to 50.9 °C at a rate of 1 °C/minute.
4. Hold at 50.9 °C for 6 hours, 50.8 °C for 6 hours, 50.7 °C for $\geq$ 24 hours
5. While holding at 50.7 °C, add unzipper strands. Wait $\geq$ 12 hours.
6. While holding at 50.7 °C, add guard strands. Wait $\geq$ 3 minutes.
7. Remove samples for imaging by AFM, thus allowing them to quickly cool to room temperature. Samples may be imaged immediately, or after spending up to a few days in the thermocycler or on the bench.

Protocl variations that were also (successfully) used included (a) holding for several days (rather than 1 day) at 50.7 °C, and (b) dropping from 90 °C to a target temperature of 50.8 °C and holding there for a day or more.

Experiments typically involved running many (up to 32) samples (circuit with an input) in parallel in a single thermocycler.

It should be noted that there is variation among thermocycler temperatures. For the machines we used (Eppendorf Mastercycler Gradient) we measured temperature variability of $\pm$1.0 °C between machines and around $\pm$0.1 °C across the metal plate of any single machine. Hence changing thermocycler involves recalibrating the experiment to the new thermocycler. Specifically, finding a good growth temperature involved searching over various holding temperatures in the previous protocol (too hot and one sees little or no growth from origami seeds, too cold and one sees both unseeded growth (growth of nanotubes with no seeds attached) as well as high tile attachment error rates. No further tuning was required per circuit, nor per circuit-with-input.

S6.3.1 Adding unzipper strands and guard strands to hot samples

Preparation of unzipper strand stock was described in Section S6.1. As noted in that section, in an experiment tile unzippers were added at 20-fold excess concentration (i.e. 2 μM) over tile strands (i.e. 100 nM), and hence at a 10-fold excess over the seam strand (there are two instances of the seam strand, each at concentration 100 nM) and origami unzippers were added at 733-fold excess concentration (i.e. 733 nM) over scaffold (i.e. 1 nM). This was achieved as follows. A volume v μl $= s \cdot 7$ μl, where s to the number of samples in the thermocycler (recall that a 'sample' is a circuit with input in a 0.5 μl test tube), of unzipper strand stock was placed in the thermocycler at the sample holding temperature, as was 50 μl of 10xTAE/Mg^{++}. After 5 minutes of temperature equilibration time, $v/10$ μl of 10xTAE/Mg^{++} was added to the v μl of unzipper strand solution, thus making the unzippers be in a 1xTAE/Mg^{++} buffer solution. After 5 more minutes, 6 μl of (hot) unzipper strand solution was added to a (circuit-with-input) sample (as described in more detail below) and the thermocycler lid closed. For nanotube unzipper strands (that remove the seam tile), this targets a concentration of roughly 2 μM per individual unzipper strand, i.e. 20-fold excess over each tile strand (each tile strand is at 100 nM) and 10-fold excess over the seam strand (seam strand is at concentration 200 nM). For DNA origami seed unzipper strands, this targets a concentration of roughly

733 nM per individual unzipper strand, i.e. a 733-fold excess concentration over scaffold (which is at 1 nM) and a 73-fold excess over individual staple concentration (which are each at 10 nM).

Preparation of guard strand stock was described in Section S6.1. Guard strands were added to samples as follows (almost the same as the unzipper protocol). A volume v μl $= s \cdot 7$ μl, where s to the number of samples in the thermocycler, of 10 μM guard strand stock was placed in the thermocycler at the sample holding temperature, as was 50 μl of 10xTAE/Mg^{++}. After 5 minutes of temperature equilibration time, $v/10$ μl of 10xTAE/Mg^{++} was added to the v μl of guard strand solution. After a further 5 minutes, 6 μl of guard strand solution (which is now hot and in 1xTAE/Mg^{++}) was added to a (circuit-with-input) sample (as described in more detail below) and the thermocycler lid closed. This targets a concentration of roughly 1 μM per individual guard strand in the circuit-with-input sample (i.e. 10-fold excess over each tile strand).

Fast, accurate pipetting of hot solutions. Accurately and quickly pipetting an intended volume of solution from one hot test tube to another in a thermocycler takes some practice. On the one hand, when accurate volume is preferred over accurate temperature we simply used a room-temperature pipette tip and without pre-heating or mixing quickly moved volume from one test tube to another (for example, when adding 10xTAE/Mg^{++} to guard or unzipper strands). On the other hand, when not overtly cooling the sample was of paramount importance (e.g. when adding unzippers or guards to a sample undergoing algorithmic growth) we developed the following procedure. The pipette tip was pre-heated by mixing up and down in the source PCR tube 2–3 times, ejecting all fluid into the source PCR tube, then withdrawing the target amount, visually inspecting to see that the volume was approximately correct and that there were no air bubbles in the tip while simultaneously quickly moving to, and fully ejecting, into the target PCR tube sample volume, then quickly removing the tip from the same, and quickly closing the target PCR tube. After doing 2 or 3 of such transfers, the PCR machine lid was firmly closed for several minutes in order not to let the samples overtly cool down or experience condensation inside the air-volume of the PCR tubes. The process was repeated until completion.[44]

S6.4 Observing the result of a 6-bit circuit computation by AFM imaging

Here we describe the imagining protocol (for example wide-field AFM images see Section S7). AFM images for all data analysed for the complete 6-bit IBC tile set were taken using Tapping Mode in fluid on a Bruker Dimension FastScan AFM. Other images were taken on the same instrument or with a Nanoscope III Multimode AFM (Veeco Metrology Group, now Bruker AXS).

S6.4.1 AFM imaging protocol

Mica pucks were prepared by cleaning a (previously used) steel puck with isopropanol and a razor blade, drying with Kimwipe, applying a tiny drop of 5 minute epoxy to one side, pushing a thin plastic (or teflon) disk (of roughly 12 mm diameter) down with one's thumb on the epoxy until it has spread evenly across the area of the plastic disk. After 5 minutes, a 10 mm diameter mica disk was similarly glued to the top side of the plastic disk.

Approximately 10 μl of either the growth buffer (1× Tris-Acetate-EDTA (TAE), 12.5 mM Mg^{++}), or a sodium buffer (1× TAE, 150 mM NaCL, 12.5 mM Mg^{++}), was placed on a freshly cleaved mica (supplier: Ted Pella, catalogue #50, Highest Grade V1 AFM Mica Discs, 10mm), and then 5–15 μl of sample in growth buffer was placed on top. The choice of sample volume placed on mica was influenced by how dense the structures appeared to be on a given day; variability in density can be caused by a number of factors including variability in stickiness for DNA nanostructures between different pieces of mica, or even different mica cleaves, concentration and variations in ribbon length (longer ribbons are more likely than shorter ribbons to overlap each other when they hit the surface — the goal is to have a good enough density to take a sufficient amount of data in reasonable time, but without having too many overlapping nanostructures). Where insufficient density of structures was observed more sample volume was added to the mica. Streptavidin protein was added as described below in Section S6.4.2.

[44]Since these procedures worked well (i.e. we did not observe unintended/spurious nucleation, or widespread algorithmic growth errors, due to sample being cooled) and were straightforward enough, we did not carry out further extensive tests to find the degree to which these exact procedures and techniques were necessary.

AFM images on the Bruker FastScan were taken in buffer (i.e. in fluid) using tapping mode with Bruker FastScan-D or FastScan-DX probes (silicon tip on silicon nitride cantilever with approx. 5 nm tip radius) always with a drive frequency of 110 Hz. Typically, large-scan AFM images were taken by setting the scan size to be 8 μm $\times$ 8 μm, with 4096 lines and 4096 samples/line, and a scan rate of 1.96 Hz. Other settings varied depending on conditions, but example setting were: integral gain 1.0 to 1.5, proportional gain 5.0, Z-range 2.5 μm, amplitude setpoint 95.0 mV, drive amplitude 675.0 mV. Obtaining clear images generally involved starting with (software default) amplitude setpoint and drive amplitude lower than those example values and then increasing to be within a factor 0.2 of those values, and then trading these parameters off against each other until both streptavidin labels and DNA nanostructures were being well-imaged. For smaller scan sizes, other values were used (fewer samples/line and higher scan rate). It was found that, depending on their orientation, streptavidin-labelled ribbons image a little differently: long-axis vertically-orientated ribbons image better when tapped slightly softer than long-axis horizontally-orientated ribbons. When optimising parameters, best streptavidin imaging results are found by choosing a "happy medium" between these two.

S6.4.2 Streptavidin labelling of DNA nanostructures

Streptavidin protein stock at 5 μM was prepared as described in Section S6.1. Streptavidin protein stock at 5 μM was added to the solution on the mica puck in increments of 1.5 μl. After adding each increment the mica surface was imaged for at least 15 minutes during which time the density of streptavidin labelling would increase as would the density of streptavidin 'background' visible on the mica. If the labelling streptavidin efficiency on the DNA ribbons was deemed too low, we increment. Good labelling efficiency was found to be anywhere in the range of 3 to 9 μl of added protein. Out of several procedures tried this was found to be optimal. (In our hands, if we ramped up the streptavidin concentration too quickly, the AFM tip was more likely to quickly pick up a streptavidin molecule, thus ruining the images (and the tip), or other forms of strange imaging conditions would occur.) Various procedures were tried to ramp up streptavidin concentration while washing streptavidin off the mica surface; including use of various temperatures, incubation times, mica washing steps, with either/both of Sodium-Magnesium or Magnesium buffers; frustratingly most attempts gave substantially worse results than the protocol described above; see Sections S5.5.2 and S7.4.1 for more details.

S7 Analysis of AFM images

S7.1 Preprocessing of AFM images

The analysed AFM data set for the complete 6-bit IBC tile set (main paper, Section S7, Section S8) was produced via large-scan (8 μm $\times$ 8 μm, or larger) images taken using a Bruker Dimension FastScan AFM. Minimal image processing was carried out on these AFM images. Specifically the following 3 steps were performed using the open-source software Gwyddion [106] (URL: http://gwyddion.net/, version 2.42, released 2015-10-06): remove polynomial background (degree 3, typically), Align Rows (via Median or Median of Differences), Color Range (Automatic color range with tails cut off). However, for images with smaller scan size (in the region of 1 μm $\times$ 1 μm) polynomial background removal was often omitted. Notably, scar removal was not carried out on these images as it was found that besides removing 'scar' imaging artifacts it could sometimes also change the appearance of streptavidin-labelled nanotubes. Files were exported to .tif format with no scale bars, no lateral scale and without any additional frame/border pixels (scale bars were added later using custom Python code).

The AFM images in Section S5 were taken using a Multimode AFM. DNA nanostructures were masked out, the image underwent polynomial background removal (degree 2 to 6), then masks were removed, and the above-mentioned Align Rows and Color Range adjustment steps were carried out. For such Multimode images, scar removal was carried out if did not overtly affect how the streptavidins appeared on the DNA nanostructures.

S7.2 Annotating of objects in AFM images

AFM image analysis was carried out by hand and with the aid of a custom software pipeline as follows.

First, nanotubes and various nanotube features were graphically annotated with help from the program Sloth[45]. Almost all of the 12,527 nanotubes analysed were graphically annotated by hand. AFM images were opened in the Sloth Graphical User Interface tool (version 1.0, released 2013-11-29) shown in Figure S37. A tight bounding box was manually drawn around each nanotube. In addition, a few hundred were automatically annotated using Python code written by Tristan Stérin. For randomised circuits, features of interest were marked by hand, as explained in the relevant parts of Section S7. Graphical annotations were exported from Sloth as annotation text files in the human-readable open-standard JSON format.

The following annotation rules were applied uniformly on all ribbons while working through the Sloth interface:

1. The bounding box was an approximate rectangle with exactly four corners drawn as follows: the first point was placed at the bottom-left corner of the origami seed (assuming we are reading the seed label "right side up"), then the other three corners were placed anti-clockwise around the entire object (origami seed and SST ribbon), so as to give a tight bounding box. In the case of a rolled or curved ribbon, the box was drawn so as to approximately minimize the vertical height. Then, the length of a ribbon is defined as the length from the bottom-left corner to the bottom-right corner, i.e. the distance from the first to the second point.

2. Ribbons that showed little or no tile growth, which we term "insignificant growth", were labeled as such. Some examples are shown in Figure S40(b). Typically, this meant that any visible tile growth from the origami seed was no more than a triangle whose base was at the perceived location of the adapters, with opposite apex at the end of such growth. However, since it can be difficult to precisely estimate adapter location on a given ribbon we tried to be conservative in this estimate, in the following senses. Cases where growth continued a few layers longer than such a triangle were marked as having no significant growth. For circuits such as Sorting or Parity, where only a few layers of growth are necessary for the correct output to appear, the distinction between significant and insignificant growth was typically based on whether enough layers had attached for the correct output to be plainly visible (e.g., all bits were sorted). To get an idea of whether we were actually conservative we can look at predicted seed length versus measured insignificant growth length, with an assumed DNA base length

[45]Sloth source code: https://github.com/cvhciKIT/sloth.
Sloth documentation: http://sloth.readthedocs.io/en/latest/index.html.

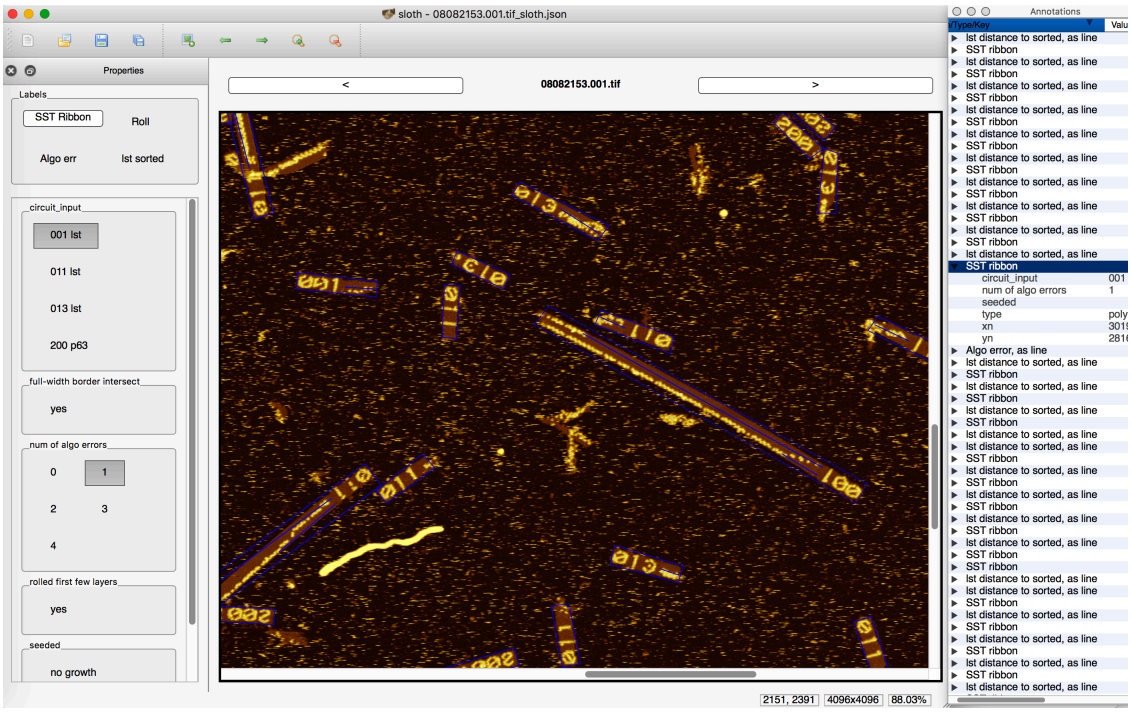


Figure S37: Screenshot of the Sloth annotation tool, customised with buttons and features for annotating nanotubes. Annotations for a small portion of Supplementary Image 1 is shown, which in turn is referenced in Section S7.3.

of 0.34 nm. Calculating origami seed length from the design (Figure S11, and including the part of the adapters bound to the scaffold) gives a predicted seed length of 147 nm. The distribution of calculated lengths for all ribbons labeled as seeds with insignificant growth had mean 155.6 nm and standard deviation 12.24. This suggests that our labelling strategy for ribbons having insignificant growth was indeed conservative, on average.

3. Tile attachment errors were annotated for 15 of the 21 circuits, specifically for those where it was deemed straightforward to detect by eye.[46] Some examples of tile attachment errors are shown in Figure S40(a). A tile attachment error is defined as an unintended bit-flip, visualised as the presence or absence of a streptavidin molecule that respectively should not, or should, be present on a correct run of the computation. Interpretation of whether the presence or absence of a streptavidin is a tile attachment error is not always straightforward due to streptavidin-labelling errors in our dataset (despite our best efforts). Strategies to distinguish tile attachment errors from mere streptavidin labelling errors are discussed in detail in Section S7.4.1. Our definition of tile attachment error has other potential biases: Two tile attachment errors could occur in a single layer, that give the appearance of only a single bit flip, leading us to undercount tile attachment errors. On the other hand a single tile attachment error can happen that gives the impression of two bit flips (since some tiles output two bits), leading us to overcount tile attachment errors. However, the overall tile attachment error rate is so low (0.03 %) that we expect these to be very infrequent. Apparent tile attachment errors within the last one or two circuit-layer of the ribbon (15–30 nm) were considered unreliable and were not counted (since one sometimes see spurious attachment of streptavidin protein or DNA to the growth end, or sides, of ribbons).

   For some circuits, tile attachment errors are easy to spot (e.g. Parity, Copy), for others it was harder (e.g. Palindrome; in which case we simply classified whether there was a tile attachment error that flipped the circuit's decision). Each of Sections S8.1–S8.21 contains details about whether tile attachment errors were recorded, and if so examples are given and, in some cases, additional

[46] The six circuits for which we did not attempt to identify tile attachment errors were: DiamondsAreForever, Rule110Random, 2Eggs, Drumlins, Rule30, and Cycle63.

explanation.

4. Some ribbons were annotated as "rolled". This annotation was uniformly applied to four different types of ribbons, those that:
   (a) appeared to be still rolled up into a tube due to incomplete unzipping (Fig. S38(a))
   (b) partially rolled upon mica deposition (possibly due to having long length) (Fig. S38(b))
   (c) those that were deposited across the top of another ribbon, usually resulting in rolling (Fig. S38(c))
   (d) those that were flat, but the AFM scan experienced a "horizontal shift" imaging artifact (Fig. S38(d))

5. If a ribbon's seed label was visible in the AFM scan window, but the other end of the ribbon was beyond the edge of the AFM scan window, this was marked. In this case, the bounding box was drawn as a possibly non-rectangular polygon capturing as much of the ribbon as possible without going off the edge of the image. The length was then defined based on the four corners of the polygon as described above.

6. A number of circuit-specific properties were analysed. In particular for many of the randomised circuits we collected annotation data for aspects of the computation that we expect to be affected by gate probabilities (i.e., tile concentrations). These include Waves, FairCoin, DiamondsAreForever, LazySorting, LazyParity, and AbsorbingRandomWalkingBit. Annotation typically involved recording the position and type of some specific sets of features per circuit. See Section S8 for examples of such features (e.g. distances to create waves and crash waves in Waves, decision type and distance to decisions for FairCoin, distance to create a diamond in DiamondsAreForever, distance to absorption for AbsorbingRandomWalkingBit etc.).

Section S8 makes use of these annotations.

We did not graphically annotate ribbons where the seed was illegible (examples shown in Figure S38(e)) or where there was no seed attached to the ribbon (Figure S40(b)). Any obscured part of the ribbons (where we can not see streptavidins) were annotated and measured; specifically the distance along which the tube was not fully unzipped (Figure S38(a)), or when the tube rolled during mica deposition (Figure S38(b)), or overlapping tubes on mica where the overlap obscured streptavidins (Figure S38(c)), or where AFM scanning caused part of the tube to shift on the mica surface (Figure S38(d)).

Custom Python code was then used to carry out two kinds of analysis on those text-based (JSON format) annotations. The first step was to use annotated image coordinate information to extract small rectangular images, each containing a single nanotube in order to display the AFM data in Figures 2–4 of the main text, as well as in this section and Section S8. The second step was to use the annotation text files as input to code that computes the various statistics reported throughout the paper, including: estimates for number of tiles attached, number of tile attachment errors, fraction of seeds with significant growth, as well as other circuit-specific properties for many of the 21 implemented circuits reported in Section S8.

S7.3 Example wide-field AFM images

Supplementary Information C, provided as a separate file because of its large size, contains representative AFM images from our dataset at a scan size of 8 μm $\times$ 8 μm and resolution of 4096 $\times$ 4096 pixels. (The images in Supplementary Information C have been JPEG compressed, which results in some smoothing; raw AFM files and PNG files for all images used to analyse the 21 circuits implemented with the complete 6-bit IBC tile set are also available from the authors' web site.) All data reported for the complete 6-bit IBC tile set was taken with those scan area and resolution parameters (except for some Parity data taken at a larger image size and similar resolution), and using the streptavidin-labelling protocol described in Section S6.4.

Supplementary Image 1 shows the circuits LazySorting with barcodes 001, 011 & 013, and Cycle63 with barcode 200. Supplementary Image 2 shows the circuits FairCoin with barcodes 130, 131, 132, 133 & 134, and Sorting with barcodes 001, 011 & 013. Supplementary Image 3 shows the circuits Rule110 with barcodes 110 & 114, and Parity with barcodes 001, 002, 003 & 004. The bottom part of the image shows some streaking due to AFM drift (we included this image to give an example of streaky streptavidin labels seen in some of the data). Supplementary Image 4 shows the circuits MultipleOf3 with barcodes 230, 231,

232 & 233, PALINDROME with barcodes 210, 211, 212 & 213, COPY with barcode 020, and CYCLE63 with barcode 200.

In the images one can see the intended computations, and some of the imaging nonidealities and growth problems described in Sections S7.4 and S7.5. One can also see a large number of streptavidin molecules as "streaky spots" on the mica background, as well as a few unexplained bright (tall) blobs and AFM scar lines. Images were minimally processed so as not to alter the data, as described in Section S7.1.

Section S8 contains a detailed analysis of results for all 21 implemented circuits.

S7.4 Imaging problems

This section describes non-idealities encountered in imaging of DNA nanotubes. They are summarized in Figure S38. None of these necessarily implies a problem with the actual experiment of algorithmic self-assembly of DNA tiles. Rather, they disrupted our ability to assess the results of the experiment.

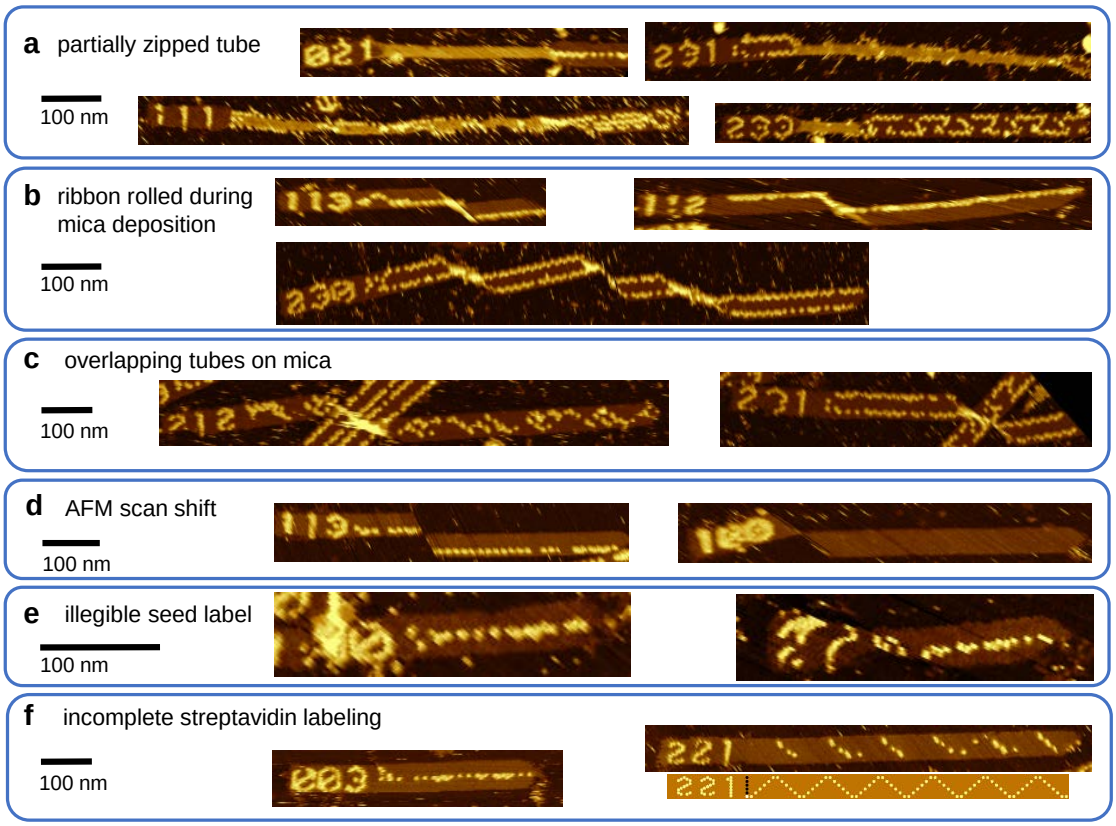


Figure S38: Nonidealities in imaging.

a) partially zipped tube. Unzipping required several hours, and unzipping for even a full day did not always result in completely unzipped nanotubes. Most (though not all) partially zipped nanotubes have a single contiguous zipped region near the seed, consistent with the hypothesis that unzipping strands initiate on the end of the nanotube and proceed towards the seed.

b) ribbon rolled during mica deposition. Sufficiently long nanotubes had a tendency to partially roll upon mica deposition, resulting in a portion that appears thinner and at an angle to the rest of the ribbon.

c) overlapping tubes on mica. Some pairs of long ribbons land so as to intersect, with the ribbon on top appearing thin, similar in appearance to ribbons that partially roll upon deposition (or possibly they

are rolled and there is some correlation between a ribbon becoming rolled intersecting another ribbon upon deposition).

d) AFM scan shift. One artifact of AFM scanning is that part of the ribbon appears shifted relative to the rest; this always occurs parallel to a scan line.

e) illegible seed label. Some seed labels were illegible due to occlusion from an intersecting structure or AFM warping.

f) incomplete streptavidin labeling. Streptavidin labeling of tiles was consistently less complete than that on the seed as discussed in more detail next. Figure S38(f) shows two examples of poor streptavidin labeling for PARITY on input 100101 and ZIG-ZAG on input 000001. Section S7.4.1 discusses streptavidin-labeling in more detail.

S7.4.1 Incomplete streptavidin labeling and its implications for analysing tile attachment errors

Figure S39 gives a detailed analysis of the two main challenges we found in regards to streptavidin labeling of tiled nanotubes.

The first, and most impactful, was low labelling yield as seen in Figure S39(a1), which we addressed by slow addition of small amounts of streptavidin at a time. The protocol is described in Section S6.4.2. This protocol gives the kind of improvement shown in Figure S39(a2).

The second challenge was that we found, by repeated imaging of the same structure (Figure S39(b1), (b2)), that even if a streptavidin molecule is present on one scan, it may not be visible on the next, which on any given scan implies there is some (presumably small) fraction of streptavidins that are not visible to the AFM or actually missing from the nanostructure. We hypothesise that there are two reasons for this:

1. Although a streptavidin molecule is present, the AFM does not "read" its presence on a given scan. The evidence for this hypothesis is threefold. Firstly, in Figure S39(b2) green arrows highlight locations where individual streptavidin molecules are partially obscured on one scan, but otherwise visible on other scans. Hence, we know the streptavidin molecule is present on the nanostructure. Since partial occlusion of a streptavidin can occur, it is a distinct possibility that total occlusion could occur via the same or a similar mechanism. Secondly, in both (b1) and (b2) of Figure S39, white arrows highlight quite a number of instances of a streptavidin being visible, not visible, and visible again on three consecutive scans. Thirdly, from experience we know that even slightly increasing AFM tip force (by decreasing amplitude setpoint or increasing amplitude) will cause streptavidins to disappear only to immediately reappear when the force is reduced. This tells us that visibility of streptavidin molecules is extremely sensitive to scanning conditions.

2. Streptavidin molecules come off nanotubes on mica. In addition to streptavidin molecules binding in higher and higher yield over time Figure S39(a2), we have seen streptavidin molecules be removed from structures for significant periods of time, or seemingly indefinitely. This can happen due to AFM tip interaction, such as when the tip removes a DNA tile with a biotin-streptavidin complex attached (which is a plausible explanation for the small dark patch highlighted by the yellow arrow in Figure S39(a2)). In addition, we see permanent or long-term streptavidin-loss occur (red arrow locations) without any noticeable loss of DNA strands from the tiled structure.

Note that the poorly-labelled data in Figure S39(a1), taken before we had optimised our imaging protocol, was *not* included in our data set (since analysing it for tile attachment errors and other features of interest is challenging and open to interpretation).

When taking data we followed the above mentioned protocol to optimise streptavidin-binding yield. However, yield was consistently lower than on the origami seed. The incomplete labelling therefore poses a challenge when analysing the error rates in algorithmic growth. Thankfully, it is often very clear how to distinguish a missing streptavidin label from an actual tile attachment error that alters circuit function. The basic principle is that for many circuits, a tile attachment error in a given layer is likely to yield a disrupted bit pattern in the next and in subsequent layers, whereas a streptavidin labeling error with correct

underlying computation will yield a correct bit pattern in the next and subsequent layers. A few further thoughts in how we analysed AFM images to determine error rates are discussed below.

- Figure S39(b1,b2) tell us that on **a given scan** some fraction of biotin modified strands will not show a streptavidin in the recorded AFM image. Hence on any given scan we expect some fraction of streptavidin molecules to be missing from 1-bit proofreading blocks, and this is true of even for perfect computations (i.e. correctly performing, with no erroneous tile attachments). That is, we should not be concerned to see some streptavidins missing; and indeed the figure tells us that missing streptavidins **do not immediately imply tile attachment errors**.

- The basic principle for distinguishing streptavidin labeling errors from algorithmic assembly errors can be illustrated with Copy. Consider, as an example, input 010010, and suppose the AFM image shows streptavidins only sporadically, but always in bit positions 2 and 5. In this case, we infer a labelling problem only, because computation errors would not be expected to be confined to just those two positions. On the other hand, if the AFM image shows that further along the ribbon, bit position 3 also starts to be sporadically labelled by streptavidin, then we can infer that a single error occurred.

- For circuits such as Parity, Palindrome, and MultipleOf3, which perform their main computation in just the first few layers and then enter a repeating cycle of states, it is often not possible to use the above principle to verify each bit during the initial computation. However, the correctness of the repeating output pattern is a strong indicator of correctness of the computation, and can easily be verified despite incomplete streptavidin labelling. For example, a single error in any gate of the Parity circuit, at any time, will flip the output and will thus be seen if there are no further (nearby) errors. In some cases, a single bit flip in certain places would not be identified – for example, in the Palindrome circuit with a non-palindromic input, most bit flips will result in a pattern that remains non-palindromic, and thus may not be easily detected. However, for palindromic input, a single bit flip would yield a non-palindromic state, causing the computation to enter a very different (and easily detectible) repeating cycle of states. Therefore, in this case our tally of (detectible) tile attachment errors will be an underestimate – but by less than a factor of two.

- Pattern-forming circuits can usually be similarly analysed based on the expected effect of a bit flip. For example, in the Zig-Zag circuit, the zig's label poorly, the zags label much better – but a bit flip would cause either a disappearance of the pattern or an additional line, which, if not observed, would suggest that the invisible zig is a labelling error rather than a tile attachment error. For circuits that do not create a deterministic pattern, such as RandomWalkingBit, the distinction between a labelling error and a tile attachment error can be evaluated not based on an exact match to a predicted pattern, but rather by consistent features of a pattern that the circuit preserves when functioning correctly. As a case in point, a bit flip in RandomWalkingBit would cause the number of randomly-walking bits to change, e.g. from 1 to 0 or from 1 to 2. Therefore, in RandomWalkingBit images where bits were seldom visible when walking up, but visible when walking down (presumably for the same molecular reasons that cause the same imaging variation in Zig-Zag, which shares many identical DNA tiles), we would not identify an error as having occurred until the number of randomly-walking bits changed (as estimated over a certain distance of the ribbon).

- There were some circuits where it was difficult to apply these principles, and therefore we did not attempt to estimate error rates. Cycle63 is one such circuit.

- On all circuits that go to the fixed point 000000, during AFM imaging a positive control was used, in the form of a deterministic circuit whose eventual cycle contains 1's (e.g., parity on an input with an odd number of 1's, or sorting an input with 1's). This helped to indicate when enough streptavidin had been added that most biotins on tiles had a streptavidin attached, to ensure that the *absence* of streptavidins implies the absence of biotinylated tiles (i.e., 1 bits). (Origami seed labelling was not a reliable positive control, since streptavidins attached much more readily to the biotinylated staple strands on origami than to biotinylated tiles.)

For future work one solution to increase the yield of the streptavidin labels would be to use multiple co-located biotin modifications, as has been done for multi-stranded double-crossover tile motifs [9].

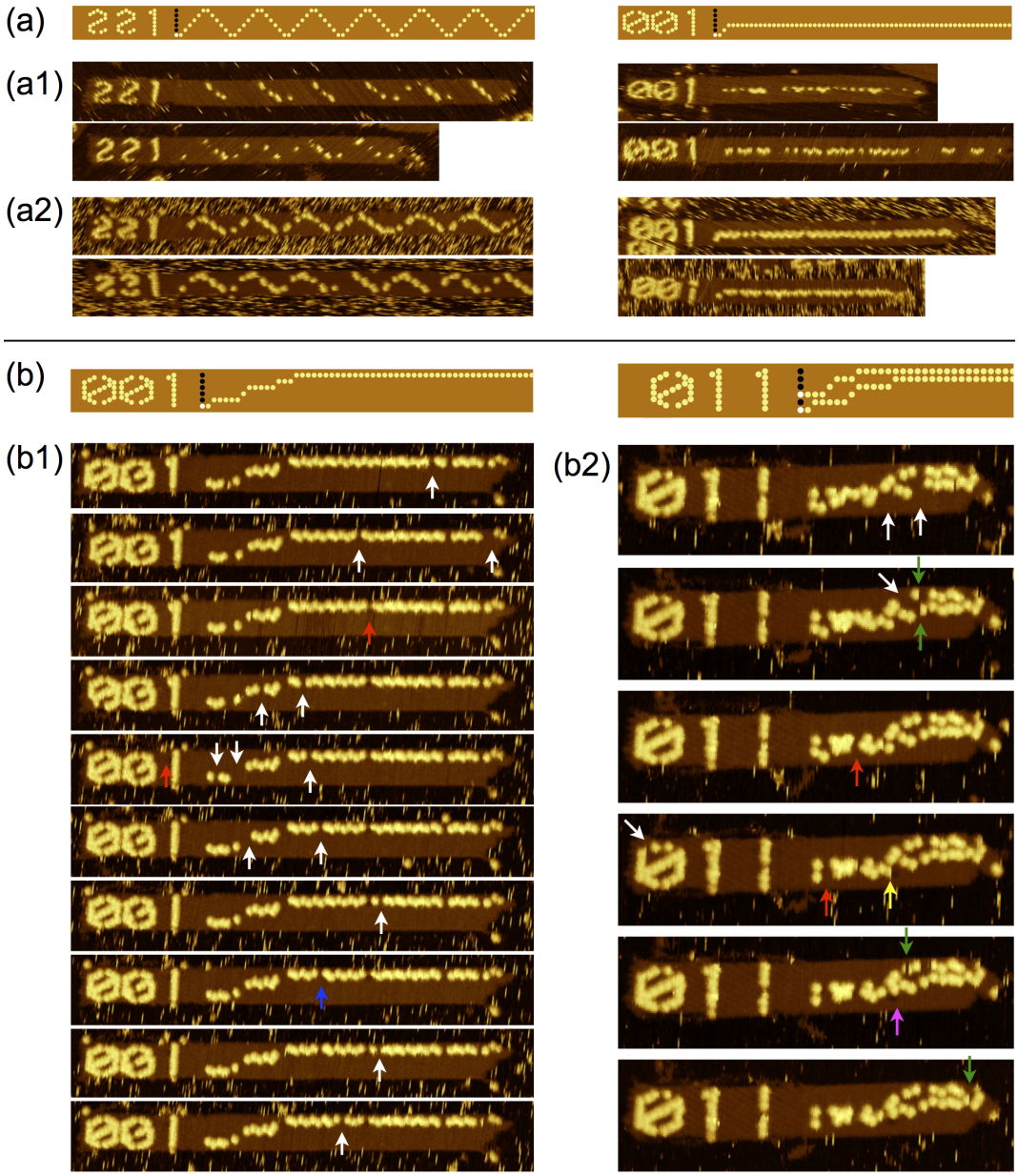


Figure S39: Understanding and addressing the challenge of low-yield streptavidin labelling. (a) Improving labelling yield by slow addition of streptavidin to samples on mica. Left: Zig-Zag with barcode 221 and 6–bit input 000001. Right: Parity with barcode 001 and 6-bit input 000001. (a1) Zig-Zag and Parity imaged after addition of 1.5 μl of 5 μM streptavidin to the $\sim$ 60 μl sample droplet on mica, and (a2) after addition of 10 μl of 5 μM streptavidin, slowly added in small increments of 1.5 to 2 μl at a time of over a period of an hour or more. In (a2) the yield of streptavidin labelling on DNA tiles is significantly increased versus (a1), although is still less than the origami barcode. However, note that the background imaging quality concomitantly decreased, which limited our ability to improve labeling by adding streptavidin. (b) Streptavidins disappearing and reappearing during imaging of LazySorting nanotubes. Simulations show representative computations, and are not intended as exact interpretations of the nanotubes shown below. Data was taken after incremental and slow addition of a total of 6.5 μl of streptavidin at 5 μM to the droplet on the mica puck. (b1) A sequence of successive AFM images, taken about 20 seconds apart, of a single LazySorting nanotube with barcode 001 and input 000001, and (b2) a single LazySorting nanotube with barcode 011 and input 000101. We interpret the image sequences using arrows as follows: Streptavidin molecule is partially obscured for a single frame (green arrows); not visible for a single frame (white arrows); not visible for two frames (blue arrow); not visible for several frames (red arrows); not visible for several frames and DNA strand(s) possibly removed from lattice (yellow arrow); appears at a position where there was none before (pink arrow). Yellow and pink arrows could also be reasonably interpreted as streptavidins moving around. Although each image has at least a few streptavidin molecules missing we infer that there were no algorithmic (tile-attachment) errors in this figure.

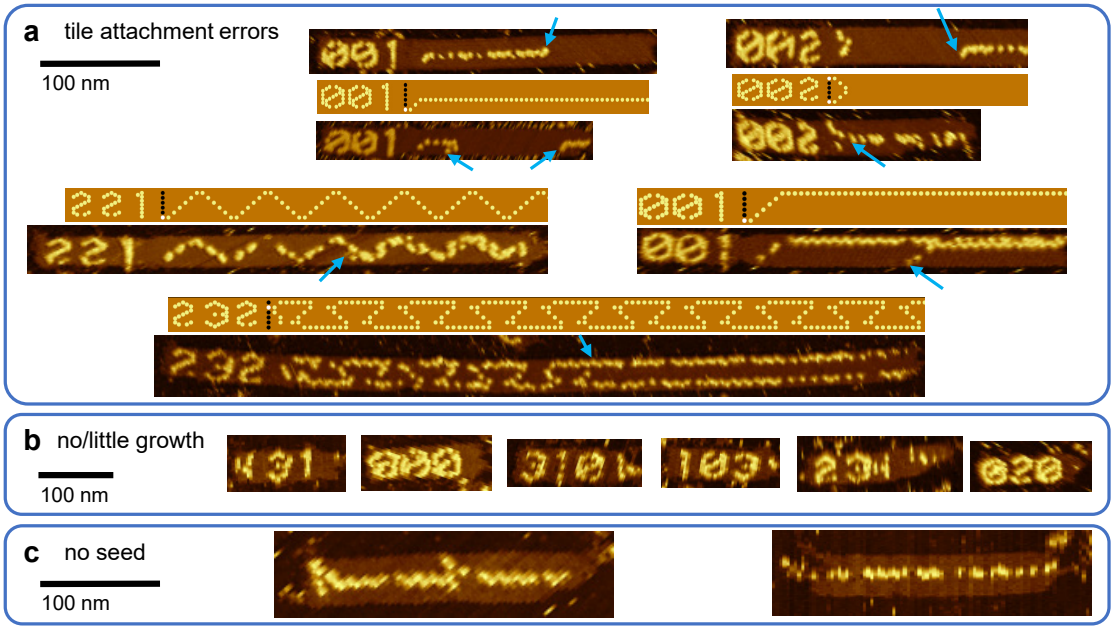


Figure S40: Errors in growth. We report statistics on tile attachment errors and seeds without tile growth. The simulations show correct (intended) behaviour, and blue arrows highlight the occurrence of a tile attachment error.

S7.5 Growth problems

This section describes difficulties encountered in growth of DNA nanotubes. They are summarized in Figure S40. Unlike imaging problems discussed in Section S7.4, these are problems with the actual experiment.

a) tile attachment errors. A *tile attachment error* means the attachment of a tile into the tube that does not match both input glues. For most circuits, a single tile attachment error will result in a predictable change in the subsequent pattern. For instance, in the parity circuit, a stripe of 1's in the middle either appears or disappears when a single tile attachment error occurs.

This effect on the pattern is different for different circuits. Representative samples of errors for each circuit are shown in Section S8. Also, Figure S40(a) shows a few examples of a ribbons with error(s) alongside an error-free simulation. Section S7.4.1 explains how we distinguished tile attachment errors from streptavidin-labelling errors.

b) little or no growth from seed (seeding errors). Almost half (38.9 %) of the DNA origami seeds did not have a significant amount of tile growth. This effect also occurs in seeded self-assembly experiments using the DX tile motif [8, 26]. We hypothesise that the differences in crossover spacing between the DNA origami lattice (32 base pairs between crossovers) and the SST lattice (21 base pairs between crossovers), visible in Figure S41, introduce a kinetic barrier to seeding. The same observation has been made about the DX tile lattice [8, Fig. S10], which alternates crossover spacing of 58 and 16 base pairs. Section S8 reports these rates for each of the 21 implemented circuits. Some examples are shown in Figure S40(b).

c) ribbon with no seed. Some ribbons (tubes unzipped onto the mica) were observed to be unattached to any seed. This was rare, but unlike algorithmic and seeding errors, no statistics were formally collected. There are two relevant points to take from our data. First, most seedless ribbons showed a pattern consistent with at least one of the circuit-input pairs as part of that experiment. Hence it was conjectured that most seedless ribbons were ripped from their seed, for example during pipetting or mica deposition. Second, control experiments were done (on two other sequence sets) in which tiles were held at a good seeded-growth temperature in the absence of any seeds and consistently it was observed that very few nanotubes nucleated on their own when grown at appropriate temperature (see and

Figures S27(a) and S34(rightmost column)). When they did, they tended to be very long, consistent with the hypothesis that there are very few other nucleated structures to compete for tile attachment. This suggests that growth without a seed is very rare, which would be a candidate explanation for the (extremely rare, although not recored in our analysis) occurrences of nanotubes that had no seed and a showed pattern not consistent with some circuit-input pair.

S7.6 Error rate estimates from physical principles

This section considers the observed error rates with respect to estimates predicted by the kinetic Tile Assembly Model (kTAM), as developed in ref. [2] and reviewed in ref. [24]. The kTAM is a more chemically realistic refinement of the aTAM, incorporating details of tile concentrations, attachment/detachment rates, and imperfectly specific glues (bonds). As a brief summary, the kTAM models the algorithmic growth of a single crystal of abstract tiles, starting from a seed tile or seed structure, under conditions where free monomer tile concentrations remain (effectively) constant. The model uses further simplifying assumptions that allow a simplified set of parameters: Following ref. [24], all tile types have the same concentration $c = \mu_0\, e^{-G_{mc}+\alpha}$ as free monomers (with $\mu_0 = 1$ M, G_{mc} being a unitless representation of the entropic free energy of co-localization at this molecular concentration, and α providing an offset to account for the initiation free energy of DNA hybridization [81]), all matching glues have the same binding energy $G_{se}RT > 0$ where $R = 0.001987$ kcal/mol/K is the gas constant and T is temperature in Kelvin (K), all mismatching glues are not perfectly orthogonal but rather bind with a fraction s of the matching-glue binding energy, all tile attachments occur with the same rate constant k_f at any site in the crystal where one or more matching glues are present (thus allowing mismatches), and any tile within the crystal may detach with rate $k_{r,b} = k_f \mu_0\, e^{-bG_{se}}$ where b indicates the number (and/or fraction) of full glue bond strengths with which the tile is attached to the crystal. Prior experimental studies on DX tiles have shown that while these simplifying assumptions are not precisely correct for those systems, they nonetheless capture important features of the self-assembly process [107, 103]. For DX tiles in solution, $\alpha = 3.0$ has been used [24].

When considering growth near the melting temperature for crystals in which tiles attach by two glues (i.e., the on-rate $k_f c$ is near the off-rate $k_{r,2}$), we can further simplify the model by assuming we are exactly at the melting temperature and that therefore $k_f\, c = k_{r,2}$ and thus $G_{se} = \frac{1}{2}(G_{mc} - \alpha) = -\frac{1}{2}\ln(c/\mu_0)$. This is convenient for comparison to experimental data – the energy of tile binding does not have to be measured because, when the temperature is adjusted to be the melting temperature, that effectively sets the binding energy to be determined by the concentration, which is known.

For this model, ref. [24] derives (c.f. the generalization of eqn (9) based on eqn (8)) an approximation for the error rate (per proofreading block) in a proofreading tile set:

$$P_{\text{error}} \approx m e^{-K(1-s)G_{se}}$$

where m is the maximum number of partially mismatching tiles that compete with the correct tile at a given site, K is the order of proofreading, and s and G_{se} are as described above. For our system, we use $m = 2$ (for tiles with the incorrect first input or second input; note this is only for deterministic circuits, since randomised circuits have $m > 2$), $K = 2$ (because we use 2×2 proofreading), $s = 0.5$ (since during sequence design, mismatched glue interactions were on average roughly half as strong according to our model), and $G_{se} = 8.06$ (because growth occurred near the melting temperature with $c = 10^{-7}$ M). Interestingly, the prediction does not depend on k_f or α.

Using these numbers, we calculate $P_{\text{error}} = 0.00063$ and therefore (since there are four tiles per proofreading block) the per-tile error rate is predicted to be 0.00016, i.e. 0.016 %. Analysis of the AFM images for all circuits and inputs yielded an overall tile-attachment error rate of $0.03\ \% \pm 0.0008$.

To assess the significance of this level of agreement, it is worth revisiting some of the simplifying assumptions and approximations that were used. Equal tile concentrations: Because unpurified DNA strands were used where possible, concentrations of fully intact tiles may well vary by a factor of two from nominal concentrations.[47] Constant tile concentrations: Since the scaffold (and thus seed) concentration was nominally 1 nM, the SST concentration was nominally 100 nM per tile type, from the data 61.1 % of seeds

[47] For simplicity, in this calculation we will consider the nominal concentration to be the targeted concentration, rather than the measured concentration, both of which are acknowledged to be different from the actual concentration.

had ribbons, and of those, the average length was 294.5 nm, i.e. 21 proofreading layers per ribbon, therefore only about 11% of free tiles would have been depleted for the worst case where some gate always has the same inputs and thus the same tile type is always incorporated along the gate's given pair of helices. Equal glue energies for correct attachment: As computed using the function `lattice_binding_spacer()` and shown in Figure S24, the mean energy was -11.99 kcal/mol with max and min within ± 2.17 kcal/mol, i.e. no more than 18 % deviations. (Compare this to the value $-2G_{se}RT = -10.45$ kcal/mol estimated based on the monomer tile concentration, above, for $T = 53$ °C $= 326.15$ K.) Mismatch glue energies for tile attachment errors: As computed and shown in Figure S24, the mean energy was -6.82 kcal/mol with max and min within ± 1.82 kcal/mol, i.e. no more than 27% deviations. (Compare this to the value $-(1+s)G_{se}RT = -7.83$ kcal/mol, as above.) The dependence of error rates on distributions of energies is complicated, but varying either G_{se} or s by the above percentages yields estimates that are within a factor of 10 of the one presented above.

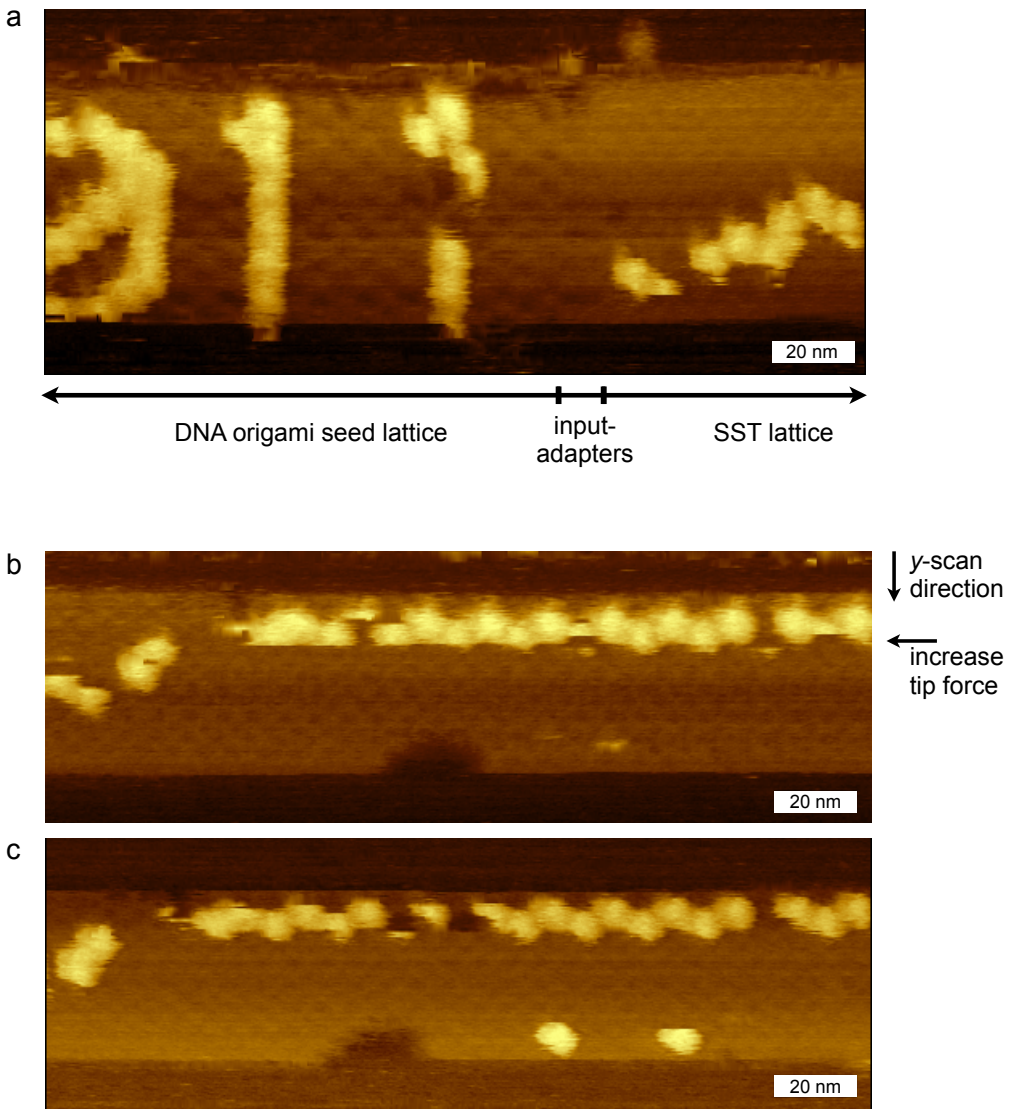


Figure S41: Origami lattice versus SST lattice, shown for 3 zoomed-in images of a LazySorting nanotube with seed 011 and input 000101. For reference, compare to the schematic design of the seed/tiles interface in Figure S16. Also, Figure S10 shows more detail on the SST lattice, including biotin locations. (Presumably there was a tile attachment error, causing one of the 1-bits to be deleted and/or some spurious bindings of streptavidin.) (a) Zoom-in showing DNA origami lattice, putative location of input-adapters, and SST lattice. As expected from frequency of crossovers per unit distance, the SST lattice appears tighter than the origami lattice. (b) Subsequent scan, further to the right along the same nanotube, showing SST lattice and streptavidins. During the scan (with x-direction scan and rescan lines running left-right, and then y-direction running form top to bottom) the force on the AFM tip was increased in order to give a clearer look at the SST lattice. Increasing the force also has the effect of appear to 'slice' the streptavidin molecules. (c) Subsequent scan, with less tip force showing clearer resolution of streptavidin molecules, but less clear resolution of SST lattice. There is a large hole at the bottom-center of the SST lattice, possibly caused by previous scans (not shown), as well as a few possible new holes that are visible in (c) but not (b). Images taken on a Nanoscope III Multimode AFM (Veeco Metrology Group, now Bruker AXS). All scale bars are 20 nm.

S8 Circuits implemented: design, results, testing, analysis

This section gives details on each of the 21 implemented 6-bit 1-layer IBCs. In Sections S8.1– S8.21, for each circuits, we give a circuit diagram, intuition behind the design, computer simulations, typical AFM data, example errors and a performance analysis. Before that, we begin by outlining our process for designing new circuits and an aggregate performance analysis of all implemented IBCs.

S8.0.1 How we designed the 21 IBCs.

Most IBCs were designed by hand with some being designed with the aid of computer programs. The design by hand process generally involves choosing some problem one would like to solve, imagining how the data should flow through the circuit to achieve the desired output, and then finding appropriate circuit layer (gates, and possibility probabilities) to achieve that behaviour. Sometimes we found techniques that could be used to program large sets of circuits; both by hand and by computer search. The following circuits were designed by hand: Copy, Sorting, LazySorting, Parity, LazyParity, Zig-Zag, DiamondsAreForever, RandomWalkingBit, LeaderElection, AbsorbingRandomWalkingBit, Rule110, Rule110Random, FairCoin, Waves and Drumlins. The circuits MultipleOf3, Palindrome and Cycle63 were designed using computer search, specifically a stochastic search algorithm. Finally, some circuits were sought for tile coverage, that is, to ensure we had used every designed DNA tile strand in at least one experiment. Those were 2Eggs and Recognise21 which were both designed by hand and Rule30 via systematic computer search of elementary CA whose local update function $F(x, y, z)$ can be expressed as $f(g(x, y), h(y, z))$ as was used in the proof of Theorem S1.1.

S8.0.2 Aggregate validation and analysis for all 21 circuits implemented.

We ran enough different kinds of circuits, and inputs, so that all 355 designed DNA SSTs in the complete 6-bit IBC tile set were used in at least one experiment (computation). In addition, for one of the circuits (Parity) we ran the circuit on all 64 inputs. Finally, for 15 of the 21 circuits we ran those circuits on a set of inputs that ensured that every tile that could be used for that circuit was used, in some computation. More precisely for the following circuits: Sorting, FairCoin, Cycle63, LazySorting, LazyParity, MultipleOf3, Palindrome, LeaderElection, Recognise21, Copy, and RandomWalkingBit all tiles that were pipetted into the test tube for an experiment were incorporated into correctly grown nanotubes. For some circuits it is actually impossible to use all tiles, in other words even if we ran all 64 6-bit inputs some tiles would never be incorporated into correct growth due to the logic of the circuit. The following circuits had that property, but were tested on enough inputs to properly incorporate all tiles that could be incorporated: Parity, Rule110, Rule110Random, and Rule30. Some of these 15 circuits are randomised; for those we assume that a large enough set of trajectories was visited to incorporate all tiles that could be incorporated.

For the entire set of 21 circuits a total of 129 unique computations were run, i.e. there were 129 unique (circuit, 6-bit input) pairs in the experimental dataset. Conventions used for annotating objects in the dataset are given in Section S7.2.

We now define some statistical conventions used to analyse data throughout the paper. The *sample standard deviation* for n independent samples is defined as $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$. where x_i is the value of the i'th sample and $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the *sample mean*. The *standard error of the mean* is defined as $\sigma_{\overline{x}} = \frac{\sigma}{\sqrt{n}}$. To denote both the mean $\overline{x}$ and standard error of the mean $\sigma_{\overline{x}}$, we write $\overline{x} \pm \sigma_{\overline{x}}$. Most units are placed after the expression, e.g. 60.1 ± 3.4 nm. However, to avoid ambiguity between percents between 0 and 100, and fractions between 0 and 1, percent units are expressed as, for example, $0.3\ \% \pm 0.08$. This means "0.3 percent (the fraction 0.003), plus or minus a standard error of 0.08 percent (the fraction 0.0008)".

Out of a total of 12,527 nanotubes analysed over the entire dataset, we classified 4,869 as having either no or very little tile growth, giving a well-seeded percentage of 61.1 %. One hypothesis for some seeds not showing significant growth is an energetic barrier to growth of single-stranded tiles from a DNA origami seed possibly due to the fact that DNA origami and single-stranded tiles have different crossover patterns and hence different geometric arrangement of helices. Such a barrier to seed-nucleated nanotube growth has been reported by Mohammed and Schulman [26] (for a different origami seed and tile design than ours).

Those 4,869 nanotubes classified as having no or little tile growth had a mean length of 155.6 nm. This value is an *estimate* of origami seed length, albeit an over-estimate as it possibly includes input-adapter strands along with a small amount of tile growth. Another way to obtain a seed length is to *calculate* it from our design: assuming a length of 0.34 nm per DNA base (double-stranded) the design suggests an estimate of 147 nm for the double-stranded parts of the seed and input-adapter strands (excluding any single-stranded regions). This calculated value is not far from our estimate from the data. We used the estimated value of 155.6 nm when computing estimates for number of tile attachments and tile attachment error rate (when subtracting estimated seed length from estimated nanotube length), in order to be conservative.

We estimated a total of 5,053,656 tile attachments for the 21 circuits implemented, assuming a DNA base pair length of 0.34 nm. Tile-attachment error rates were analysed for 15 of the 21 circuits. We observed 1,419 tile attachment errors errors out of an estimated 4,600,351 tile attachments for those 15 circuits, giving a mean and standard error of the mean for the tile-attachment error rate of $0.0308\ \% \pm 0.0008$. Because tile attachment errors may not be independent due to correlations within an individual ribbon, in all tile attachment error rate calculations in the paper, for the standard error of the mean calculation we used a bootstrapping method as follows. Suppose that for a total of m ribbons, each ribbon r has an observed number e_r of errors and an estimated number t_r of tile attachments. We do the following 1,000 times to generate 1,000 "random subpopulation" error rates: Sample $m/2$ ribbons, chosen uniformly with replacement from the set of all ribbons, calling the resulting set S. Calculate the error rate $e_S = \frac{\sum_{r \in S} e_r}{\sum_{r \in S} t_r}$ of S. The standard error of the mean is then estimated as $\frac{\sigma}{\sqrt{2}}$ where σ is the standard deviation of the distribution of e_S over the set of 1000 subpopulations. (The $\sqrt{2}$ factor comes from the scaling of the standard error for $m/2$ samples vs the full m samples.) This technique of bootstrapping was only used for calculating standard error for tile attachment errors, not for other measured statistics.

The six circuits for which we did not analyse error rates were circuits for which we deemed it not straightforward to unambiguously spot a tile attachment error (namely DiamondsAreForever, Rule110Random, 2Eggs, Drumlins, Rule30, and Cycle63).

For analysis of all circuits we needed a value for the length in nanometers of a DNA SST implementation of a single IBC layer. Based on the value of 0.34 nm/bp for double-stranded DNA, we obtain a theoretical estimate as follows. From our design, a single IBC layer corresponds of a horizontal length of two SSTs, which corresponds to 14.28 nm = 0.34 nm/bp × 42 bp. Thus we assume throughout that the length of one IBC layer is 14.28 nm. We used the FairCoin circuit to obtain an independent experimental estimate of the length of an IBC layer (not used elsewhere in analysis, however). The estimation works as follows. For the FairCoin circuit (see Section S8.5) we manually measured the length of each distance to a decision along with reading the streptavidin pattern (of bits) to obtain the integer number of coin-toss-pairs to each decisions. The mean length for a *pair* of coin tosses (two IBC layers) was 30.6 nm, giving 15.3 nm per IBC circuit layer, within 7 % of the theoretical prediction of 14.28 nm.

For some randomised circuits, annotation data was collected that we expect to be affected by gate probabilities (i.e., tile concentrations). These include Waves, FairCoin, DiamondsAreForever, LazySorting, LazyParity, and AbsorbingRandomWalkingBit. In each of these cases, we discuss a theoretical model of the expected measurements based on probability theory for comparison with measured results. Some measurements agree closely, and others quantitatively deviate a little. All measurements agree "qualitatively" in the sense that, if one expects a measured quantity to change in one direction as concentrations change (e.g., the number of iterations until a certain tile first binds should increase if its concentration is decreased), then the quantity does in fact move in that direction. For the randomised circuits Rule110Random and RandomWalkingBit relevant data was not collected, and for LeaderElection the analysis is not straightforward, so we omit a formal analysis of the randomised behavior of these circuits.

S8.0.3 Other comments and observations on the dataset.

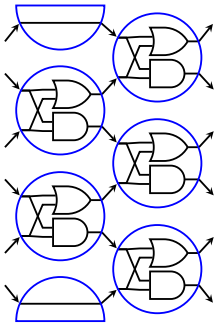
It should be noted that input-adapter strands did not have biotin modifications (labels), which means that in AFM images of nanotubes the input bits are not visible. In the simulations, input bits are denoted in black (0) and white (1).

The scale of the images in this section can be see from the barcode written on the origami seed. For scale, each origami seed was measured to be 155.6 nm. Scale bars are shown in figures in the main text.

S8.1 Circuit: SORTING

Intuition and motivation. This circuit sorts its input bits, so that the 1's end up on the top and the 0's end up on the bottom. Specifically, the SORTING circuit performs an Odd-Even Transposition Sort [51]. Each 2-input gate computes the same function: if there is a 1 on bottom and a 0 on top (i.e., the bits are out of order), swap them, otherwise copy.

SORTING computes a function, as defined in Section S1.1.1.



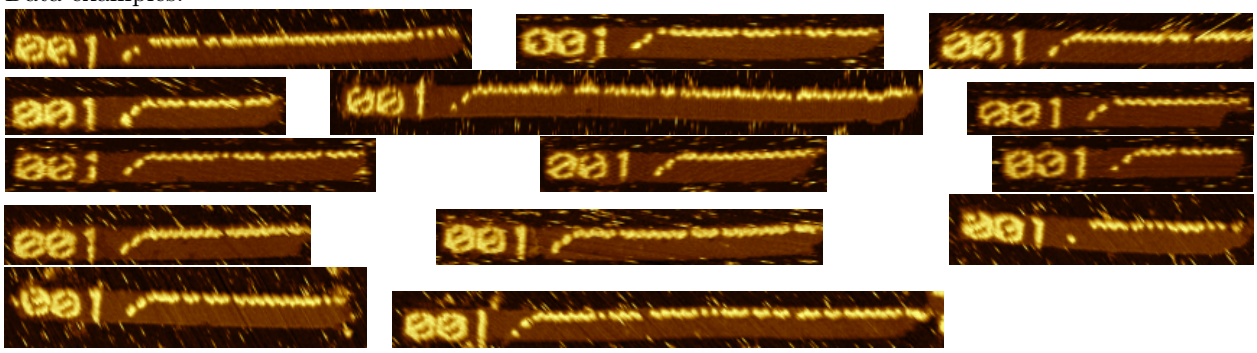
SORTING circuit

Validation and statistics. 36.7 % of seeds had significant growth (477 seeds with significant growth, 822 seeds without). The measured total amount of tile growth for all SORTING nanotubes was 120,054 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 269,028 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 77 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.03\ \% \pm 0.003$.

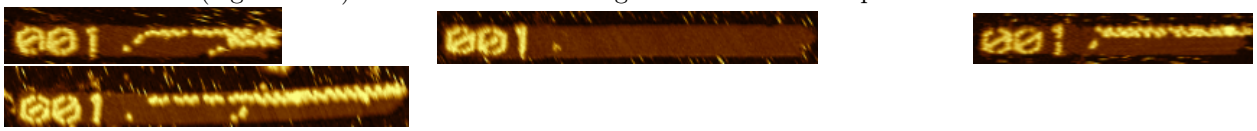
The 4 inputs shown below represent a full coverage test for the SORTING circuit in the sense that correct computation on these inputs requires incorporation of every tile type for the SORTING circuit.

Seed: 001, input: 000001, output: 100000, simulation: 

Data examples:

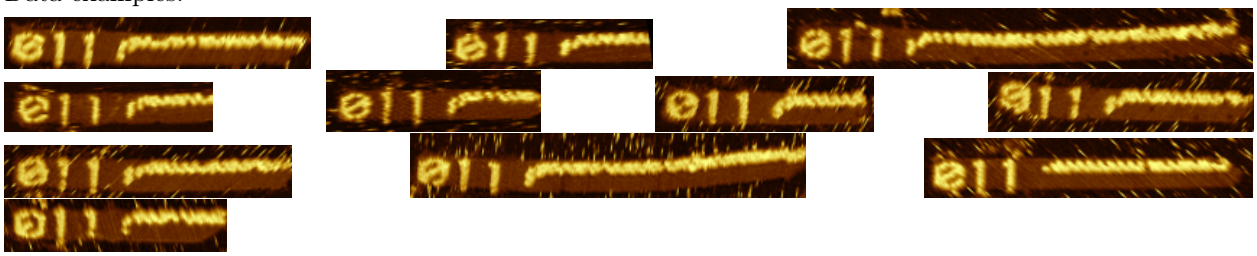


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:

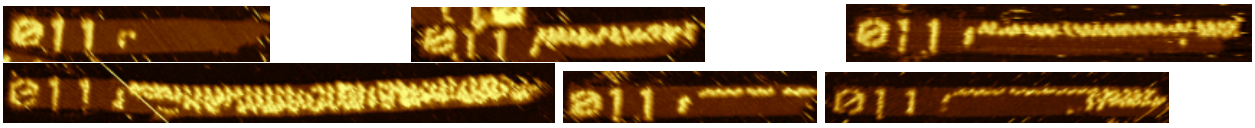


Seed: 011, input: 000101, output: 110000, simulation: 

Data examples:

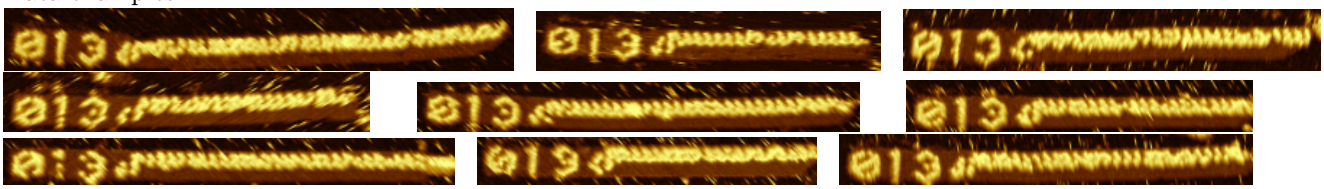


Tile attachment (algorithmic) error rate: 0.07 %. Algorithmic error examples:

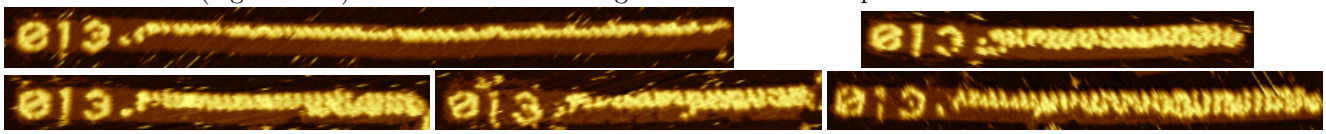


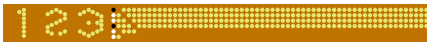
Seed: 013, input: 000111, output: 111000, simulation: 

Data examples:

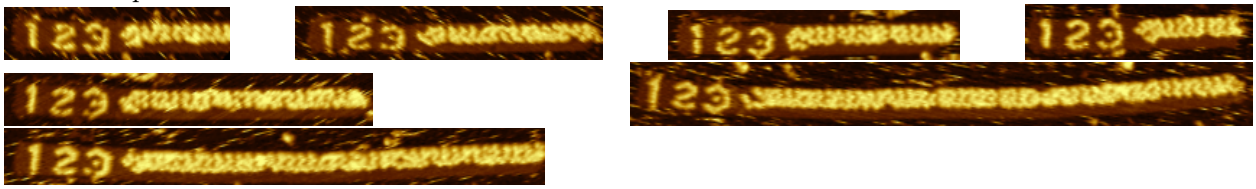


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:



Seed: 123, input: 011011, output: 111100, simulation: 

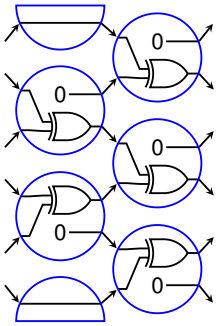
Data examples:



Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error example:

S8.2 Circuit: Parity

Intuition and motivation. This Parity circuit determines whether the number of 1's in the input is even or odd. Gates above the middle move 1's down (i.e., swapping the bits if a 1 is on top and a 0 is on bottom), while gates below the middle move 1's up. When two 1's meet, they cancel (i.e., $g(1,1) = (0,0)$ for all gates g).



Parity circuit

The parity decision problem, on input words of any length $n \in \mathbb{N}$ is solvable by polynomial size, logarithmic depth (in n) Boolean circuits, and is provably impossible to solve on polynomial-size constant-depth Boolean circuits even when the gates may have arbitrary fan-in [49, 20]. It is easy to generalise our 6-bit 1-layer Parity IBC to an n-bit 1-layer IBC for any $n \in \{2, 4, 6, \ldots\}$ giving a family of IBCs that decide any instance of the parity problem (for odd n simply pad the input with an extra 0). This shows the power of iteration in the IBC model to "simulate circuit depth" in order to overcome the above-cited limitation of constant-depth polynomial size Boolean circuits. We found the Parity circuit soon after defining the model, and it gave us motivation to more carefully investigate the computational power of the IBC model, especially the n-bit 1-layer IBC model.

Parity decides a language, as defined in Section S1.1.1.

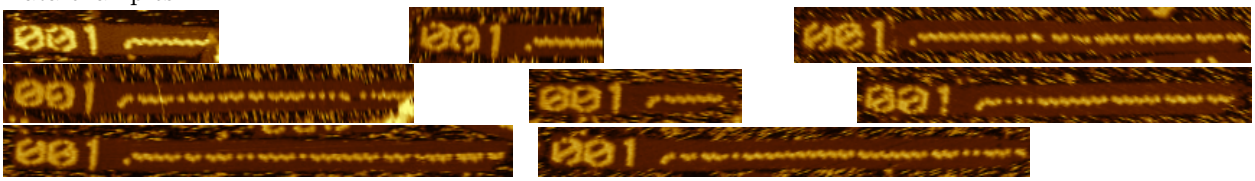
Validation and statistics. 60.5 % of seeds had significant growth (2,729 seeds with significant growth, 1,785 seeds without). The measured total amount of tile growth for all Parity nanotubes was 588,047 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 1,317,753 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 390 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.03\ \% \pm 0.001$.

The 6 inputs shown below represent a full coverage test for the Parity circuit in the sense that correct computation on all of those inputs requires incorporation of every tile type that would be incorperated on all of the possible 64 inputs.

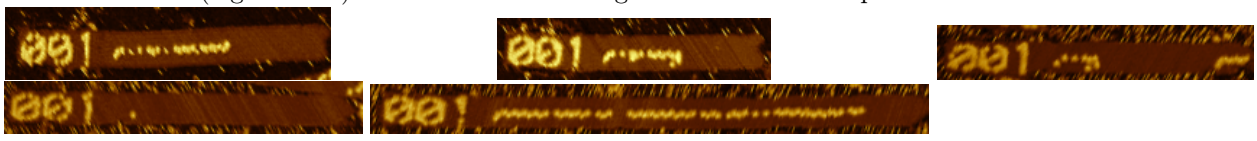
The Parity circuit was also tested on all 64 inputs; one nanotube for each input is shown in Figure S44. In the case of inputs for which the output is 0, a positive control whose output is 1 (input 010000, barcode 111) was simultaneously imaged. This ensured that streptavidin labeling was reliable, in order to give confidence that ribbons lacking streptavidins also lack biotins (i.e., they got the correct answer of 0).

Seed: 001, input: 000001, output: yes, simulation: 

Data examples:

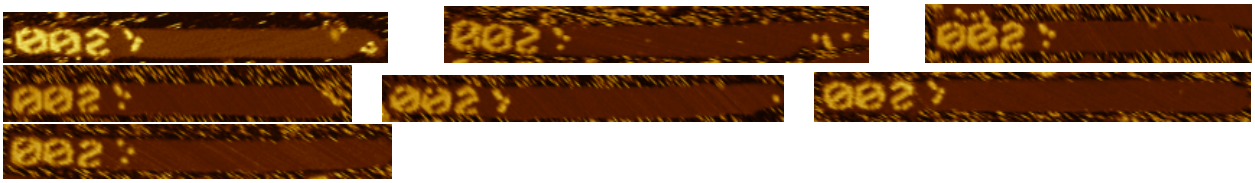


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:

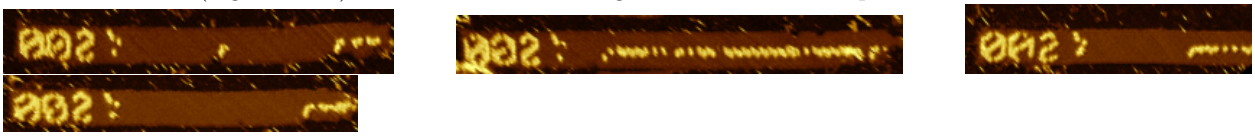


Seed: 002, input: 100001, output: no, simulation: 

Data examples:

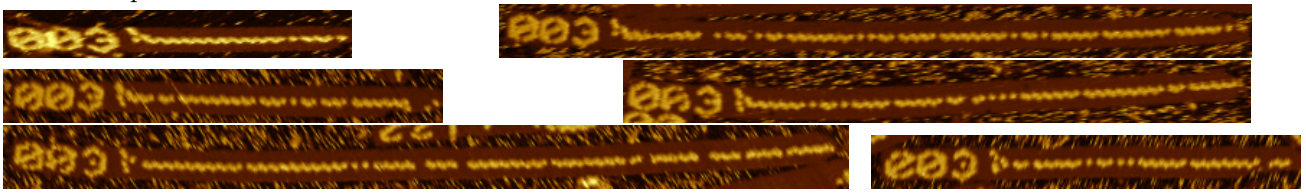


Tile attachment (algorithmic) error rate: 0.06 %. Algorithmic error examples:



Seed: 003, input: 100101, output: yes, simulation: 

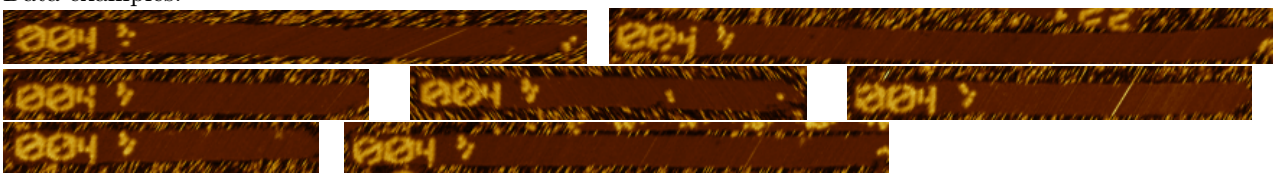
Data examples:



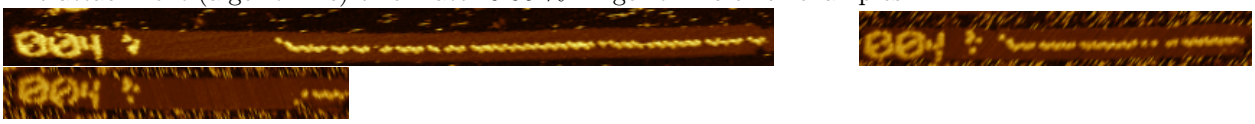
Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:

Seed: 004, input: 110101, output: no, simulation: 

Data examples:



Tile attachment (algorithmic) error rate: 0.06 %. Algorithmic error examples:



Seed: 441, input: 001000, output: yes, simulation: 

Data examples:

Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

Seed: 442, input: 011000, output: no, simulation: 

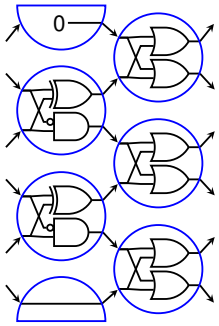
Data examples:

Tile attachment (algorithmic) error rate: 0.08 %. Algorithmic error examples:

Figure S44: Parity circuit on all 64 inputs, showing one nanotube for each input. 'No' instances of the parity problem are on the left, 'yes' instances are on the right. The notation $\sigma(x_0)$ can be read as: "The seed label for input bit string x_0".

S8.3 Circuit: RULE110

Intuition and motivation. The RULE110 circuit simulates the rule 110 cellular automaton. Figure S4 explains how the simulation works. The proof of Theorem S1.1 exploits the idea behind this simulation result to prove that uniform families of n-bit 1-layer IBCs simulate Turing machines.



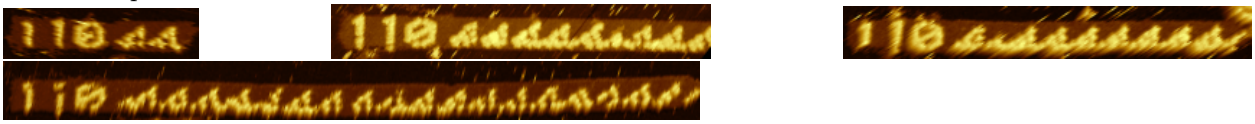
RULE110 circuit

Validation and statistics. 83.1 % of seeds had significant growth (59 seeds with significant growth, 12 seeds without). The measured total amount of tile growth for all RULE110 nanotubes was 21,768 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 48,780 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 21 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.04\ \% \pm 0.009$.

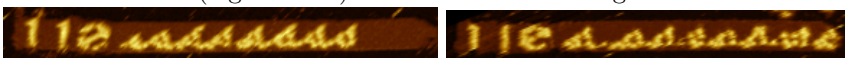
The 2 inputs shown below represent a full coverage test for the RULE110 circuit in the sense that correct computation on all of those inputs requires incorporation of every tile type that would be incorperated on all of the possible 64 inputs.

Seed: 110, input: 000001, simulation: 

Data examples:

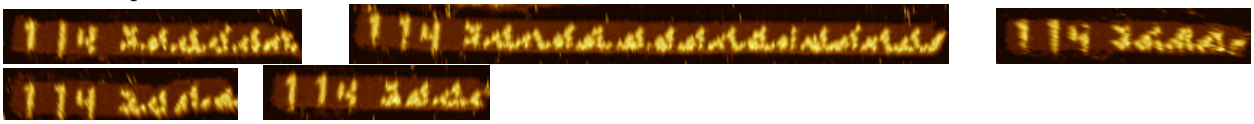


Tile attachment (algorithmic) error rate: 0.06 %. Algorithmic error examples:

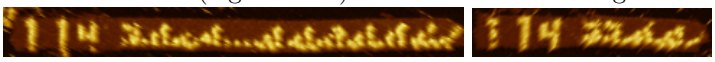


Seed: 114, input: 110001, simulation: 

Data examples:

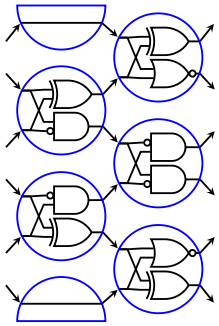


Tile attachment (algorithmic) error rate: 0.03 %. Algorithmic error examples:



S8.4 Circuit: MultipleOf3

Intuition and motivation. The MultipleOf3 circuit determines whether its input bit string represents a binary number that is a multiple of 3. The circuit was found by a stochastic computer search algorithm.



MultipleOf3 circuit

The MultipleOf3 circuit has the following interesting property. After a decision is reached, the circuit repeatedly iterates through a sequence of 6-bit strings. If the input is a multiple of 3, and thus the answer is "yes", that repeated sequence has a single element, 100001, which is 33 in binary and is a multiple of 3. Otherwise the circuit iterates through a sequence of 6-bit strings none of which represent a multiple of 3. In fact, any IBC for the multiple-of-3 decision problem necessarily has the property of only visiting multiples of 3 on accepting computations and non-multiples of 3 on rejecting computations.

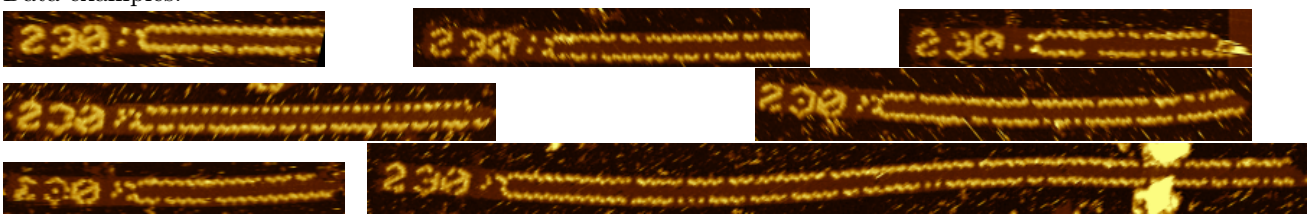
MultipleOf3 decides a language, as defined in Section S1.1.1.

Validation and statistics. 72.8 % of seeds had significant growth (426 seeds with significant growth, 159 seeds without). The measured total amount of tile growth for all MultipleOf3 nanotubes was 210,346 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 471,364 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 146 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of 0.03 % ± 0.002.

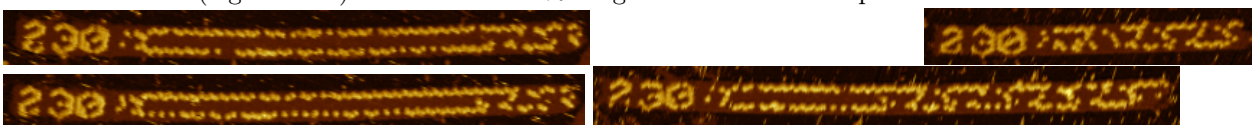
The 4 inputs shown below represent a full coverage test for the MultipleOf3 circuit in the sense that correct computation on these inputs requires incorporation of every tile type for the MultipleOf3 circuit.

Seed: 230, input: 000011, output: yes, simulation: 

Data examples:

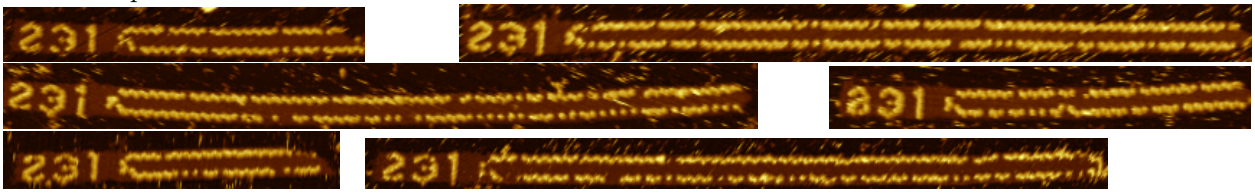


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:

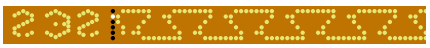


Seed: 231, input: 010101, output: yes, simulation: 

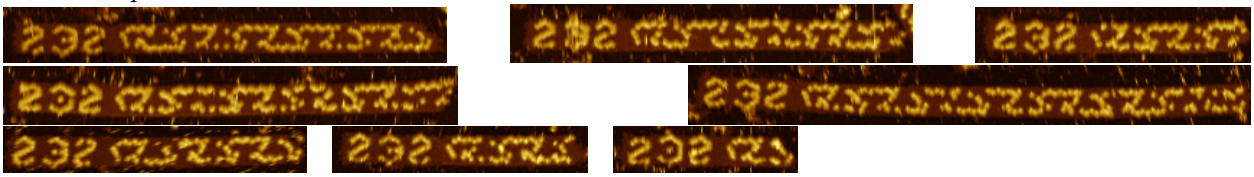
Data examples:



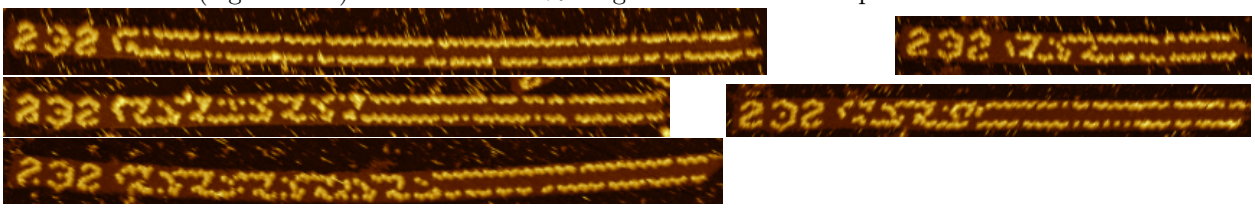
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:

Seed: 232, input: 010000, output: no, simulation: 

Data examples:

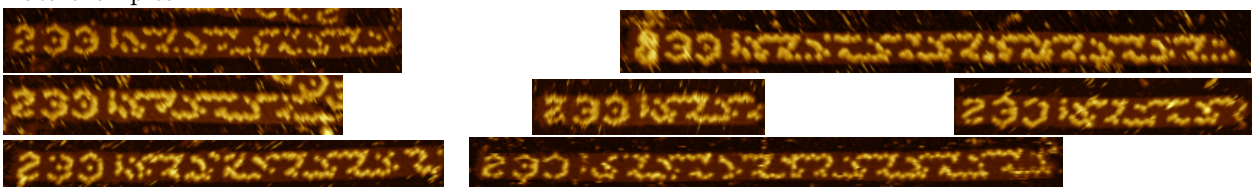


Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:

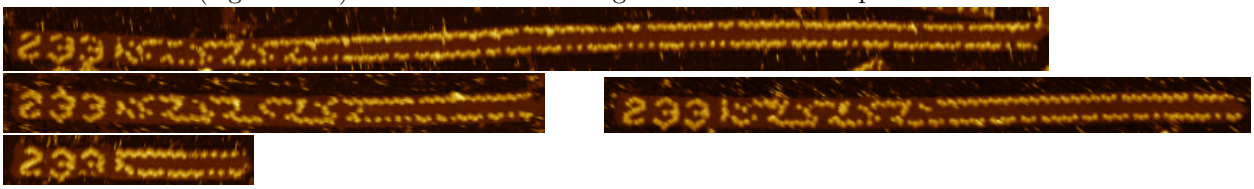


Seed: 233, input: 110101, output: no, simulation: 

Data examples:

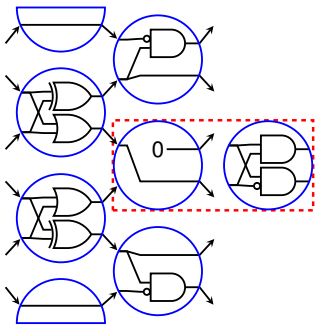


Tile attachment (algorithmic) error rate: 0.05 %. Algorithmic error examples:



S8.5 Circuit: FairCoin

Intuition and motivation. The randomised circuit FairCoin is inspired by Chalk et al's [31] tile-assembly based implementation of von Neumann's procedure [108] for using a biased coin to simulate a fair coin (that has exact 50/50 odds), for any non-zero bias. The circuit has a randomized gate position with one of two outputs (biased coin flips) of bias p (for heads, H) and $1-p$ (for tails, T). The circuit does the following: flip the biased coin twice, if the two results agree, repeat the procedure, but if they differ then report the outcome as the presence (output = yes, if the result was HT) or absence (output = no, if the result was TH) of a stripe. Call this output the *decision* of the circuit.



FairCoin circuit

The following values for p were programmed: $p = 0.5$ for seed 130, $p = 0.9$ for seed 131, $p = 0.4$ for seed 132, $p = 0.7$ for seed 133, and $p = 0.3$ for seed 134. Randomisation was implemented using the "first-tile randomisation" method described in Section S2.3.2.

One way to understand the intuition behind the circuit is the tile assembly outlined in Fig. S48. Two runs are shown: one with decision "Yes" (H followed by T) and another with decision "No" (T followed by H). Due to the constraint that each wire in the circuit can carry one of only two different values, we must sometimes reuse the same bits in the circuit to mean different things depending on the surrounding context. Each glue describes a bit, but the bits are given names that evoke their purpose. As in every other tile implementation of a circuit, each glue is implicitly marked with its row, so that for example H in the middle row is a different glue from H in another row.

Two glues in the same row with the same letter, where one is uppercase and the other is lowercase, represent the same glue, but the case is intended to indicate that, due to the surrounding context, the glue has a different semantic meaning. For example, the glue H in the middle row means "H was the outcome of the last coin flip", whereas h, which is actually the same glue, represents "we are no longer flipping coins but are using this glue to propagate different information".

The green tiles in the middle row represent the coin flips. They come in pairs: first a dark green tile then a light green tile (the tiles are identified solely by their glues, so a tile with two f's on the left and two H's on the right is the same whether it is dark or light, but the surrounding tiles are used to give a "parity" to the two coin flips and identify which flip is "first" and which is "second". The red tiles represent tiles that are used only after a decision has been obtained (because of the two coin flips having different outcomes).

middle row. Glues on the left are f or s, meaning "flip" or "stop flipping", the latter occurring after two consecutive flips have different outcomes, meaning we have obtained a decision to report. The glue F (which is the same as f) is used after a decision is reached in the case of a "Yes" decision (but obviously no longer means "flip", hence the case change in the figure).

Glues on the right of tiles in the middle row represent heads (H) or tails (T). After a decision is obtained, we use h and t instead of H and T. Recall that as described above, the glues h and H are the same glue, but the lowercase gives a visual reminder that the meaning has changed due to the surrounding context, and no longer represents the outcome of a coin flip.

Whenever two consecutive tiles are the same (both H or both T), the tiles above the middle row are symmetric to those below the middle row, so below we describe mainly the rows above the middle, referencing the bottom row only to point out when there is an asymmetry between them (which only occur after two consecutive flips are different: HT or TH).

top row. The bits e and o alternate once every full layer, until either the outcomes HT or TH occur, at which point they alternate twice as often, every half-layer. Prior to a decision occurring, this provides a "parity" bit to help distinguish dark purple tiles in the row below (carrying information about the outcome of the first coin flip) from the gray tiles in the same row (which do not communicate a coin flip outcome, but just indicate whether a decision has occurred or not by virtute of whether the bottom left glue is h or t).

two rows just above middle. The glues H and T represent head and tail coin flips as in the middle row. However, the glue h is used whether the coin flip outcome below it is heads or tails. The dark purple tiles can be thought to be propagating the outcome of the first coin flip through their bottom-left and top-right glues. The top-left glue h in a dark purple tile is the same as H in a light purple tile; but the

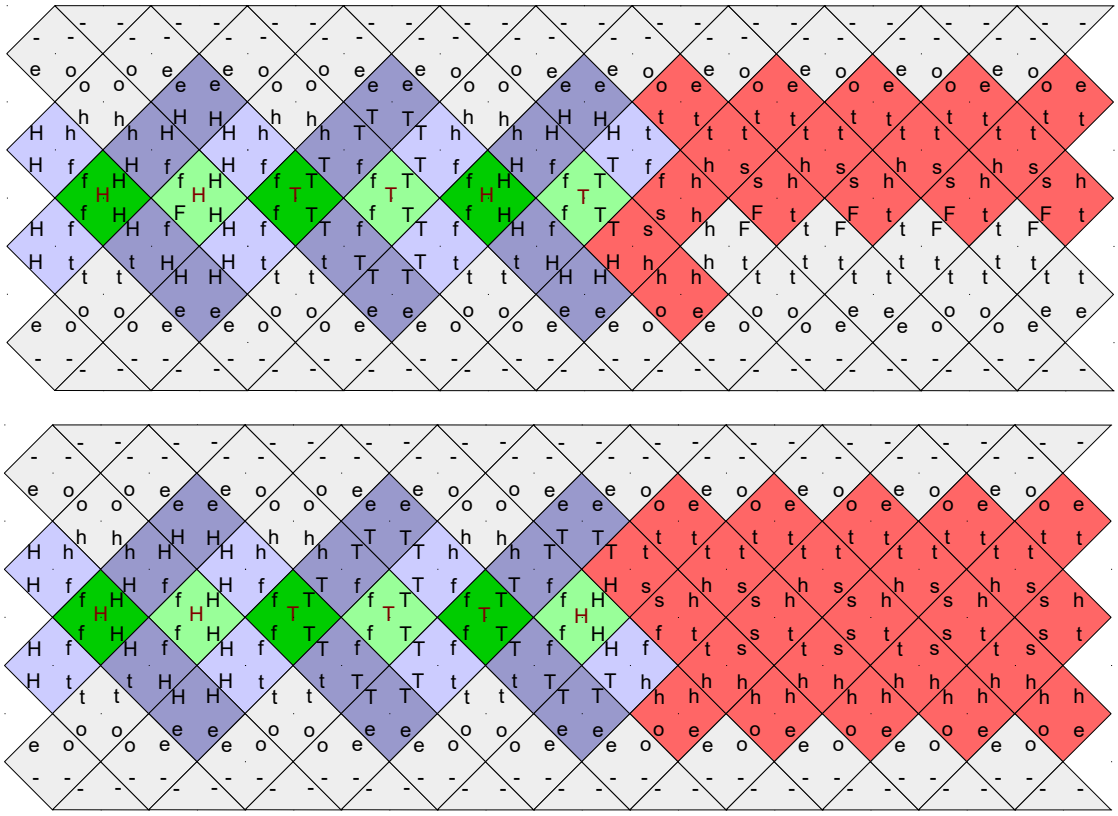


Figure S48: Tiles implementing the FairCoin circuit. Top shows a "Yes" decision (HT). Bottom shows a "No" decision (TH). Both have two coin flip pairs HH and TT before the decision is obtained. Green tile types implement the biased coin tosses. Purple tile types communicate and process these outcomes to compare and see if two consecutive coin tosses are equal. Red tile types are those that do not appear until a decision is obtained.

lowercase h does not represent the outcome of a coin flip; it is used in every dark purple tile simply to allow cooperative attachment but it otherwise meaningless. In the light purple tile, the top-left glue is H or T, which is communicated from the topmost dark purple tile.

Thus, a light purple tile whose two left glues are both H, or are both T, implies that the two consecutive coin flip outcomes were identical, and an f glue appears on the bottom right to continue the experiment. However, if one is H and the other is T, then the two consecutive coin flip outcomes were different.

If the first was H (represented on top left glue of light purple tile) and the second was T (represented on bottom left glue of light purple tile), then the f glue is still placed to the bottom right, but the glue t rather than h is put on the top right. (Note that in the symmetric light purple tile below the middle row, the t glue, which had previously appeared in that position, is changed to h, and f is changed to s.

In the case of the other decision, where the first of the two consecutive coin flips was H (represented on the top left glue of the light purple tile) and the second was T (represented on the bottom left glue of the light purple tile), the roles are reversed between the light purple tiles above and below the middle row. Both flip h to t or t to h as before, but now the top tile changes f to s, whereas the bottom tile remains f.

Both decisions render the circuit deterministic from this point on. The various bits (H/T, f/s, e/o) were assigned to 0 and 1 in the implemented circuit so as to have the bottom decision be all 0's on the red tiles, so the top decision has a visible stripe of 1's. This made the two decisions easy to distinguish by AFM.

Validation and statistics. 73.8 % of seeds had significant growth (714 seeds with significant growth, 254 seeds without). The measured total amount of tile growth for all FairCoin nanotubes was 243,509 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 545,678 tiles

attached (assuming a DNA base pair length of 0.34 nm). We counted 75 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.01\ \% \pm 0.001$.

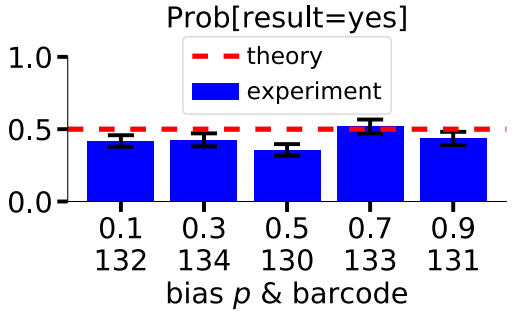
The 5 inputs shown below represent a full coverage test for the FairCoin circuit in the sense that a large enough set of correct randomised computations on these inputs requires incorporation of every tile type for the FairCoin circuit.

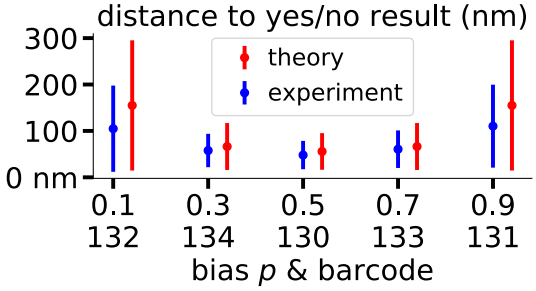
The FairCoin circuit was tested over a 100-fold range of concentration ratios. Out of a total of 714 analysed computations, 643 reached a decision before the end of the nanotube. 274 of those had a stripe (decision "yes"; mean and s.d. distance to decision 76.8 ± 70.2 nm) and 369 had no stripe (decision "no"; mean distance to decision 76.9 ± 71.7 nm). Thus, in aggregate over all five inputs, 43.0 % of the nanotubes had a stripe and 57.0 % did not (not too far from the expected 50 %). As shown below, we also saw decision times that (as expected) increase for an increasingly biased source of randomness.

Say a *trial* of the FairCoin circuit is two IBC layer iterations, i.e., enough to flip the coin twice. For each nanotube we estimated the number of trials until a decision, or no decision by end of the nanotube, by manually reading the streptavidin pattern; mean and s.d. number of trials for "yes" was 2.58 ± 2.4 and for "no" was 2.46 ± 2.4. From these data we computed an estimated mean trial length of 30.6 nm.

Theoretical analysis. With bias p, each trial has probability $2p(1-p)$ to reach a decision, so the number of trials until a decision is a geometric random variable with parameter $2p(1-p)$ [109, Chapter 2], corresponding to expected trials $\frac{1}{2p(1-p)}$ before a decision is reached. Converting from trials to nanometers (each trial is 30.6 nm, see above), the expected distance to a decision is $\frac{30.6}{2p(1-p)}$ nm. The variance of a geometric random variable with parameter r is $\frac{1-r}{r^2}$, so standard deviation $\frac{\sqrt{1-r}}{r}$. Setting $r = 2p(1-p)$, this is $\frac{\sqrt{1-2p(1-p)}}{2p(1-p)}$. Translating from trials of the FairCoin circuit to nanometers, this is $30.6 \cdot \frac{\sqrt{1-2p(1-p)}}{2p(1-p)}$ nm. Results for each of the five biases implemented, and comparisons to these expected values, are summarised in Figure S49.

Experiments with longer predicted lengths ($p = 0.1$ or 0.9) seem to deviate from prediction more than those with shorter predicted lengths. Some of this may be due to bias from limited nanotube length: tubes that took longer to reach a decision were more likely to stop growing before a decision was made and thus not be counted in the dataset.



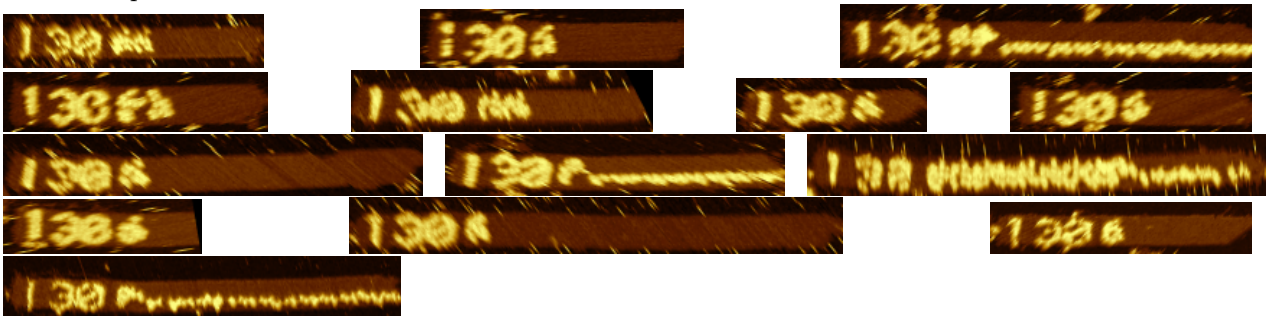


barcode	**132**	**134**	**130**	**133**	**131**
bias	0.1	0.3	0.5	0.7	0.9
measured Prob[decision=yes]	0.42 ± 0.04	0.43 ± 0.04	0.36 ± 0.04	0.52 ± 0.05	0.44 ± 0.05
expected Prob[decision=yes]	0.5	0.5	0.5	0.5	0.5
sample mean distance to decision	104.8 ± 7.5 nm	57.8 ± 3.3 nm	48.0 ± 2.6 nm	60.6 ± 3.9 nm	110.2 ± 8.3 nm
expected distance to decision	170.0 nm	72.9 nm	61.2 nm	72.9 nm	170.0 nm
sample stddev of distance	92.9 nm	35.9 nm	30.5 nm	40.5 nm	89.4 nm
theoretical stddev of distance	153.9 nm	55.5 nm	43.3 nm	55.5 nm	153.9 nm

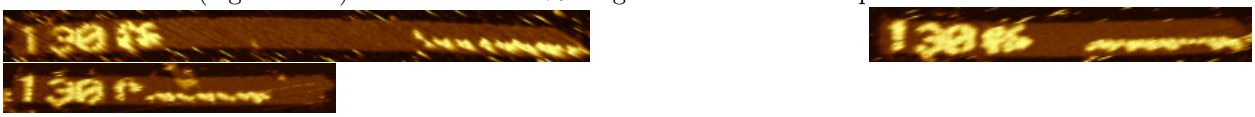
Figure S49: Summary of results for the FairCoin circuit on five different biases. Error bars on fraction of "yes" decisions represent standard error of the mean as defined in Section S8.0.2. Standard deviation of distance measurements, both theoretical and sampled, are represented by error bars in plot called "stddev" in the table. In measured Prob[decision]="yes" (i.e., the mean of 0/1-valued Bernoulli variables) and sample mean distance to decision, $m \pm e$ is mean m and standard error of the mean e as defined in Section S8.0.2.

Seed: 130, input: 010100, bias $p = 0.5$, simulations:

Data examples:



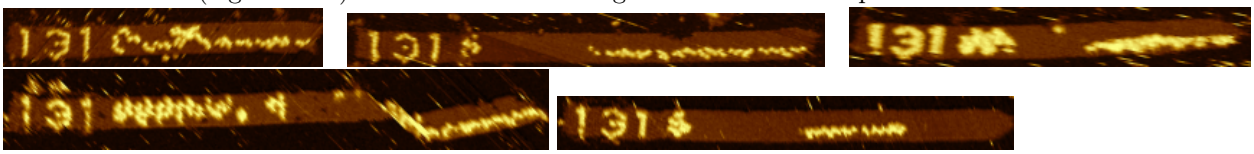
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:



Seed: 131, input: 010100, bias $p = 0.9$, simulations:

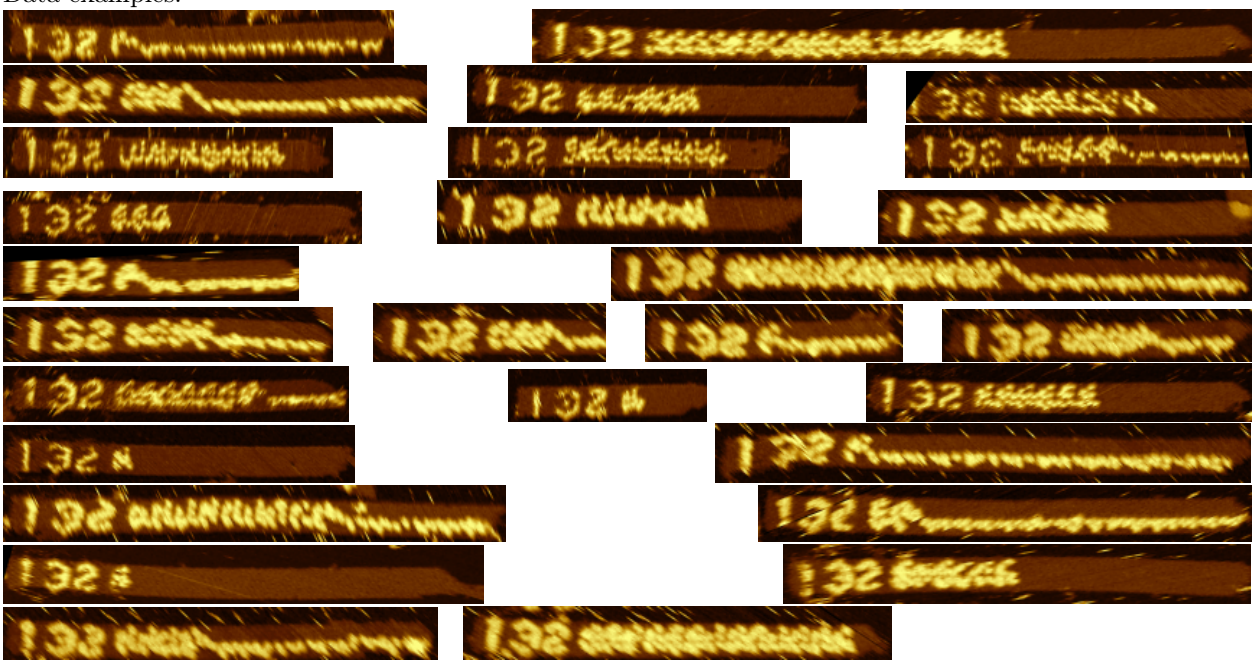
Data examples:

Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:

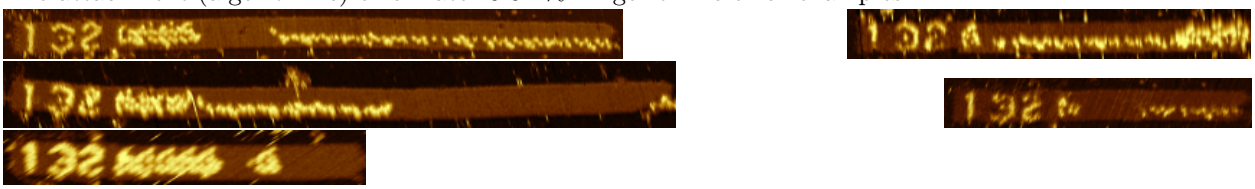


Seed: 132, input: 010100, bias $p = 0.1$, simulations:

Data examples:



Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:



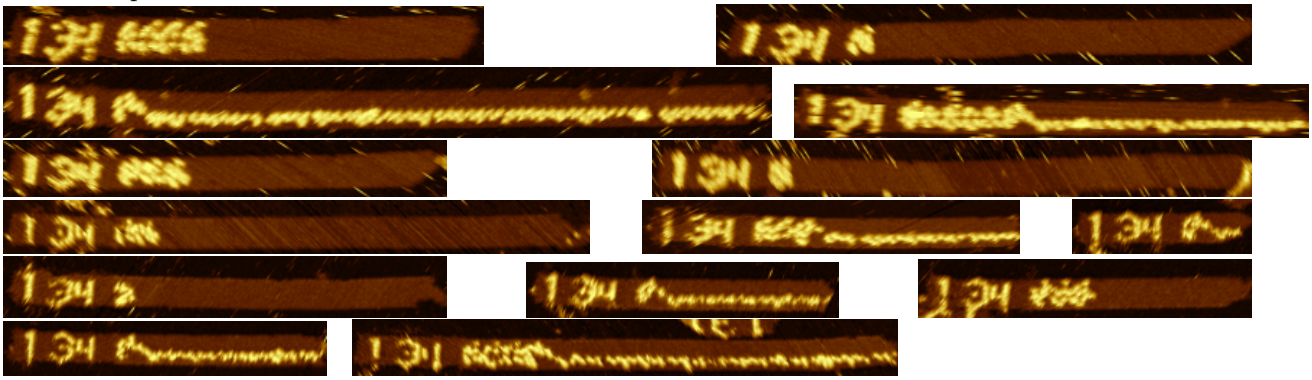
Seed: 133, input: 010100, bias $p = 0.7$, simulations:

Data examples:

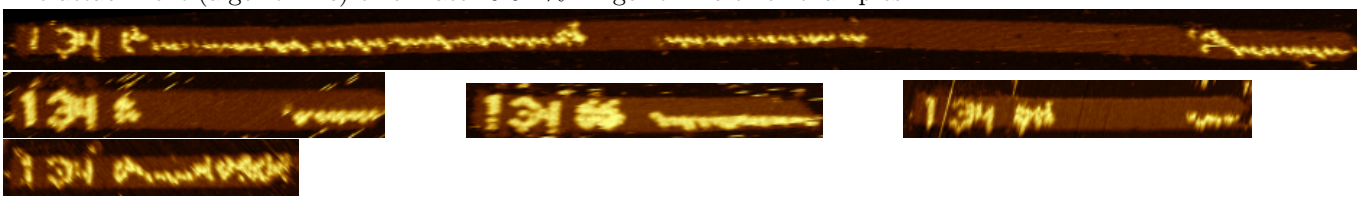
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:

Seed: 134, input: 010100, bias $p = 0.3$, simulations:

Data examples:

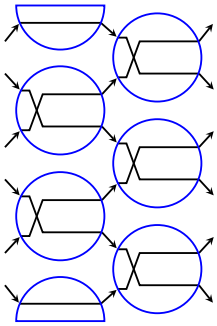


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:



S8.6 Circuit: Zig-Zag

Intuition and motivation. The Zig-Zag circuit makes a simple quickly-repeating zig-zag pattern. The circuit is programmed by having every gate give an output that is its two inputs swapped. Any 1-bit in the 6-bit input that is at (wire) position 2, 4 or 6 will be set downwards, and any 1 bit- at positions 1, 3 or 5 will be sent upwards, making a 1-signal. These 1-signals forever bounce off the top and bottom giving a simple zig-zag pattern. Sending two 1's spaced far enough apart (e.g. input 001001) gives a pattern reminiscent of a 2D projection of the DNA double-helix.



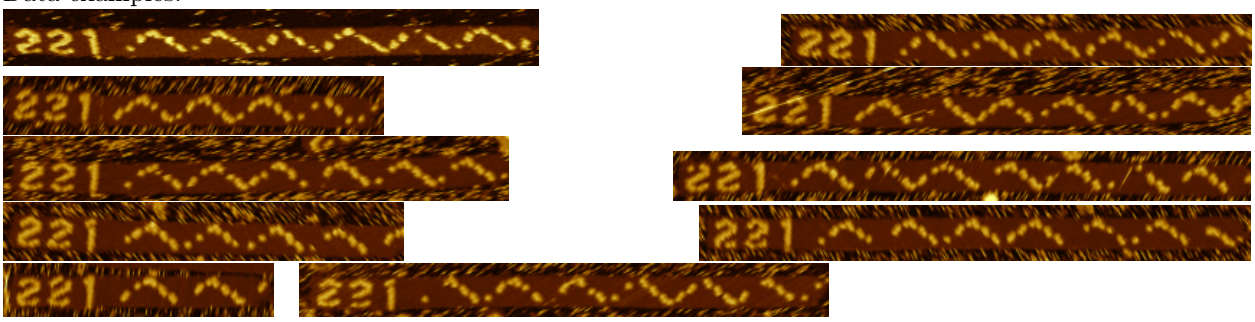
Zig-Zag circuit

Validation and statistics. 65.2 % of seeds had significant growth (596 seeds with significant growth, 318 seeds without). The measured total amount of tile growth for all Zig-Zag nanotubes was 249,051 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 558,097 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 387 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.07\ \% \pm 0.003$.

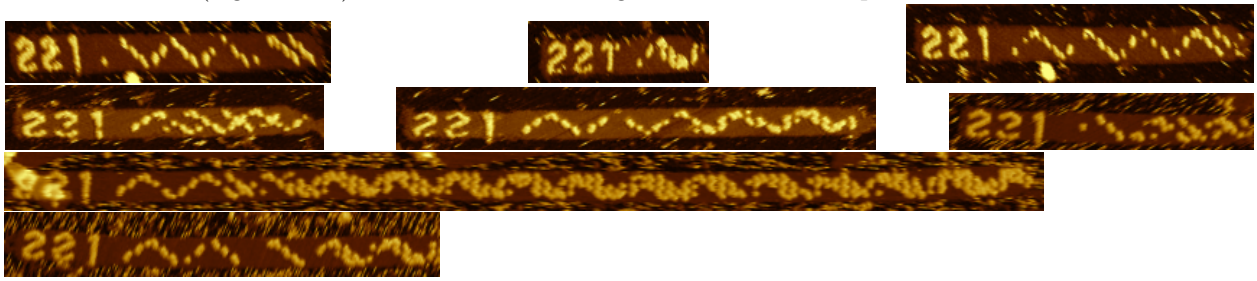
The Zig-Zag circuit happened to have the highest algorithmic error rate of any circuit. It was also challenging to get high-yield of streptavidin labelling; the downward moving "zags" labelled well, but the upward moving "zigs" did not. As part of debugging the labelling issue we happened to take a lot of Zig-Zag data (more than 10 % of total measured nanotube length in the dataset was for Zig-Zag), which may have biased the algorithmic error rate of the entire dataset upwards (worse).

Seed: 221, input: 000001, simulation: 

Data examples:

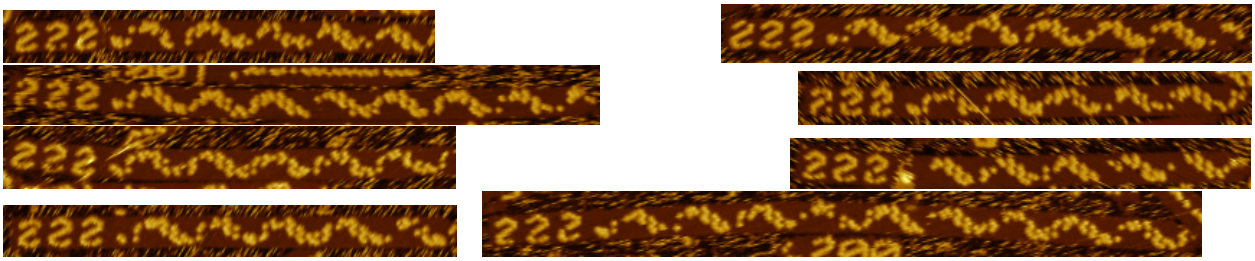


Tile attachment (algorithmic) error rate: 0.06 %. Algorithmic error examples:

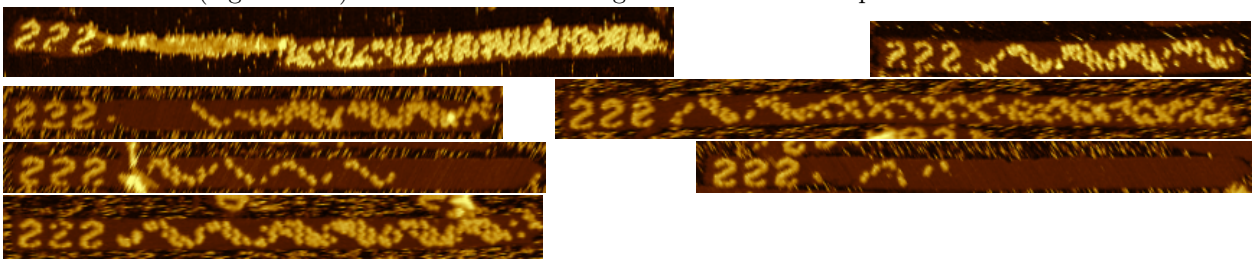


Seed: 222, input: 000101, simulation: 

Data examples:

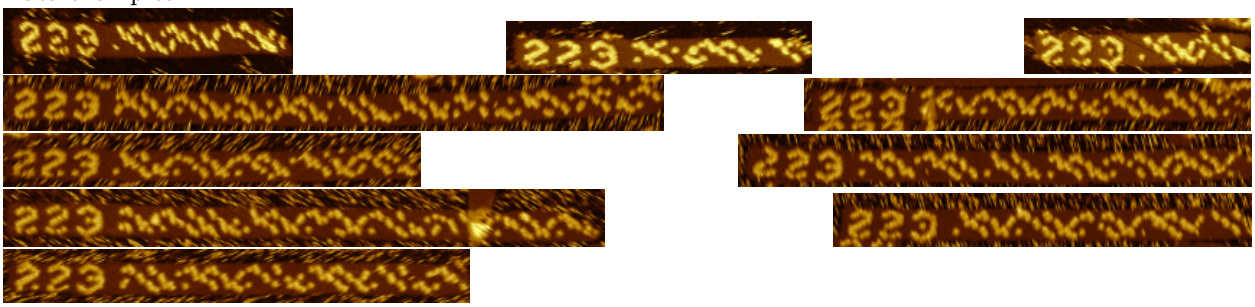


Tile attachment (algorithmic) error rate: 0.07 %. Algorithmic error examples:

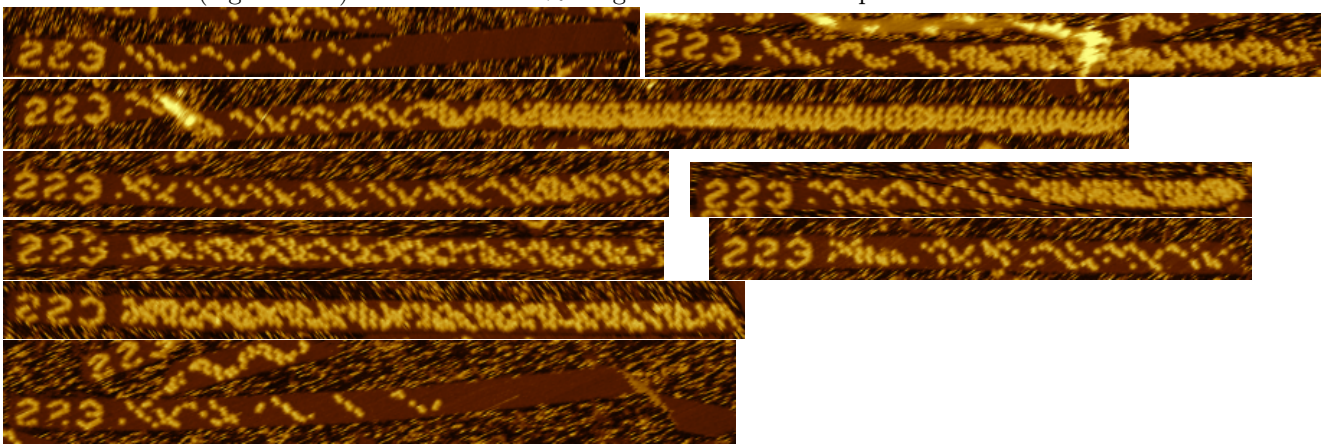


Seed: 223, input: 001001, simulation: 

Data examples:



Tile attachment (algorithmic) error rate: 0.08 %. Algorithmic error examples:

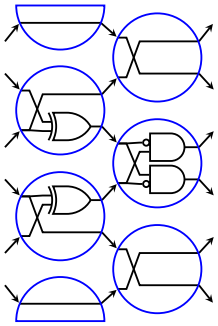


S8.7 Circuit: PALINDROME

Intuition and motivation. The PALINDROME circuit determines whether its 6-bit input bit string is a palindrome. The circuit was found by a stochastic computer search algorithm.

The PALINDROME circuit has the following interesting property (that would be shared by any circuit for the palindrome problem). After a decision is reached, the circuit repeatedly iterates through a sequence of 6-bit strings. If the input is a palindrome, and thus the answer is "yes", then every string in that repeated sequence is a palindrome. Otherwise the circuit iterates through a sequence of non-palindromes. In fact, any n-bit IBC for the palindrome decision problem necessarily has the property of only visiting n-bit palindromes on accepting computations and non-palindromes on rejecting computations.

PALINDROME decides a language, as defined in Section S1.1.1.



PALINDROME circuit

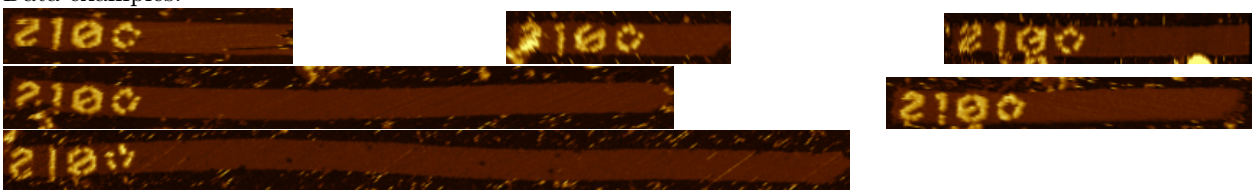
Validation and statistics. 69.5 % of seeds had significant growth (407 seeds with significant growth, 179 seeds without). The measured total amount of tile growth for all PALINDROME nanotubes was 162,422 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 363,970 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 73 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of 0.02 % ± 0.002.

The 4 inputs shown below represent a full coverage test for the PALINDROME circuit in the sense that correct computation on these inputs requires incorporation of every tile type for the PALINDROME circuit.

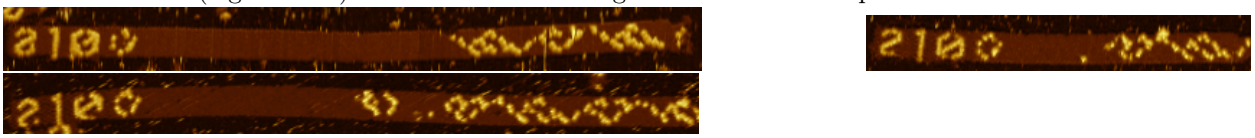
Analysing algorithmic errors for the PALINDROME circuit on "no" instances (i.e. inputs 111011, 110101) is challenging due to the complicated output bit/streptavidin pattern on the nanotubes. We could only reliably resolve whether the circuit flipped its answer to "yes" for a few layers in which case we recorded an algorithmic error. This highlights the fact that for certain circuits on certain inputs spotting algorithmic errors is not straightforward.

Seed: 210, input: 001100, output: yes, simulation:

Data examples:

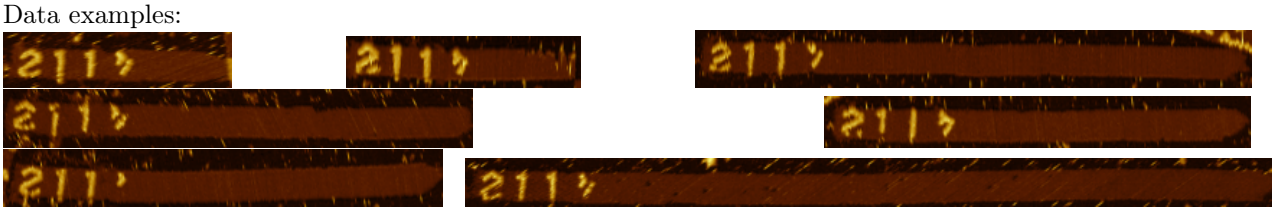


Tile attachment (algorithmic) error rate: 0.03 %. Algorithmic error examples:

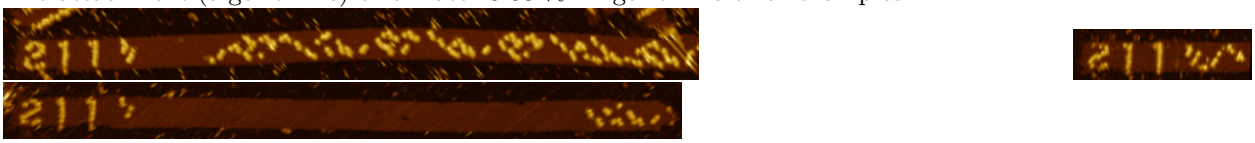


Seed: 211, input: 110011, output: yes, simulation:

Data examples:

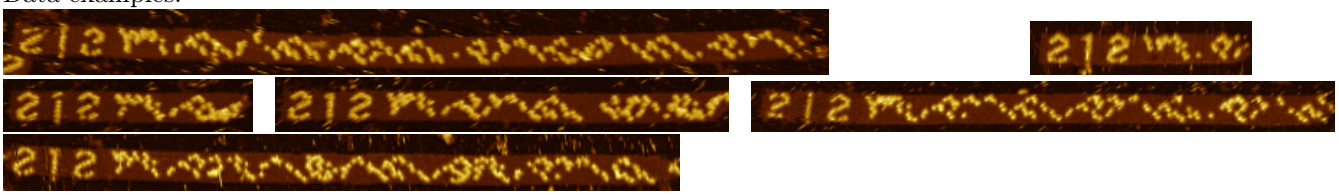


Tile attachment (algorithmic) error rate: 0.03 %. Algorithmic error examples:

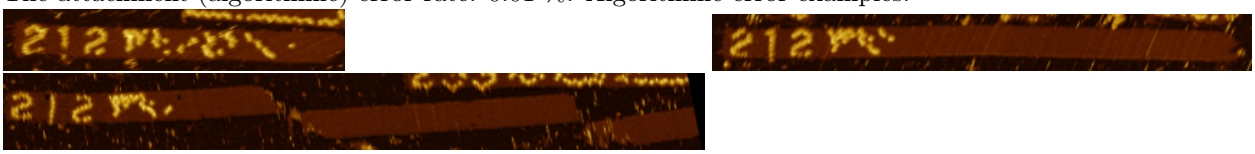


Seed: 212, input: 111011, output: no, simulation: 

Data examples:

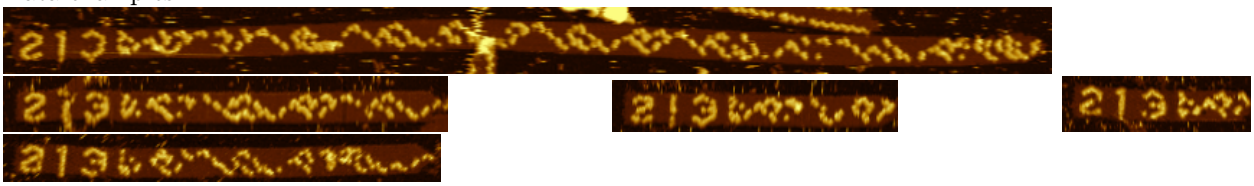


Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:



Seed: 213, input: 110101, output: no, simulation: 

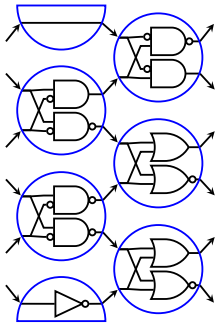
Data examples:



Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:

S8.8 Circuit: Recognise21

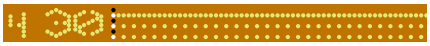
Intuition and motivation. The Recognise21 circuit is a decider IBC that accepts 010101, the binary representation of the number 21, and rejects all other 6-bit words. This was one of the final circuits we designed and was created for tile coverage reasons: to ensure that every tile was used in at least one circuit.



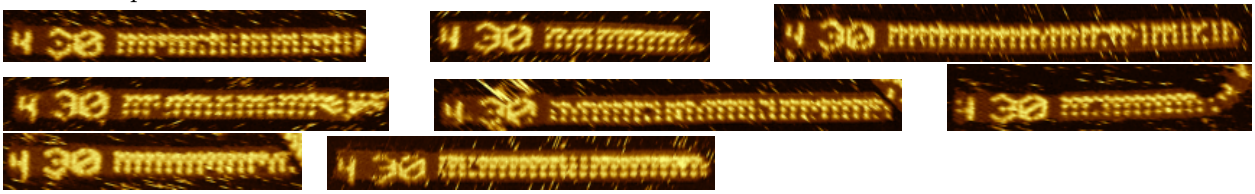
Recognise21 circuit

Validation and statistics. 67.7 % of seeds had significant growth (313 seeds with significant growth, 149 seeds without). The measured total amount of tile growth for all Recognise21 nanotubes was 77,646 nm (excludes the 21-circuit dataset mean seed length of 155.6±0.18 nm) giving an estimated 173,997 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 10 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of 0.01 % ± 0.001.

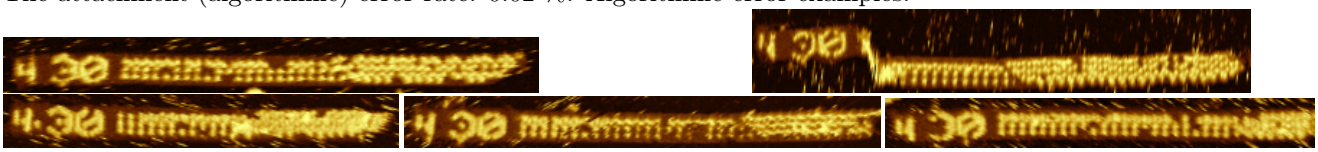
The 4 inputs shown below represent a full coverage test for the Recognise21 circuit in the sense that correct computation on these inputs requires incorporation of every tile type for the Recognise21 circuit.

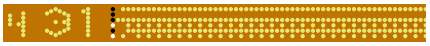
Seed: 430, input: 010101, output: yes, simulation: 

Data examples:

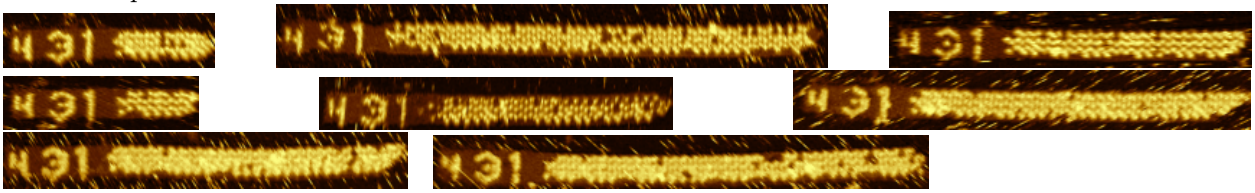


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:

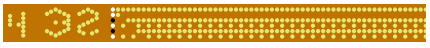


Seed: 431, input: 000101, output: no, simulation: 

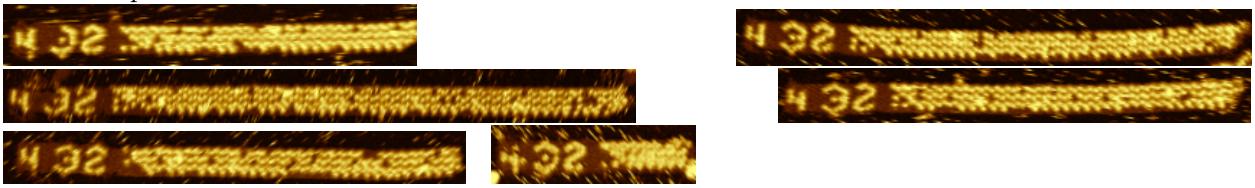
Data examples:



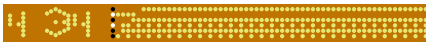
Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

Seed: 432, input: 110101, output: no, simulation: 

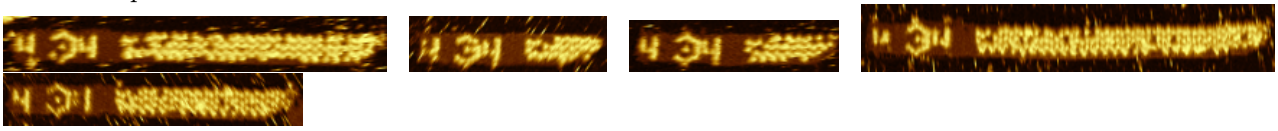
Data examples:



Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

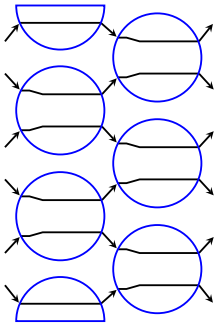
Seed: 434, input: 010100, output: no, simulation: 

Data examples:



Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

S8.9 Circuit: Copy



Copy circuit

Intuition and motivation. The Copy circuit copies its 6-bit input string to the right. Each gates simply copies the bit(s) on its input wire(s) to their (respective) output wire(s). In other words, each gate computes the identity function, and the circuit computes the identity function on 6 bits.

Part of the motivation for the Copy circuit was to compare with previous bit-copying self-assembly systems [92, 8, 9] as well as the 4-bit copy system in Section S5.2 developed as part of this study. Furthermore, it is a relatively easy tile set to analyse for algorithmic errors.

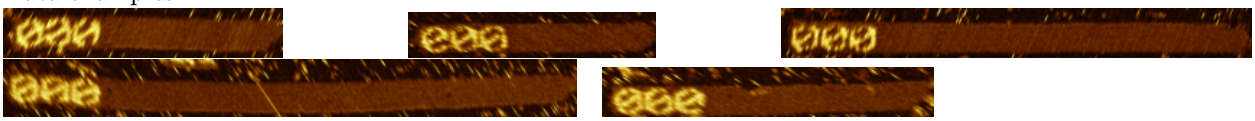
Validation and statistics. 48.5 % of seeds had significant growth (205 seeds with significant growth, 218 seeds without). The measured total amount of tile growth for all Copy nanotubes was 50,754 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 113,734 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 17 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.01\ \% \pm 0.003$.

The 8 inputs shown below represent a full coverage test for the Copy circuit in the sense that correct computation on these inputs requires incorporation of every tile type for the Copy circuit.

There have been previous DNA-tile algorithmic self-assembly systems that copy bits. The algorithmic error rate of $0.01\ \% \pm 0.003$ for our Copy circuit is comparable to the best error rates previously reported [9], which was $0.017\ \% \pm 0.013$, estimated from 18,029 tile attachments. Although the DX tile design used in [9] is different than our SST motif, and the system design pipeline is quite different in both cases (our Copy is taken from a larger designed DNA tile set) it is instructive to have a metric on which to compare the two.

Seed: 000, input: 000000, simulation: 

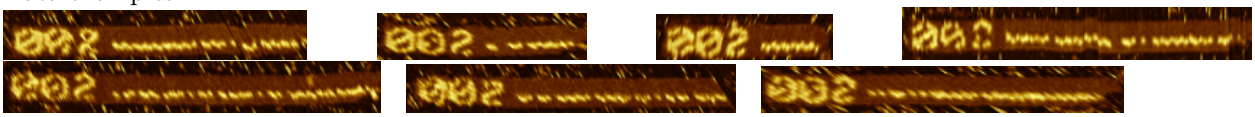
Data examples:



Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

Seed: 002, input: 000010, simulation: 

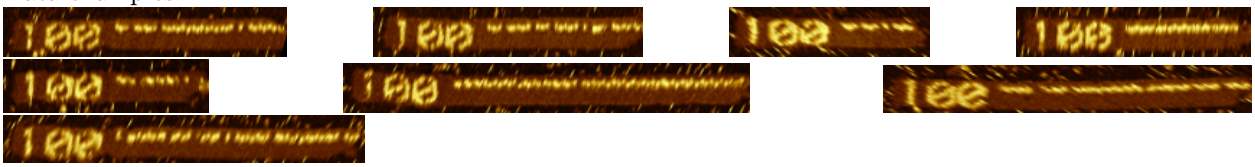
Data examples:



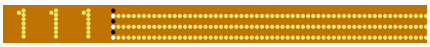
Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

Seed: 100, input: 010000, simulation: 

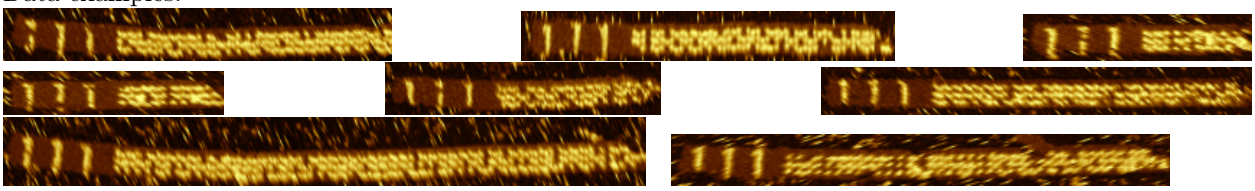
Data examples:



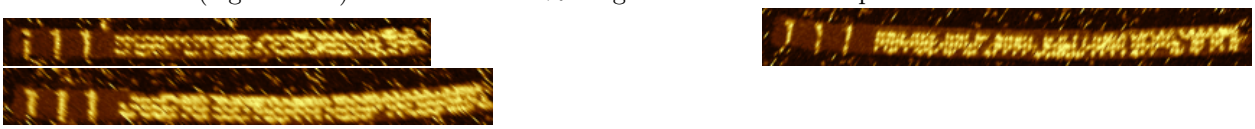
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:

Seed: 111, input: 010101, simulation: 

Data examples:



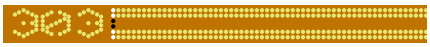
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:



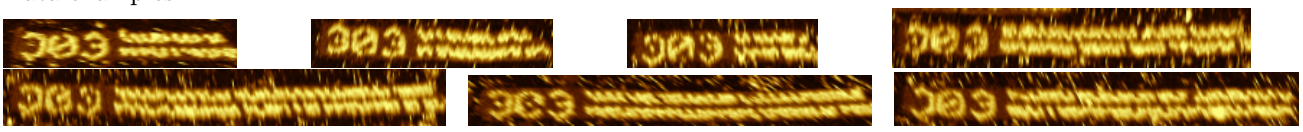
Seed: 102, input: 010010, simulation: 

Data examples:

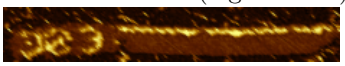
Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

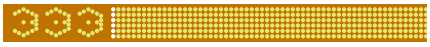
Seed: 303, input: 110011, simulation: 

Data examples:

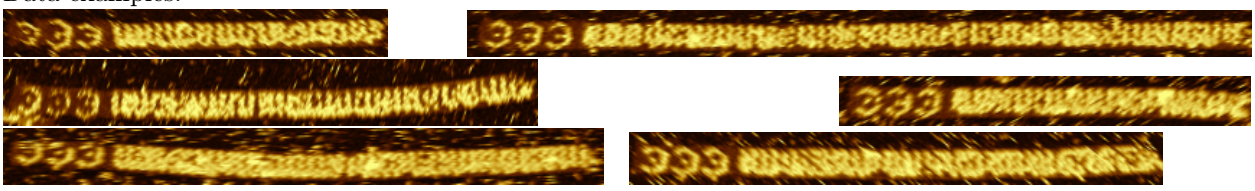


Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error example:

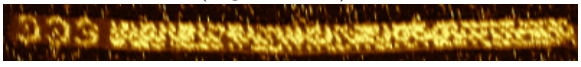


Seed: 333, input: 111111, simulation: 

Data examples:

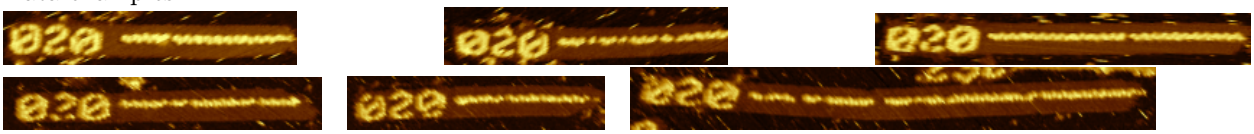


Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error example:



Seed: 020, input: 001000, simulation: 

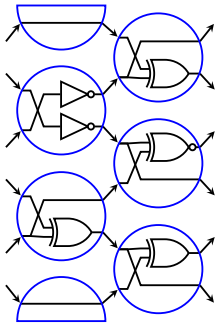
Data examples:



Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:

S8.10 Circuit: Cycle63

Intuition and motivation. The Cycle63 circuit has one of two repeating patterns: one of which repeats every layer the other repeats every 63 layers. The circuit was found by computer search. Since circuits have 6 "wires" we know that there is no circuit that has a minimum repeat interval of strictly more than $2^6 = 64$ layers. Using a group-theoretic argument, there is a proof that 64 is not achievable [110]. Hence, the Cycle63 circuit achieves the maximum number of layers before a repeat in the 6-bit 1-layer deterministic model.



Cycle63 circuit

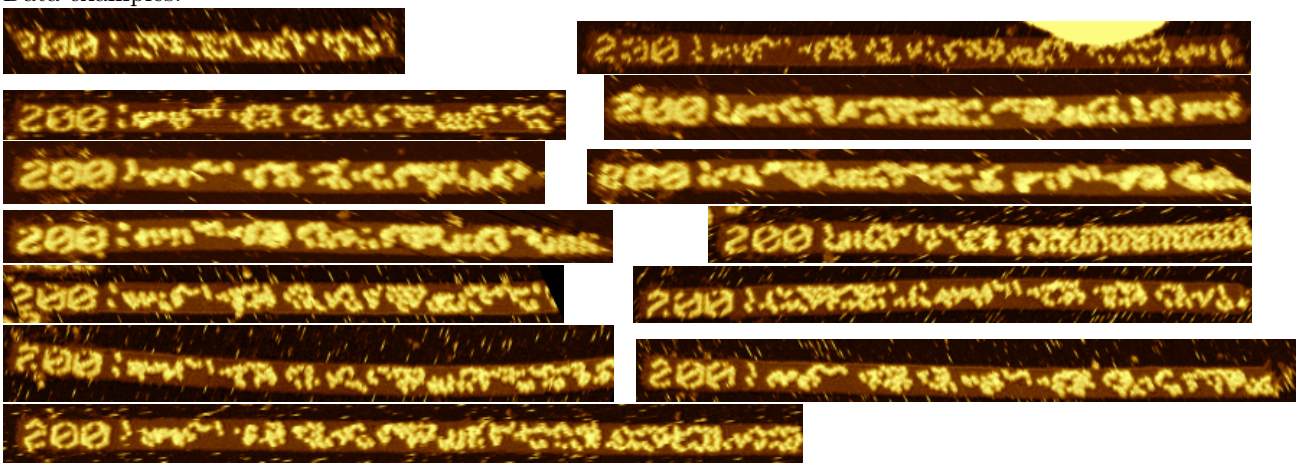
Validation and statistics. 53.7 % of seeds had significant growth (197 seeds with significant growth, 170 seeds without). The measured total amount of tile growth for all Cycle63 nanotubes was 76,108 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 170,550 tiles attached (assuming a DNA base pair length of 0.34 nm). Tile attachment errors (algorithmic errors) were not analysed for this circuit.

The 1 inputs shown below represent a full coverage test for the Cycle63 circuit in the sense that correct computation on these inputs requires incorporation of every tile type for the Cycle63 circuit.

The reason we did not analyse algorithmic errors for Cycle63 was that due to the complexity of the pattern of bits (streptavidins), and also due to partially incomplete streptavidin labelling, it is often difficult to pinpoint where an error occurs. However, from looking at the few representative data examples below, it can be seen that at least a single algorithmic error occurred in every sufficiently long enough nanotube.

Seed: 200, input: 111101, simulation: 

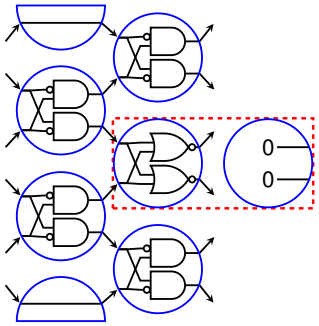
Data examples:



S8.11 Circuit: DiamondsAreForever

Intuition and motivation. The randomised circuit DiamondsAreForever generates a pattern of diamonds at random intervals, forever. Increasing seed number implies increasing probability of creating a diamond per circuit iteration, termed Pr[diamond/step].

The following values of Pr[diamond/step] = p were programmed: $p = 0.1$ for seed 300, $p = 0.3$ for seed 301, $p = 0.4$ for seed 302. Randomisation was implemented using the "first-tile randomisation" method described in Section S2.3.2.



DiamondsAreForever circuit

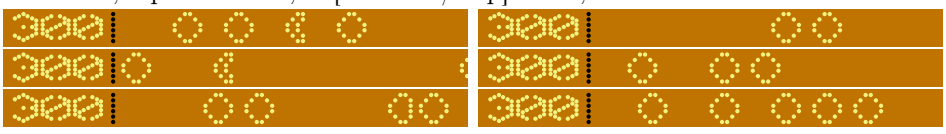
Validation and statistics. 76.4 % of seeds had significant growth (84 seeds with significant growth, 26 seeds without). The measured total amount of tile growth for all DiamondsAreForever nanotubes was 25,500 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 57,143 tiles attached (assuming a DNA base pair length of 0.34 nm). Tile attachment errors (algorithmic errors) were not analysed for this circuit.

Additional data specific to the DiamondsAreForever circuit was analysed as follows. For each nanotube, and each diamond pattern on a nanotube, the distance from the input-adapters to the beginning of the diamond pattern was measured. From those measurements we computed what we term "distance between diamond creation" per diamond, which is the distance from the beginning of each diamond to whichever comes first: the beginning of a diamond before it or the input-adapters.

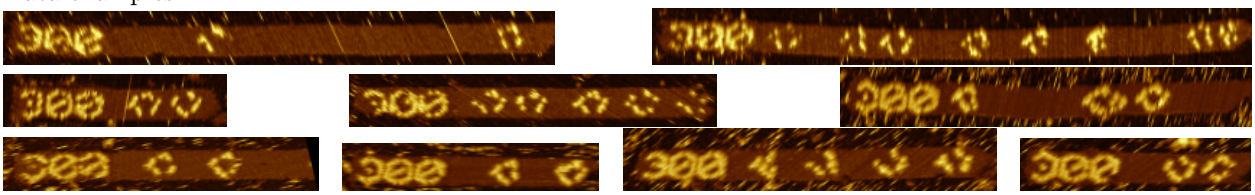
Theoretical analysis. We model the distance between diamond creation as a geometric random variable with parameter p [109, Chapter 2], where p = Prob[diamond/step], with expected number of IBC layer iterations $\frac{1}{p}$. Since each iteration is 14.28 nm, this corresponds to expected length $\frac{14.28}{p}$ nm. The variance of a geometric random variable with parameter p is $\frac{1-p}{p^2}$, so standard deviation $\frac{\sqrt{1-p}}{p}$. Since each IBC layer iteration is 14.28 nm, the predicted standard deviation in length is $14.28 \cdot \frac{\sqrt{1-p}}{p}$. In the table below, both sample mean distance (measured from data) and expected distance (predicted from theory), as well as sample standard deviation (sample stddev) and theoretical standard deviation (theoretical stddev), are shown. For sample mean m, the standard error of the mean e is shown as $m \pm e$.

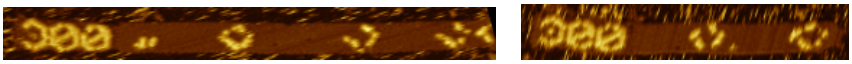
barcode	**300**	**301**	**302**
nanotubes analysed	54	93	176
Prob[diamond/step]	0.1	0.3	0.4
sample mean distance between diamonds	103.0 ± 12.1 nm	61.4 ± 5.0 nm	56.1 ± 3.2 nm
expected distance between diamonds	142.8 nm	47.6 nm	35.7 nm
sample stddev of distance	89.0	47.9	42.4
theoretical stddev of distance	135.5	39.8	27.7

Seed: 300, input: 000000, Pr[diamond/step] = 0.1, simulations:

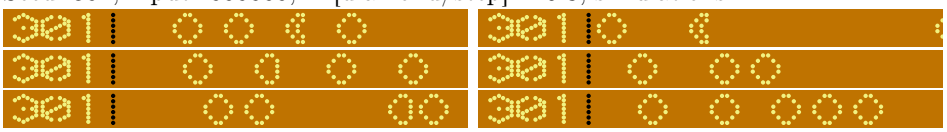


Data examples:

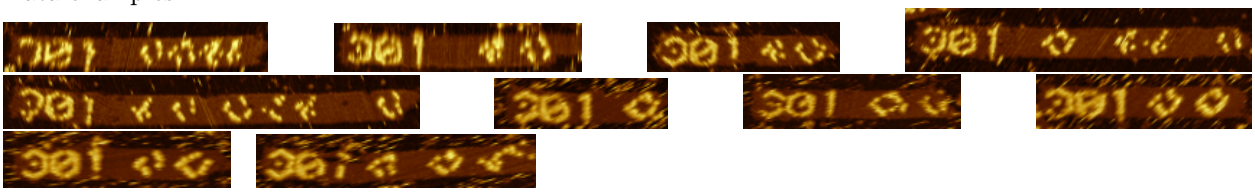




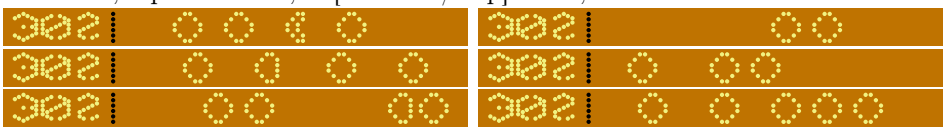
Seed: 301, input: 000000, Pr[diamond/step] = 0.3, simulations:



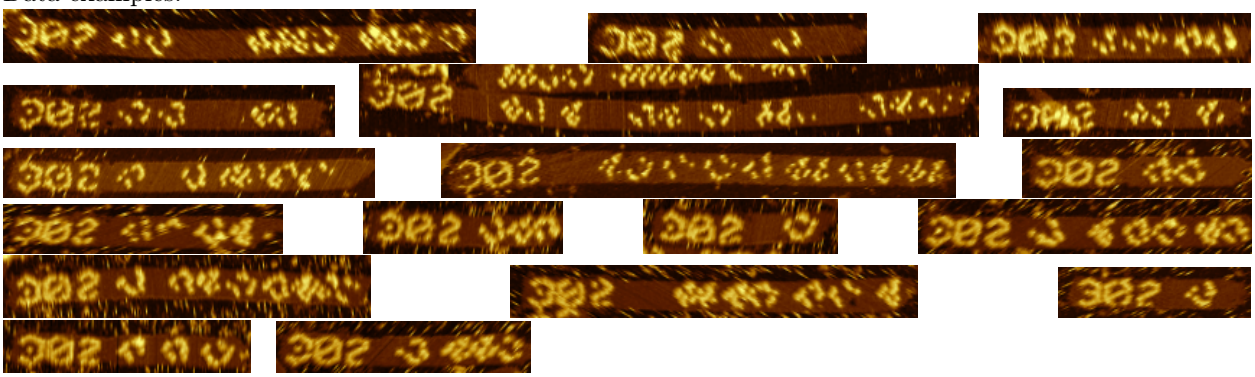
Data examples:



Seed: 302, input: 000000, Pr[diamond/step] = 0.4, simulations:

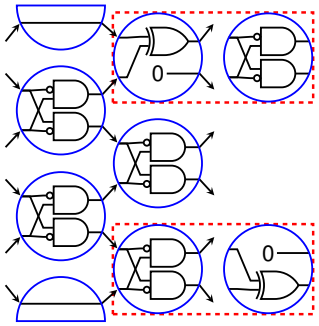


Data examples:



S8.12 Circuit: Waves

Intuition and motivation. The Waves circuit is a randomised circuit that generates a wave like pattern. On input 000001 (a single 1 on the bottom of the 6 inputs), at each iteration the 1 is either copied along the bottom with probability Pr[create wave], or moved upwards with probability 1 − Pr[create wave]. In other words, a wave is created with probability Pr[create wave]. If upwards, the 1 continues to move diagonally upwards until it hits the top. Along the top at each iteration, the 1 is moved down with probability Pr[crash wave] and copied along the top with probability 1 − Pr[crash wave]. In other words, a wave is crashed with probability Pr[crash wave].



Waves circuit

For seed label 320 the programmed probabilities were Pr[create wave] = 0.1 and Pr[crash wave] = 0.5. For seed label 321 the programmed probabilities were Pr[create wave] = 0.5 and Pr[crash wave] = 0.5. Randomisation was implemented using the "first-tile randomisation" method described in Section S2.3.2.

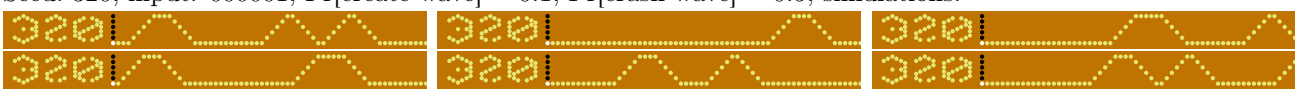
Validation and statistics. 69.3 % of seeds had significant growth (61 seeds with significant growth, 27 seeds without). The measured total amount of tile growth for all Waves nanotubes was 14,922 nm (excludes the 21-circuit dataset mean seed length of 155.6±0.18 nm) giving an estimated 33,438 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 10 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of 0.03 % ± 0.009.

Additional data specific to the Waves circuit was analysed as follows. The mean distance per nanotube without a wave, that is where the 1 is being copied along the bottom, was computed as the mean of the lengths of the set of line segments along the bottom per nanotube. The mean wave-top distance per nanotube, that is where the 1 is being copied along the top, was measured as the mean of the lengths of the set of line segments along the top per nanotube.

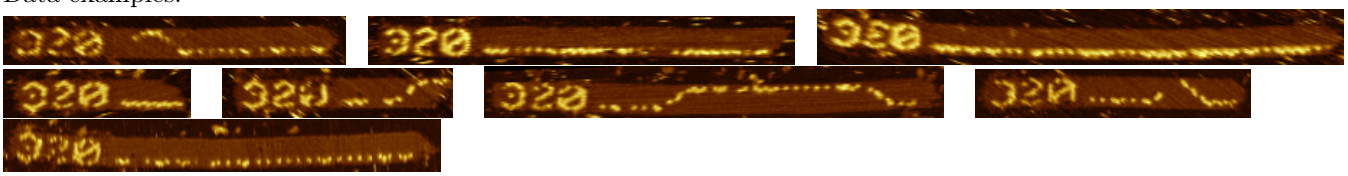
Theoretical analysis. We model the length of wave troughs/peaks (meaning the length of the consecutive sequence of 1's along the bottom/top row, respectively) as a geometric random variable with parameter p [109, Chapter 2], where p is the probability of creating/crashing a wave, respectively. This has expected number of IBC layer iterations $\frac{1}{p}$. Since each IBC layer iteration is estimated as 14.28 nm, the expected length is $\frac{14.28}{p}$ nm. The variance of a geometric random variable with parameter p is $\frac{1-p}{p^2}$, so standard deviation $\frac{\sqrt{1-p}}{p}$. In the table below, both sample mean length (measured from data) and expected length (predicted from theory), as well as sample standard deviation (sample stddev) and theoretical standard deviation (theoretical stddev), are shown. For sample mean m, the standard error of the mean e is shown as $m \pm e$.

barcode	**320**	**321**
Prob[create wave]	0.1	0.5
sample mean length bottom	193.2 ± 25.3 nm	42.9 ± 4.6 nm
expected length bottom	142.8 nm	28.6 nm
sample stddev length bottom	157.8 nm	34.8 nm
theoretical stddev length bottom	135.5 nm	20.2 nm
Prob[crash wave]	0.5	0.5
sample mean length top	62.6 ± 30.1 nm	37.3 ± 5.6 nm
expected length top	28.6 nm	28.6 nm
sample stddev length top	73.7 nm	33.2 nm
theoretical stddev length top	20.2 nm	20.2 nm

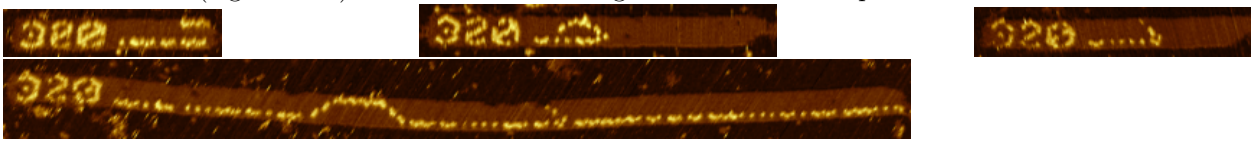
Seed: 320, input: 000001, Pr[create wave] = 0.1, Pr[crash wave] = 0.5, simulations:



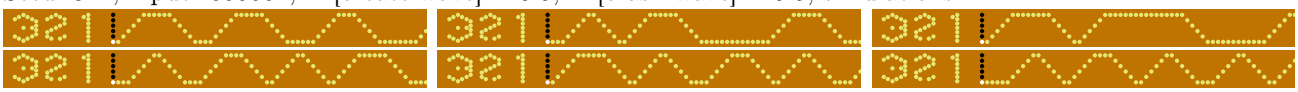
Data examples:



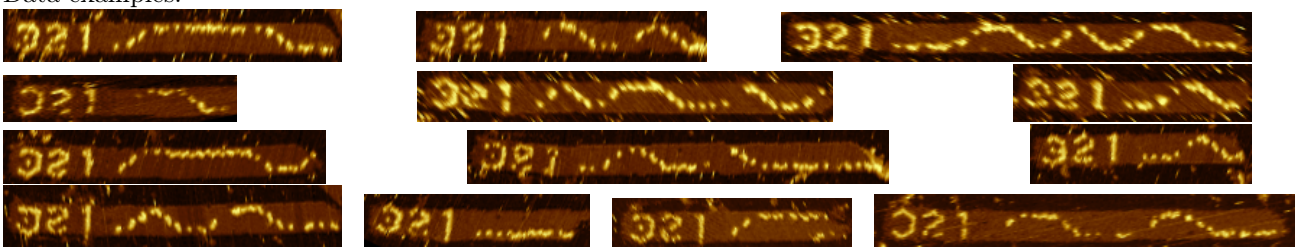
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:



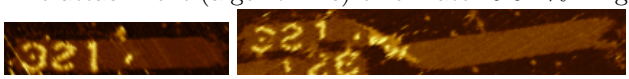
Seed: 321, input: 000001, Pr[create wave] = 0.5, Pr[crash wave] = 0.5, simulations:



Data examples:

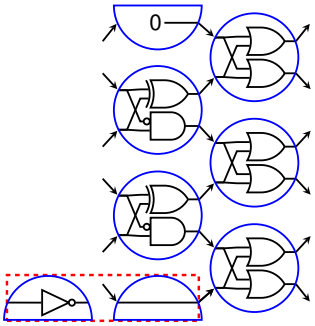


Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:



S8.13 Circuit: Rule110Random

Intuition and motivation. The Rule110Random circuit simulates the rule 110 cellular automaton, but with the bottom IBC gate providing a random 0 bit (with probability 0.5) or 1 bit (also with probability 0.5) at each iteration. Besides the randomisation feature the circuit is the same as Rule110, which was explained in Figure S4.



Rule110Random circuit

Intuitively, the randomisation feature was designed to give the impression that each IBC computation is simulating a thin "slice" out of some space-time diagram of rule 110 (although presumably some of the IBC computations would never arise in any space-time diagram). It is perhaps instructive to think of the sequence of random bits chosen along the bottom gate as a "random input" to the circuit: at each iteration a random bit is given. Indeed, the Rule110Random IBC could be written using a formalisation similar to that of the global-input IBC model (Section S1.3.4), where a Rule110Random IBC computation with t-iterations is instead thought of as a t-layer global-input IBC that gets one input bit at each iteration, with the input bits being randomly selected from the uniform distribution.

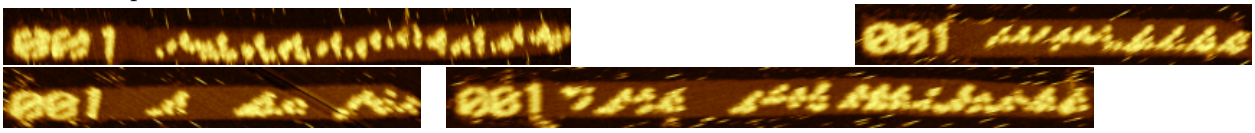
Randomisation was implemented using the "block randomisation" method described in Section S2.3.2.

Validation and statistics. 68.2 % of seeds had significant growth (161 seeds with significant growth, 75 seeds without). The measured total amount of tile growth for all Rule110Random nanotubes was 50,650 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 113,502 tiles attached (assuming a DNA base pair length of 0.34 nm). Tile attachment errors (algorithmic errors) were not analysed for this circuit.

The 3 inputs shown below represent a full coverage test for the Rule110Random circuit in the sense that correct computation on all of those inputs requires incorporation of every tile type that would be incorperated on all of the possible 64 inputs.

Seed: 001, input: 000001, simulations:

Data examples:



Seed: 110, input: 010100, simulations:

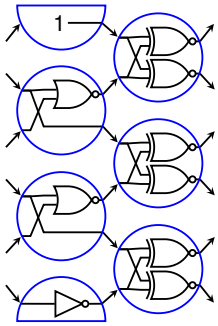
Data examples:

Seed: 301, input: 110001, simulations:

Data examples:

S8.14 Circuit: RULE30

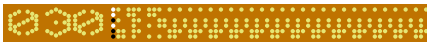
Intuition and motivation. The RULE30 circuit was found via a computer search of elementary CA whose local update function $f(x, y, z)$ can be expressed as $f(g(x, y), h(y, z))$ as was used in the proof of Theorem S1.1.



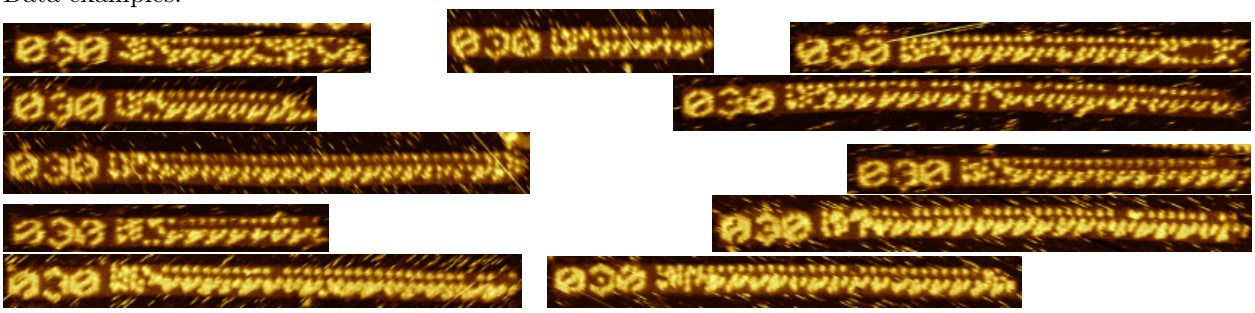
RULE30 circuit

Validation and statistics. 77.5 % of seeds had significant growth (158 seeds with significant growth, 46 seeds without). The measured total amount of tile growth for all RULE30 nanotubes was 42,324 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 94,843 tiles attached (assuming a DNA base pair length of 0.34 nm). Tile attachment errors (algorithmic errors) were not analysed for this circuit.

The 1 inputs shown below represent a full coverage test for the RULE30 circuit in the sense that correct computation on all of those inputs requires incorporation of every tile type that would be incorperated on all of the possible 64 inputs.

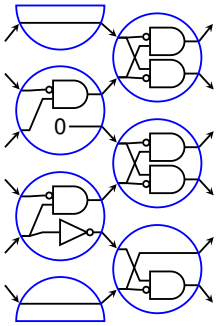
Seed: 030, input: 110100, simulation: 

Data examples:



S8.15 Circuit: Drumlins

Intuition and motivation. The Drumlins circuit makes a simple repeating pattern, reminiscent of hills in Co. Monaghan, Ireland and other locations around the world. This was one of the final circuits we designed and was created for tile coverage reasons: to ensure that every tile was used in at least one circuit.

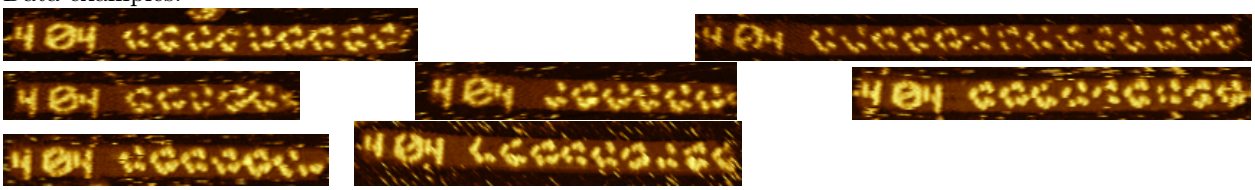


Drumlins circuit

Validation and statistics. 72.9 % of seeds had significant growth (70 seeds with significant growth, 26 seeds without). The measured total amount of tile growth for all Drumlins nanotubes was 14,037 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 31,455 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 9 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.03\ \% \pm 0.01$.

Seed: 404, input: 000010, simulation: 

Data examples:

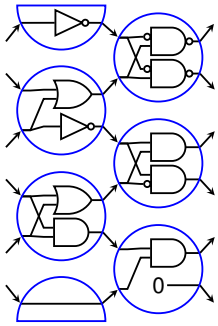


Tile attachment (algorithmic) error rate: 0.03 %. Algorithmic error examples:

S8.16 Circuit: 2Eggs

Intuition and motivation. To make an omelette you gotta break some eggs.

The 2Eggs circuit was designed for the purpose of tile coverage at a time point where most other circuits had been designed and implemented and there were a few left-over tiles that had not been used in any experiment. The omelette was full tile coverage, and the eggs were the tiles in yet-to-be-cracked test tubes hiding in the back of the freezer.

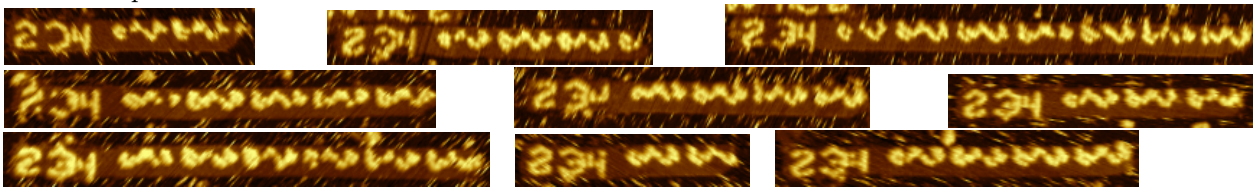


2Eggs circuit

Validation and statistics. 38.8 % of seeds had significant growth (33 seeds with significant growth, 52 seeds without). The measured total amount of tile growth for all 2Eggs nanotubes was 7,705 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 17,267 tiles attached (assuming a DNA base pair length of 0.34 nm). Tile attachment errors (algorithmic errors) were not analysed for this circuit.

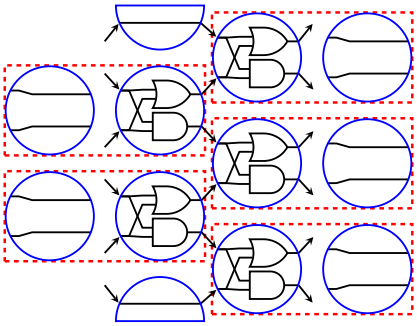
Seed: 234, input: 100000, simulation: 

Data examples:



S8.17 Circuit: LazySorting

Intuition and motivation. The randomised circuit LazySorting[48] sorts its input bits, so that the 1's end up on the top and the 0's end up on the bottom, thus computing the same function as the deterministic Sorting circuit. However, LazySorting works by, at each gate, (a) copying bits that are in order and (b) swapping or copying with equal probability if the bits are out of order. Hence the circuit sometimes copies, and sometimes sorts. This process has the effect of slowing down the sorting process but also giving a nice visualisation of the computation history of a randomised algorithm.



LazySorting circuit

Randomisation was implemented using the "block randomisation" method described in Section S2.3.2, by mixing the Copy and Sorting tiles in equal amounts (concentrations).

LazySorting computes a function, as defined in Section S1.1.1.

Validation and statistics. 59.3 % of seeds had significant growth (334 seeds with significant growth, 229 seeds without). The measured total amount of tile growth for all LazySorting nanotubes was 62,809 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 140,748 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 48 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.03\ \% \pm 0.005$.

The 3 inputs shown below represent a full coverage test for the LazySorting circuit in the sense that a large enough set of correct randomised computations on these inputs requires incorporation of every tile type for the LazySorting circuit.

A total of 111 of the analysed nanotubes (i.e. ~ 33 %) were sorted before the end of the growth of the nanotube and the rest were not. For those that completed the sorting process, they did so in mean distance and standard error of the mean 158.2 ± 7.9 nm per nanotube. For those that did not complete the sorting process, the mean length per nanotube and standard error of the mean was 128.7 ± 5.0 nm.

All randomised gate probabilities were 0.5. The barcode values 001, 011, and 013 correspond to respective inputs 000001, 000101 and 000111. The respective mean distances and standard error of the mean to complete sorting were 148.0 ± 10.5 nm, 157.3 ± 14.0 nm and 192.4 ± 20.2 nm.

In the LazySorting data, more computations than not failed to finish by the end of the nanotube. However, completion of sorting was not the main goal here — if one wanted faster sorting one can simply use the (deterministic) Sorting (Section S8.1) circuit for which almost all nanotubes complete the sorting process before growth ends. LazySorting was our first attempt at a randomised circuit and gave us confidence to design more. Like other randomised circuits, LazySorting has the nice property that the assembled nanotube contains within it a visual "tape-recording" of a sequence of randomised events.

Theoretical analysis. We analyze only the case of sorting input 000001. Say a half-layer has *odd parity* if it has 1-input 1-output gates in it, and it has *even parity* otherwise. The 1 starts on wire 6 and a half-layer of odd parity. Note that it can only move up to wire 5 on the other half layer of even parity. In general, the 1 can move up if it goes through a half-layer of the same parity to its wire. We distinguish the first move up, from wire 6 to 5, from the remaining moves. After starting, the number of half-layers before the first move up from wire 6 to 5 will take a positive even number of half-layers, i.e., a positive integer number of full layers. The number of full layers is described by a geometric random variable with Pr[success] = 0.5, having expected value 2. So the expected number of half-layers is 4.

In the other cases, we measure the number of half-layers in between the previous move up and the next move up. Unlike the first case, immediately after moving up, the wire and half-layer are the same parity. So, the move up could occur immediately (after 1 half-layer). If not, the next would be after 3 half-layers, then 5, etc. The random variable X describing the number of half-layers before the next move up is not quite a geometric random variable, but we can analyze it similarly. For all $n \in \mathbb{N}^+$, let $p_n = \Pr[X \geq n]$. Since X cannot take even values, we have $p_1 = 1$, $p_2 = p_3 = 0.5$, $p_4 = p_5 = 0.5^2$, $p_6 = p_7 = 0.5^3$, ...

[48] We thank Paul Rothemund for suggesting the name LazySorting.

One way to characterize expected value of a variable X taking values in range R (see [109, Lemma 2.9]) is $\mathrm{E}[X] = \sum_{n \in R} \Pr[X \geq n]$. Thus

$$\begin{aligned} \mathrm{E}[X] &= p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + \ldots \\ &= p_1 + 2 \cdot p_3 + 2 \cdot p_5 + 2 \cdot p_7 + \ldots \\ &= 1 + 2 \cdot \sum_{i=1}^{\infty} 0.5^i = 3. \end{aligned}$$

Similarly, let $q_n = \Pr[X^2 \geq n]$. Then $q_1 = 1$, $q_2 = q_3 = \ldots = q_9 = 0.5$, $q_{10} = q_{11} = \ldots = q_{25} = 0.5^2$, $q_{26} = q_{27} = \ldots = q_{49} = 0.5^3$, ... Thus

$$\begin{aligned} \mathrm{E}[X^2] &= q_1 + q_2 + \ldots \\ &= q_1 + 8 \cdot q_9 + 16 \cdot q_{25} + 24 \cdot q_{49} + 32 \cdot q_{81} + \ldots \\ &= 1 + 8 \cdot \sum_{i=1}^{\infty} i \cdot 0.5^i \\ &= 1 + 8 \cdot 2 = 17. \end{aligned}$$

Thus $\mathrm{Var}[X] = \mathrm{E}[X^2] - \mathrm{E}[X]^2 = 17 - 9 = 8$.

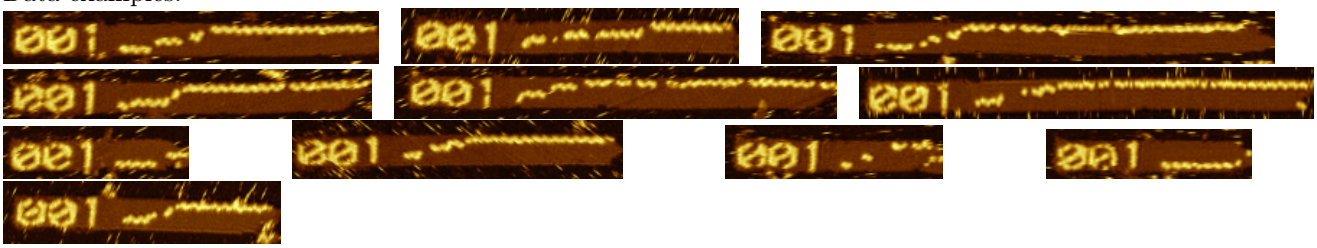
Let X_1, X_2, X_3, X_4 be four independent random variables with this distribution, and let F the random variable describing the number of half layers before the *first* move up (recall $F = 2G$, where G is a geometric random variable with $\Pr[\text{success}] = 0.5$). For G a geometric random variable with $\Pr[\text{success}] = 0.5$, it is known that $\mathrm{Var}[G] = 2$, so $\mathrm{Var}[F] = 4 \cdot \mathrm{Var}[G] = 8$. Then the number of half-layers until the 1 reaches from wire 6 to wire 1 (i.e., is sorted) is $L = F + X_1 + X_2 + X_3 + X_4$. By linearity of expectation, $\mathrm{E}[L] = \mathrm{E}[F] + \mathrm{E}[X_1] + \mathrm{E}[X_2] + \mathrm{E}[X_3] + \mathrm{E}[X_4] = 4 + 3 + 3 + 3 + 3 = 16$. Since each half-layer is 7.14 nm, translating from half-layers to length in nm, this is $16 \cdot 7.14$ nm $= 114.24$ nm. Since the variables are independent, variance is additive, so $\mathrm{Var}[L] = 8 + 8 + 8 + 8 + 8 = 40$, so standard deviation $\sqrt{40} = 6.32$. Translating from half-layers to length in nm, this is $6.32 \cdot 7.14$ nm $= 45.1248$ nm. These are compared to the experimentally sampled mean length until the 1 reaches the top wire, and standard deviation, for input 000001 in the following table.

sample mean length	148.0 ± 10.5 nm
expected length	114.24 nm
sample stddev length	82.7 nm
theoretical stddev length	45.1248 nm

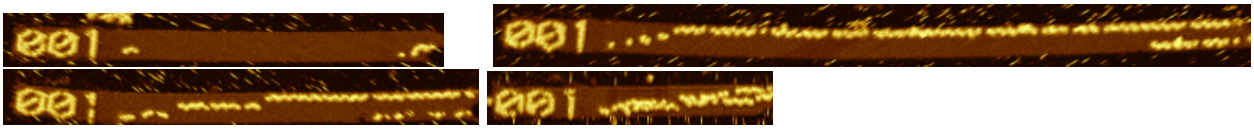
The random process for sorting inputs with multiple 1's is not as straightforward, since the movement of 1's is not independent, so we omit a formal analysis of inputs 000101 and 000111.

Seed: 001, input: 000001, output: 100000, simulations:

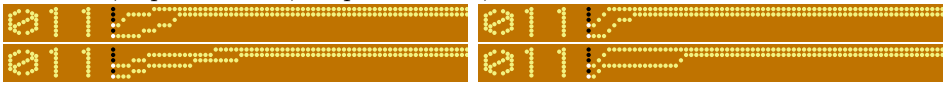
Data examples:



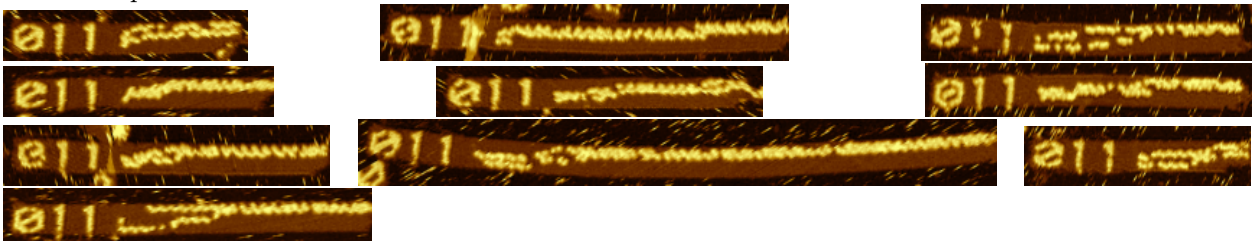
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:



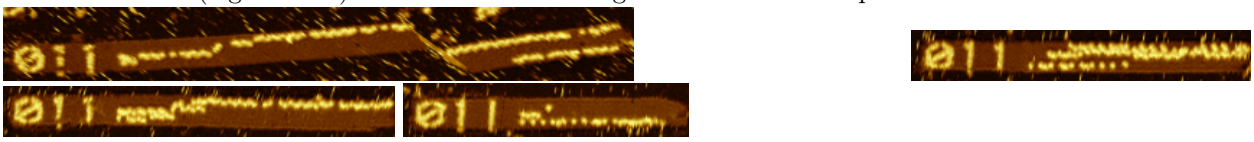
Seed: 011, input: 000101, output: 110000, simulations:



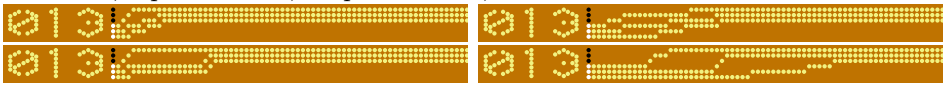
Data examples:



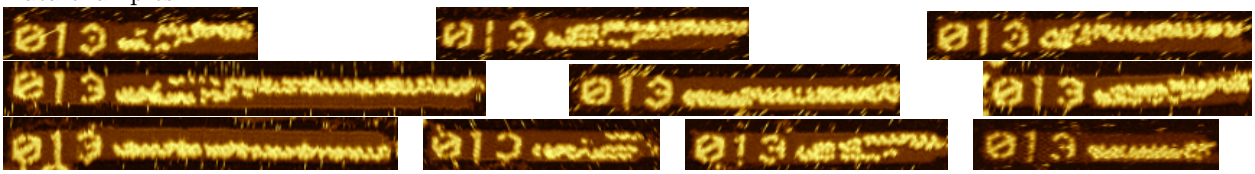
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:



Seed: 013, input: 000111, output: 111000, simulations:



Data examples:



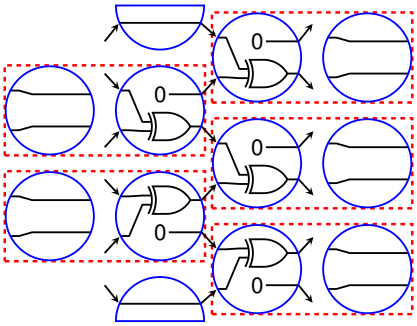
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:

S8.18 Circuit: LazyParity

Intuition and motivation. The randomised circuit LazyParity determines whether the number of 1's in the input is even or odd, thus solving the same decision problem as the deterministic Parity circuit. LazyParity works by, at each gate position with equal probability, either (a) copying both input bits or (b) computing the "parity functions" described for the Parity circuit (move 1's to the middle, if the two 1's meet they both flip to 0). Hence the circuit sometimes copies, and sometimes computes parity, having the effect of slowing down the parity decision process.

Randomisation was implemented using the "block randomisation" method described in Section S2.3.2.

LazyParity decides a language, as defined in Section S1.1.1.



LazyParity circuit

Validation and statistics. 48.8 % of seeds had significant growth (126 seeds with significant growth, 132 seeds without). The measured total amount of tile growth for all LazyParity nanotubes was 30,629 nm (excludes the 21-circuit dataset mean seed length of 155.6±0.18 nm) giving an estimated 68,636 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 8 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of 0.01 % ± 0.004.

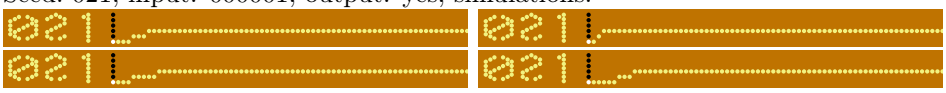
The 5 inputs shown below represent a full coverage test for the LazyParity circuit in the sense that a large enough set of correct randomised computations on these inputs requires incorporation of every tile type for the LazyParity circuit. All randomised gate probabilities were 0.5. Out of those 118 nanotubes that made a decision before the end of nanotube growth, 76.3 % made a correct decision and the rest did not. For those that completed the parity decision process, they did so in mean distance and standard error of the mean 82.0 ± 4.8 nm per nanotube. For those did not complete the parity decision process, the mean and length of tile growth per nanotube, and standard error of the mean, was 45.8 ± 7.0 nm.

The measured respective mean distance and standard error of the mean from the input-adapter strands to the earliest point with a parity decision for barcodes 021, 022, 023, 024 and 020, was, respectively, 80.8 ± 13.7 nm, 80.9 ± 10.1 nm, 92.4 ± 7.4 nm, 82.2 ± 9.3 nm and 64.7 ± 10.5 nm, showing no strong correlation between number (say) of 1's in the input bit-string and nanotube length to make a decision.

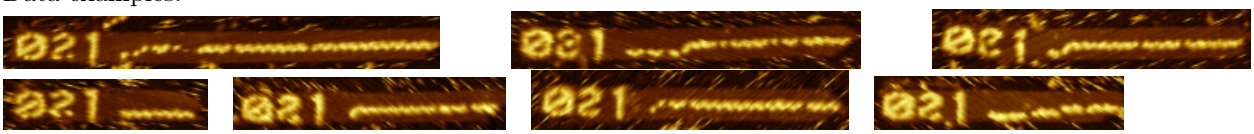
Theoretical analysis. As with the LazySorting circuit, the random process governing the time for the LazyParity circuit to reach a decision is complex, with the movement or cancellation of the 1's being heavily dependent on the positions/movement of the others. Thus, as with LazySorting, we omit a formal analysis except in the simplest case of input 000001. In this case, the decision is reached when the 1 bit at wire 6 reaches wire 4 (recall the analysis of LazySorting analysed the case of the 1 bit at wire 6 reaching wire 1). By a similar analysis, the number of half-layers until the 1 reaches from wire 6 to wire 4 (i.e., the parity is computed) is $L = F + X$, where $F = 2G$ and G is a geometric random variable with Pr[success] = 0.5, and X is a random variable distributed equivalently to X_1 in the analysis of LazySorting. By linearity of expectation, $\mathrm{E}[L] = \mathrm{E}[F] + \mathrm{E}[X] = 4 + 3 = 7$. Since each half-layer is 7.14 nm, translating from half-layers to length in nm, this is $7 \cdot 7.14$ nm = 49.98 nm. Similarly to the analysis of LazySorting, we have $\mathrm{Var}[L] = 8 + 8 = 16$, so standard deviation $\sqrt{16} = 4$. Translating from half-layers to length in nm, this is $4 \cdot 7.14$ nm = 28.56 nm. These are compared to the experimentally sampled mean length until the 1 reaches wire 4, and standard deviation, for input 000001 in the following table.

sample mean length	80.8 ± 13.7 nm
expected length	49.98 nm
sample stddev length	62.9 nm
theoretical stddev length	28.56 nm

Seed: 021, input: 000001, output: yes, simulations:



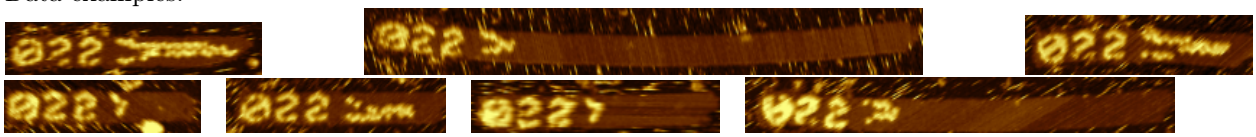
Data examples:



Tile attachment (algorithmic) error rate: 0.01 %. No algorithmic error examples.

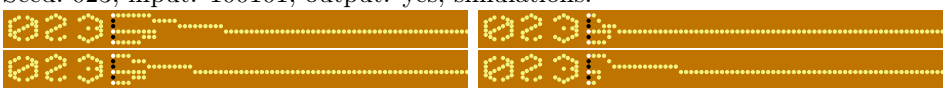
Seed: 022, input: 100001, output: no, simulations:

Data examples:

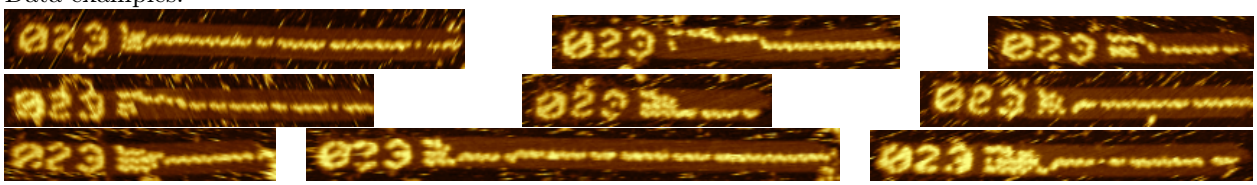


Tile attachment (algorithmic) error rate: 0.00 %. No algorithmic error examples.

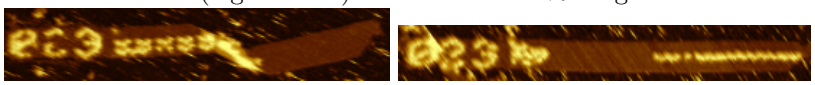
Seed: 023, input: 100101, output: yes, simulations:



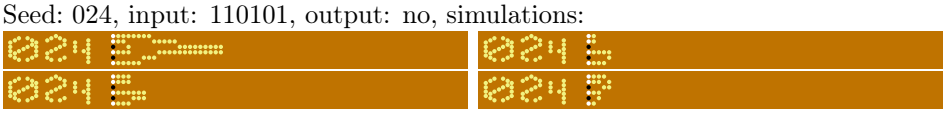
Data examples:



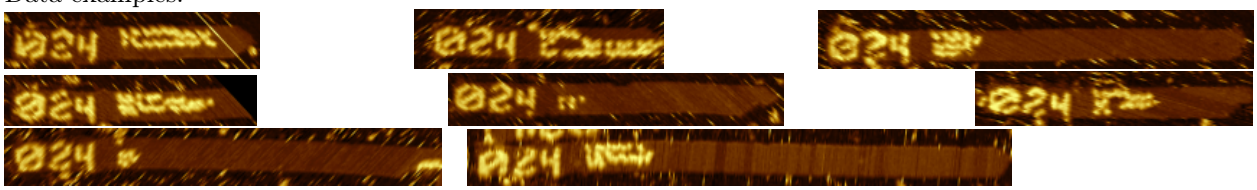
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error examples:



Seed: 024, input: 110101, output: no, simulations:



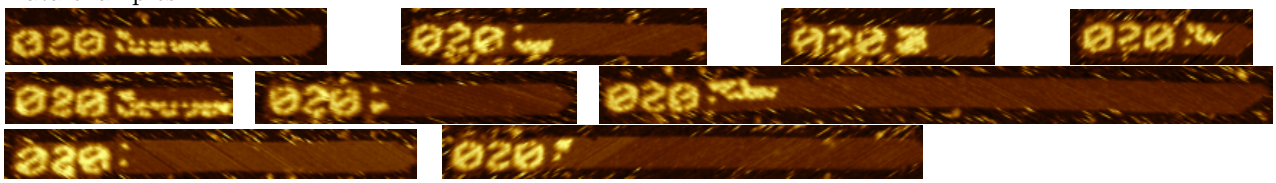
Data examples:



Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:

Seed: 020, input: 110011, output: no, simulations:

Data examples:

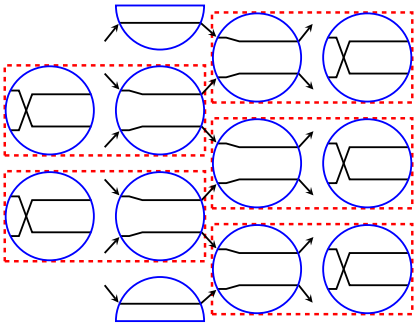


Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error example:

S8.19 Circuit: RandomWalkingBit

Intuition and motivation. In the RandomWalkingBit circuit[49] any 1 bit that is input to a 2-input 2-output gate may move forward on the same wire with probability 0.5 or to swap to an adjacent wire with the same probability.

Randomisation was implemented using the "block randomisation" method described in Section S2.3.2.

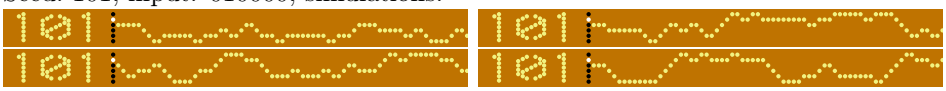


RandomWalkingBit circuit

Validation and statistics. 68.7 % of seeds had significant growth (191 seeds with significant growth, 87 seeds without). The measured total amount of tile growth for all RandomWalkingBit nanotubes was 63,107 nm (excludes the 21-circuit dataset mean seed length of 155.6±0.18 nm) giving an estimated 141,416 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 48 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of 0.03 % ± 0.005.

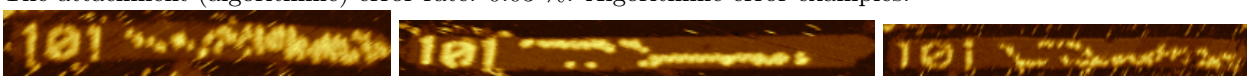
The 4 inputs shown below represent a full coverage test for the RandomWalkingBit circuit in the sense that a large enough set of correct randomised computations on these inputs requires incorporation of every tile type for the RandomWalkingBit circuit. There appears to be a bias for the random walking 1 to move toward the bottom. This could be due to the effective concentrations of some tiles being lower or higher than nominal values, or it could be due to some binding domains being stronger than others. We did not investigate these hypotheses, however.

Seed: 101, input: 010000, simulations:

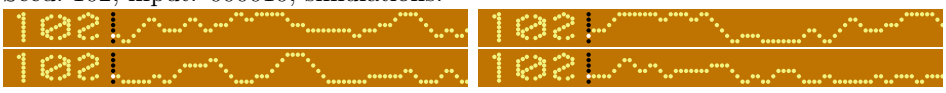


Data examples:

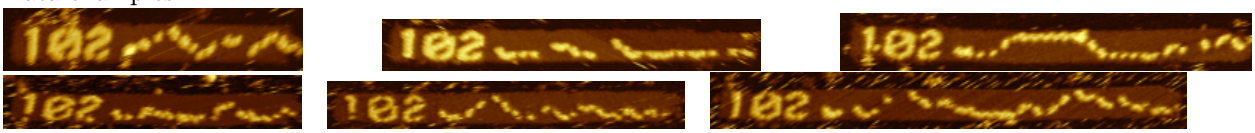
Tile attachment (algorithmic) error rate: 0.03 %. Algorithmic error examples:



Seed: 102, input: 000010, simulations:

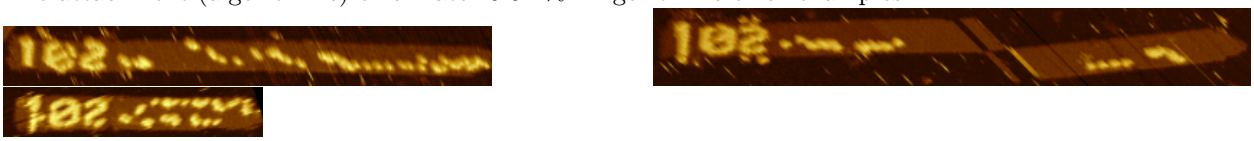


Data examples:

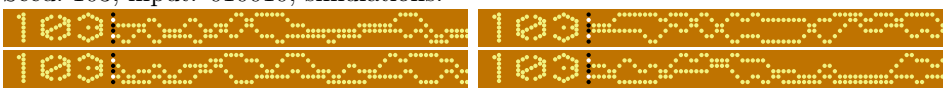


[49] We thank Cody Geary for suggesting this circuit.

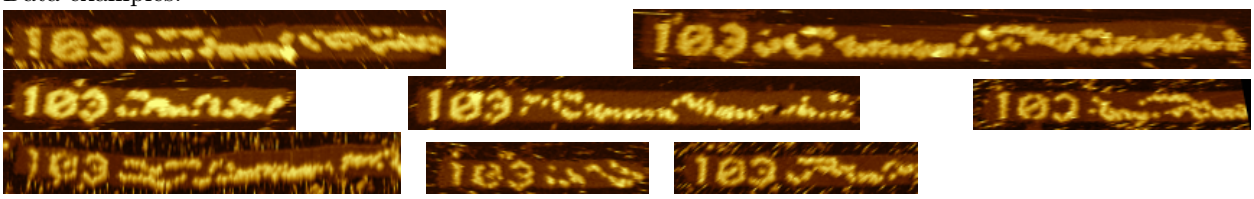
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:



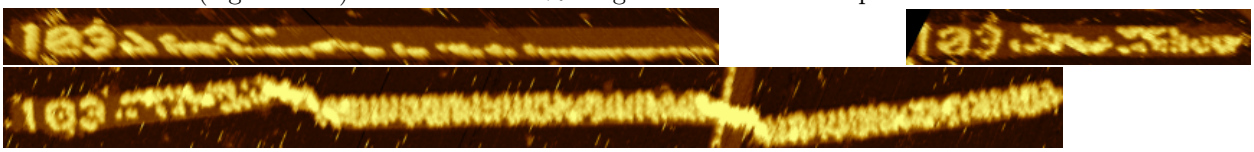
Seed: 103, input: 010010, simulations:



Data examples:

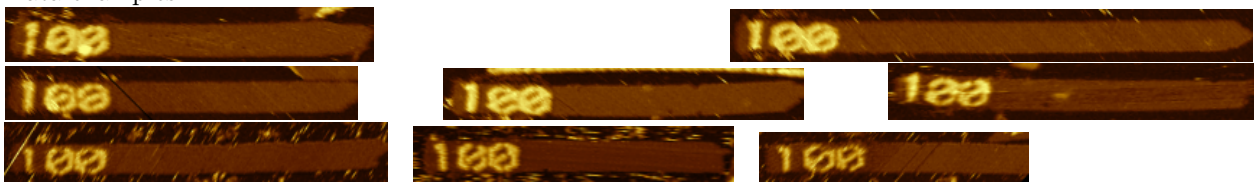


Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error examples:

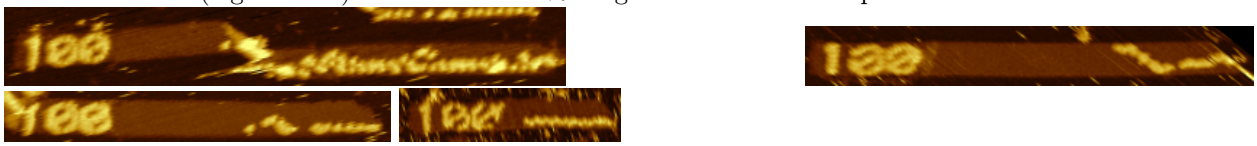


Seed: 100, input: 000000, simulations:

Data examples:



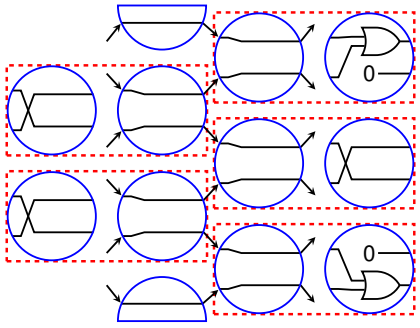
Tile attachment (algorithmic) error rate: 0.03 %. Algorithmic error examples:



S8.20 Circuit: AbsorbingRandomWalkingBit

Intuition and motivation. The AbsorbingRandomWalkingBit circuit starts from an input with five 0's and a single 1. The 1 randomly chooses in each layer to go up one row or down one row, unless it is in the top or bottom row, in which case it "absorbed to", or stays on, in that row.

Randomisation was implemented using the "block randomisation" method described in Section S2.3.2.



AbsorbingRandomWalkingBit circuit

Validation and statistics. 73.6 % of seeds had significant growth (187 seeds with significant growth, 67 seeds without). The measured total amount of tile growth for all AbsorbingRandomWalkingBit nanotubes was 90,267 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 202,279 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 86 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.04\ \% \pm 0.004$.

For this circuit $p = 0.5$ was used, i.e., equal probability for the 1 to move up or down. We measured 176 nanotubes where the randomly walking 1 had absorbed to the top or bottom.

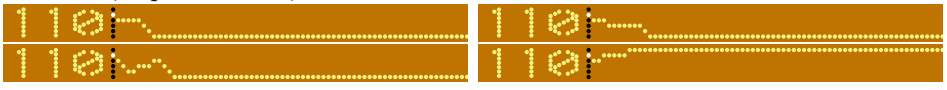
Barcodes 110, 111, 112, and 113 all had inputs with a single 1, increasing barcode number puts the 1 closer to the bottom, respectively: 010000, 001000, 000100 and 000010. It is tempting to think that the movement of the 1 until it hits a boundary can be modeled as an unbiased random walk on the wires 1 through 6. However, it is a more complex process than a standard random walk: a 1 entering the bottom (respectively, top) input of a gate can either stay in place or move up (resp. down). Therefore we omit a formal analysis of the process.

From the annotated data, the results by barcode were as follows. Recall that, as defined in Section S8.0.2, $x \pm y$ denotes a sample mean x with standard error of the mean y.

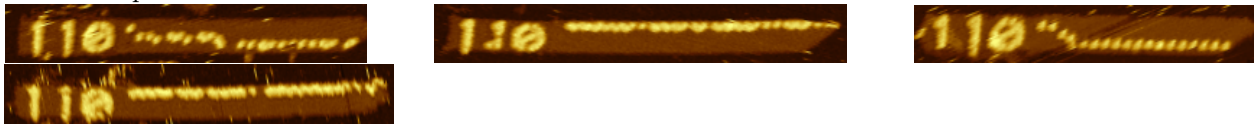
barcode	110	111	112	113
input	010000	001000	000100	000010
nanotubes analysed (sig. growth)	25	35	88	39
number of 1's absorbed	23	33	81	39
absorbed to top (measured)	69.57 %	12.12 %	18.52 %	2.56 %
top: mean absorbed distance	22.8 ± 3.2 nm	57.8 ± 5.6 nm	74.1 ± 14.0 nm	248.3 ± 0.0 nm
bottom: mean absorbed distance	57.0 ± 11.9 nm	58.8 ± 4.6 nm	54.6 ± 4.3 nm	28.7 ± 2.2 nm

Thus, in the data, input 010000 was biased to the top and 000100, 000010 were biased to the bottom, qualitatively agreeing with intuition, although quantitatively the frequency was not close to predicted. However, input 001000 was strongly biased to go to the bottom, disagreeing with our somewhat simplified theoretical model. Note that the bias toward the bottom may be due to the same (unknown) cause of the bias in the RandomWalkingBit circuit.

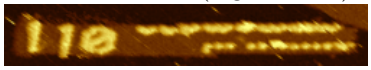
Seed: 110, input: 010000, simulations:



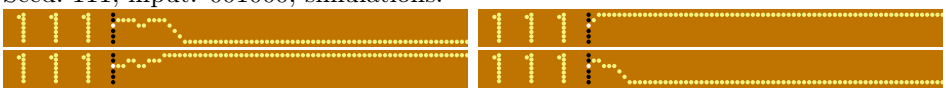
Data examples:



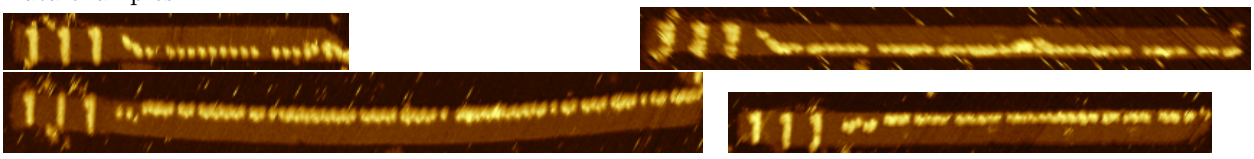
Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error example:



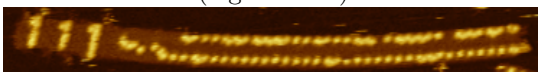
Seed: 111, input: 001000, simulations:



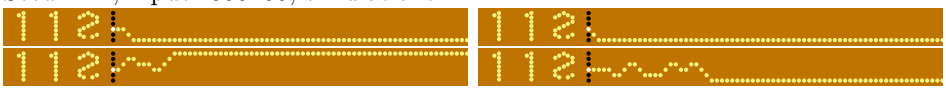
Data examples:



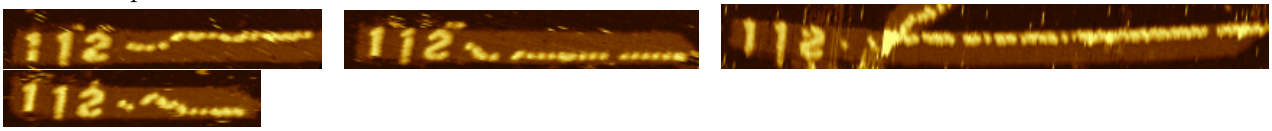
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error example:



Seed: 112, input: 000100, simulations:

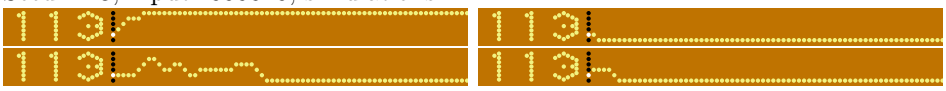


Data examples:



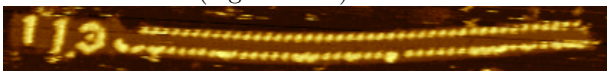
Tile attachment (algorithmic) error rate: 0.04 %. Algorithmic error example:

Seed: 113, input: 000010, simulations:



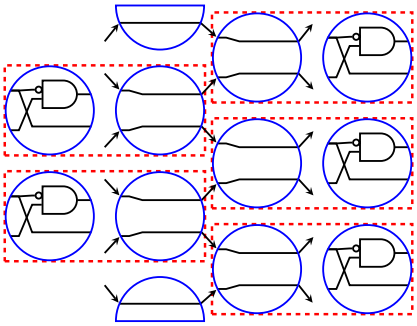
Data examples:

Tile attachment (algorithmic) error rate: 0.06 %. Algorithmic error example:



S8.21 Circuit: LeaderElection

Intuition and motivation.
"There can be only one" — The Kurgan, Highlander


LeaderElection circuit

The LeaderElection circuit solves a variant of the *leader election* problem in distributed computing [111], that of getting several agents in a network to identify a unique agent as the "leader". The circuit takes as an input a string with one or more 1s and after some "voting rounds" ends with a single, randomly walking, 1 called the *leader*. The circuit is designed as follows: if two 1s are given as input to a gate then with probability 0.5 they are simply copied, and with probability 0.5 one of them (the top one) is changed to 0. An adjacent pair of 0 and 1 are copied (with probability 0.5) or swapped (with probability 0.5), making 1's take a random walk up and down as they move form left to right. After the leader (a single 1) is elected, it randomly walks forever, assuming no errors occur.

The circuit is robust to errors in the following sense: if after electing a leader, a second leader appears due to an error, then another round of leader election will eventually take place. If a leader ever disappears, then since there is a non-zero error rate, eventually a new leader will be born.

Randomisation was implemented using the "block randomisation" method described in Section S2.3.2.

Validation and statistics. 78.3 % of seeds had significant growth (130 seeds with significant growth, 36 seeds without). The measured total amount of tile growth for all LeaderElection nanotubes was 53,540 nm (excludes the 21-circuit dataset mean seed length of 155.6 ± 0.18 nm) giving an estimated 119,978 tiles attached (assuming a DNA base pair length of 0.34 nm). We counted 14 erroneous tile attachments, implying a mean tile attachment error rate and standard error of the mean of $0.01\ \% \pm 0.003$.

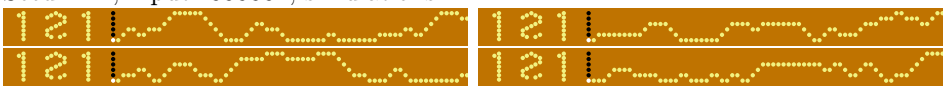
The 3 inputs shown below represent a full coverage test for the LeaderElection circuit in the sense that a large enough set of correct randomised computations on these inputs requires incorporation of every tile type for the LeaderElection circuit.

On each nanotube that elected a single leader we measured the distance from the input-adapters to the point of leader election. If no leader was elected, we measured the distance from the input-adapters to the end of the leaderless nanotube. After electing a leader, in some rare cases an algorithmic error created a new 1 (creating a potential leader candidate) which may go on to force another leadership contest before the end of the nanotube; although we counted the algorithmic error in such cases we did not measure/annotate the subsequent leadership contests as we felt they would complicate our analysis.

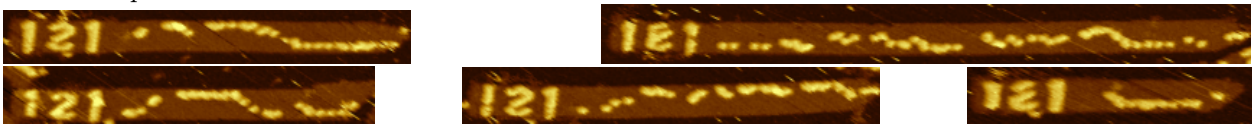
The mean distance before a leader was elected, and standard error of this mean, was 86.5 ± 6.8 nm, for those 120 nanotubes that did so. 7 nanotubes did not elect a leader by the end of the nanotube; their total length had mean and standard error of the mean 158.5 ± 40.5 nm.

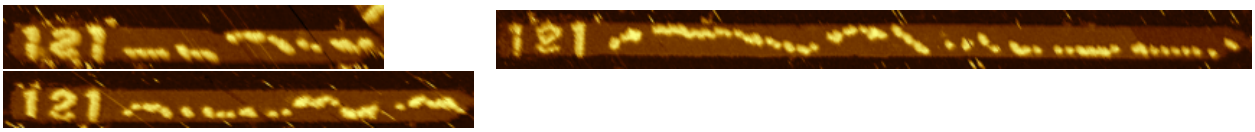
We omit a formal theoretical analysis of the random process, since it is not straightforward to analyse even for two leaders. With an increasing number of 1's in the 6-bit input we saw an increasing distance to elect a leader – as expected more leadership battles require more time and distance on average. Mean distances and standard error of the mean to elect a leader were as follows: for barcode 121 with input 000001: 21.2 ± 4.2 nm (44 leaders elected); for barcode 122 with input 100001: 115.0 ± 11.6 nm (43 leaders elected); for barcode 122 with input 111111: 136.5 ± 8.7 nm (33 leaders elected).

Seed: 121, input: 000001, simulations:

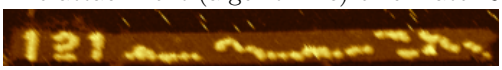


Data examples:

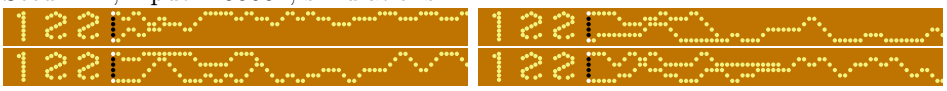




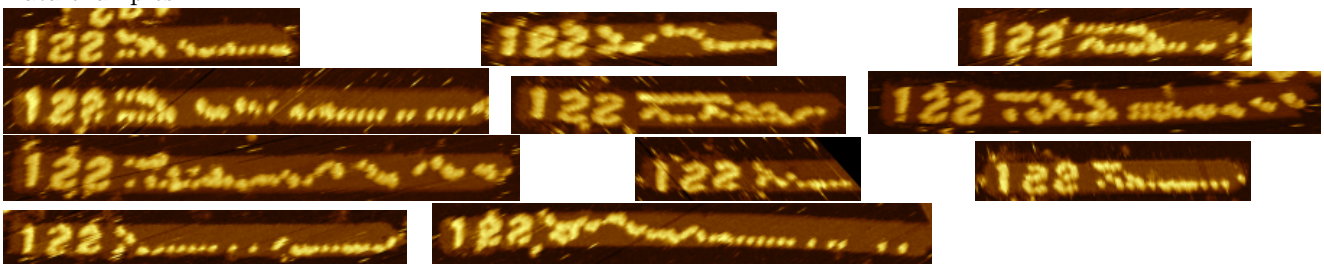
Tile attachment (algorithmic) error rate: 0.01 %. Algorithmic error example:



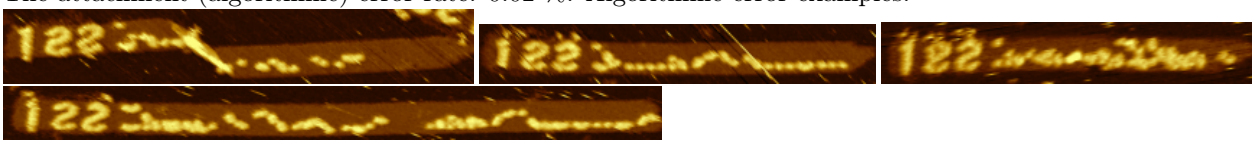
Seed: 122, input: 100001, simulations:



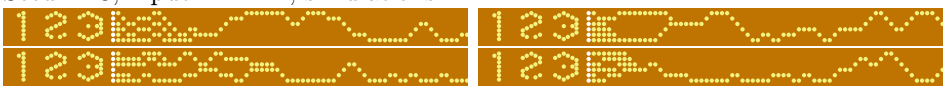
Data examples:



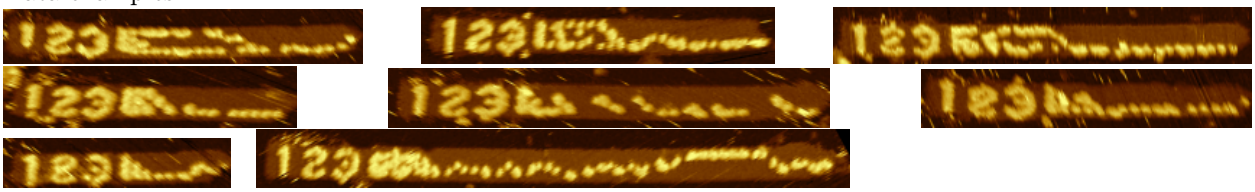
Tile attachment (algorithmic) error rate: 0.02 %. Algorithmic error examples:



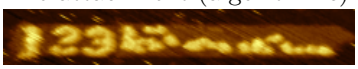
Seed: 123, input: 111111, simulations:



Data examples:



Tile attachment (algorithmic) error rate: 0.00 %. Algorithmic error example:



References

[1] George M Whitesides and Bartosz Grzybowski. Self-assembly at all scales. *Science*, 295(5564):2418–2421, 2002.

[2] Erik Winfree. Simulations of computing by self-assembly. Technical Report CaltechCSTR:1998.22, California Institute of Technology, 1998.

[3] David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55(12):78–88, December 2012.

[4] Nadrian C Seeman and Hanadi F Sleiman. DNA nanotechnology. *Nature Reviews Materials*, 3:17068, 2017.

[5] Chengde Mao, Thomas H LaBean, John H Reif, and Nadrian C Seeman. Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. *Nature*, 407(6803):493–496, 2000.

[6] Paul W K Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLoS Biology*, 2(12):e424, 2004.

[7] Rebecca Schulman and Erik Winfree. Synthesis of crystals with a programmable kinetic barrier to nucleation. *Proceedings of the National Academy of Sciences*, 104(39):15236–15241, 2007.

[8] Robert D Barish, Rebecca Schulman, Paul W K Rothemund, and Erik Winfree. An information-bearing seed for nucleating algorithmic self-assembly. *Proceedings of the National Academy of Sciences*, 106(15):6054–6059, 2009.

[9] Rebecca Schulman, Bernard Yurke, and Erik Winfree. Robust self-replication of combinatorial information via crystal growth and scission. *Proceedings of the National Academy of Sciences*, 109(17):6405–6410, 2012.

[10] Constantine G Evans. *Crystals that count! Physical principles and experimental investigations of DNA tile self-assembly*. PhD thesis, Caltech, 2014.

[11] Rebecca Schulman, Christina Wright, and Erik Winfree. Increasing redundancy exponentially reduces error rates during algorithmic self-assembly. *ACS Nano*, 9(6):5760–5771, 2015.

[12] Erik Winfree, Furong Liu, Lisa A Wenzler, and Nadrian C Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394(6693):539–544, 1998.

[13] Peng Yin, Rizal F Hariadi, Sudheer Sahu, Harry M T Choi, Sung Ha Park, Thomas H LaBean, and John H Reif. Programming DNA tube circumferences. *Science*, 321(5890):824–826, 2008.

[14] Paul W K Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[15] Bryan Wei, Mingjie Dai, and Peng Yin. Complex shapes self-assembled from single-stranded DNA tiles. *Nature*, 485(7400):623–626, 2012.

[16] Wen Wang, Tong Lin, Suoyu Zhang, Tanxi Bai, Yongli Mi, and Bryan Wei. Self-assembly of fully addressable DNA nanostructures from double crossover tiles. *Nucleic Acids Research*, 44(16):7989–7996, 2016.

[17] L L Ong, N Hanikel, O K Yaghi, C Grun, M T Strauss, P Bron, J Lai-Kee-Him, F Schueder, B Wang, P Wang, J Y Kishi, C A Myhrvold, A Zhu, R Jungmann, G Bellot, Y Ke, and P Yin. Programmable self-assembly of three-dimensional nanostructures from 10^4 unique components. *Nature*, 552:72–77, 2017.

[18] Paul W K Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *STOC: Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 459–468. ACM, 2000.

[19] David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007.

[20] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[21] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[22] Turlough Neary and Damien Woods. P-completeness of cellular automaton Rule 110. In *ICALP 2006: International Colloquium on Automata, Languages, and Programming*, pages 132–143. Springer, 2006.

[23] Rebecca Schulman and Erik Winfree. Programmable control of nucleation for algorithmic self-assembly. *SIAM Journal on Computing*, 39(4):1581–1616, 2009.

[24] Constantine G Evans and Erik Winfree. Physical principles for DNA tile self-assembly. *Chemical Society Reviews*, 46(12):3808–3829, 2017.

[25] Erik Winfree and Renat Bekbolatov. Proofreading tile sets: Error correction for algorithmic self-assembly. In Junghuei Chen and John Reif, editors, *DNA9: Proceedings of the 9th International Conference on DNA Computing*, volume 2943 of *LNCS*, pages 126–144. Springer, 2004.

[26] Abdul M Mohammed and Rebecca Schulman. Directing self-assembly of DNA nanotubes using programmable seeds. *Nano Letters*, 13(9):4006–4013, 2013.

[27] Constantine G Evans and Erik Winfree. DNA sticky end design and assignment for robust algorithmic self-assembly. In David Soloveichik and Bernard Yurke, editors, *DNA19: Proceedings of the 19th International Conference on DNA Computing and Molecular Programming*, volume 8141 of *LNCS*, pages 61–75. Springer, 2013.

[28] Joseph N Zadeh, Conrad D Steenberg, Justin S Bois, Brian R Wolfe, Marshall B Pierce, Asif R Khan, Robert M Dirks, and Niles A Pierce. NUPACK: analysis and design of nucleic acid systems. *Journal of Computational Chemistry*, 32(1):170–173, 2011.

[29] Ronny Lorenz, Stephan H Bernhart, Christian Hoener Zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1):26, 2011.

[30] Bryan Wei, Luvena L Ong, Jeffrey Chen, Alexander S Jaffe, and Peng Yin. Complex reconfiguration of DNA nanostructures. *Angewandte Chemie*, 126(29):7605–7609, 2014.

[31] Cameron T Chalk, Bin Fu, Eric Martinez, Robert Schweller, and Tim Wylie. Concentration independent random number generation in tile self-assembly. *Theoretical Computer Science*, 667:1–15, 2017.

[32] William M Jacobs, Aleks Reinhardt, and Daan Frenkel. Rational design of self-assembly pathways for complex multicomponent structures. *Proceedings of the National Academy of Sciences*, 112(20):6313–6318, 2015.

[33] Lester O Hedges, Ranjan V Mannige, and Stephen Whitelam. Growth of equilibrium structures built from a large number of distinct component types. *Soft Matter*, 10(34):6404–6416, 2014.

[34] Alexander Graham Cairns-Smith. *Genetic takeover and the mineral origins of life*. Cambridge University Press, 1982.

[35] David Doty, Jack H Lutz, Matthew J Patitz, Robert T Schweller, Scott M Summers, and Damien Woods. The tile assembly model is intrinsically universal. In *FOCS: Proceedings of the 53rd Annual Symposium on the Foundations of Computer Science*, pages 302–310. IEEE, 2012.

[36] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1937.

[37] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[38] Simon C Knowles, John G McWhirter, Roger F Woods, and John V McCanny. Bit-level systolic architectures for high performance IIR filtering. *The Journal of VLSI Signal Processing*, 1(1):9–24, 1989.

[39] Mathilde Noual, Damien Regnault, and Sylvain Sené. About non-monotony in Boolean automata networks. *Theoretical Computer Science*, 504:12–25, 2013.

[40] Nicolas Ollinger. Universalities in cellular automata a (short) survey. In *JAC: Symposium on Cellular Automata Journées Automates Cellulaires*, pages 102–118, 2008.

[41] Damien Woods. Intrinsic universality and the computational power of self-assembly. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

[42] Matthew Cook. A concrete view of Rule 110 computation. In Proceedings International Workshop on *The Complexity of Simple Programs,* Cork, Ireland, 6-7th December 2008, volume 1 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–55. Open Publishing Association, 2009. arXiv:0906.3248.

[43] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601, 1983.

[44] Allan Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, 1977.

[45] José L Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural complexity II*, volume 22 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.

[46] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.

[47] David A Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274–306, 1990.

[48] Turlough Neary. *Small universal Turing machines*. PhD thesis, Maynooth University, 2008.

[49] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[50] Leslie M Goldschlager. The monotone and planar circuit value problems are log space complete for P. *ACM SIGACT News*, 9(2):25–29, 1977.

[51] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[52] Kristoffer Arnsfelt Hansen. Constant width planar computation characterizes ACC^0. *Theory of Computing Systems*, 39(1):79–92, 2006.

[53] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing. In *STOC: Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 209–213. ACM, 1979.

[54] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1996.

[55] Eric Goles, P-É Meunier, Ivan Rapaport, and Guillaume Theyssier. Communication complexity and intrinsic universality in cellular automata. *Theoretical Computer Science*, 412(1-2):2–21, 2011.

[56] Christoph Dürr, Ivan Rapaport, and Guillaume Theyssier. Cellular automata and communication complexity. *Theoretical Computer Science*, 322(2):355–368, 2004.

[57] Anne Condon. A theory of strict P-completeness. *Computational Complexity*, 4(3):220–241, 1994.

[58] Branko Grünbaum and Geoffrey Colin Shephard. *Tilings and patterns*. Freeman, 1987.

[59] A L Mackay. Generalised crystallography. *Izvj. Jugosl. centr. krist. (Zagreb)*, 10:15–36, 1975.

[60] Lester O Hedges and Stephen Whitelam. Limit of validity of Ostwald's rule of stages in a statistical mechanical model of crystallization. *The Journal of Chemical Physics*, 135(16):164902, 2011.

[61] Matthew J Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.

[62] Erik Winfree. On the computational power of DNA annealing and ligation. In Richard J. Lipton and Eric B. Baum, editors, *DNA Based Computers (DIMACS Series in Discrete Mathematics and Theoretical Computer Science)*, volume 27, pages 199–221. American Mathematical Society, 1996.

[63] Erik Winfree, Xiaoping Yang, and Nadrian C Seeman. Universal computation via self-assembly of DNA: Some theory and experiments. In Harvey Rubin and David Harlan Wood, editors, *DNA Based Computers II (DIMACS Series in Discrete Mathematics and Theoretical Computer Science)*, volume 44, pages 191–214. American Mathematical Society, 1999.

[64] Ming-Yang Kao and Robert T Schweller. Randomized self-assembly for approximate shapes. In *ICALP 2008: International Colloqium on Automata, Languages, and Programming*, volume 5125 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008.

[65] Ho-Lin Chen and Ashish Goel. Error free self-assembly using error prone tiles. In Claudio Ferretti, Giancarlo Mauri, and Claudio Zandron, editors, *DNA10: Proceedings of the 10th International Conference on DNA Computing*, volume 3384 of *LNCS*, pages 62–75. Springer, 2004.

[66] David Soloveichik, Matthew Cook, and Erik Winfree. Combining self-healing and proofreading in self-assembly. *Natural Computing*, 7(2):203–218, 2008.

[67] Ho-Lin Chen, Rebecca Schulman, Ashish Goel, and Erik Winfree. Reducing facet nucleation during algorithmic self-assembly. *Nano Letters*, 7(9):2913–2919, 2007.

[68] Ho-Lin Chen and Ming-Yang Kao. Optimizing tile concentrations to minimize errors and time for DNA tile self-assembly systems. In Yasubumi Sakakibara and Yongli Mi, editors, *DNA16: Proceedings of the 16th International Conference on DNA Computing and Molecular Programming*, volume 6518 of *LNCS*, pages 13–24. Springer, 2011.

[69] Hieu Bui, Craig Onodera, Carson Kidwell, YerPeng Tan, Elton Graugnard, Wan Kuang, Jeunghoon Lee, William B Knowlton, Bernard Yurke, and William L Hughes. Programmable periodicity of quantum dot arrays with DNA origami nanotubes. *Nano :etters*, 10(9):3367–3372, 2010.

[70] Shawn M Douglas, James J Chou, and William M Shih. DNA-nanotube-induced alignment of membrane proteins for NMR structure determination. *Proceedings of the National Academy of Sciences*, 104(16):6644–6648, 2007.

[71] Dongran Han, Suchetan Pal, Jeanette Nangreave, Zhengtao Deng, Yan Liu, and Hao Yan. DNA origami with complex curvatures in three-dimensional space. *Science*, 332(6027):342–346, 2011.

[72] Shawn M Douglas, Adam H Marblestone, Surat Teerapittayanon, Alejandro Vazquez, George M Church, and William M Shih. Rapid prototyping of 3D DNA-origami shapes with caDNAno. *Nucleic Acids Research*, 37(15):5001–5006, 2009.

[73] James C Wang. Helical repeat of DNA in solution. *Proceedings of the National Academy of Sciences*, 76(1):200–203, 1979.

[74] Sungwook Woo and Paul W K Rothemund. Programmable molecular recognition based on the geometry of DNA nanostructures. *Nature Chemistry*, 3(8):620–627, 2011.

[75] A M Mohammed, L Velazquez, A Chisenhall, D Schiffels, D K Fygenson, and R Schulman. Self-assembly of precisely defined DNA nanotube superstructures using DNA origami seeds. *Nanoscale*, 9(2):522–526, 2017.

[76] Tsu Ju Fu and Nadrian C Seeman. DNA double-crossover molecules. *Biochemistry*, 32(13):3211–3220, 1993.

[77] Do-Nyun Kim, Fabian Kilchherr, Hendrik Dietz, and Mark Bathe. Quantitative prediction of 3D solution shape and flexibility of nucleic acid nanostructures. *Nucleic Acids Research*, 40(7):2862–2868, 2011.

[78] Na Wu, Daniel M Czajkowsky, Jinjin Zhang, Jianxun Qu, Ming Ye, Dongdong Zeng, Xingfei Zhou, Jun Hu, Zhifeng Shao, Bin Li, et al. Molecular threading and tunable molecular recognition on DNA origami nanostructures. *Journal of the American Chemical Society*, 135(33):12172–12175, 2013.

[79] Leena Mallik, Soma Dhakal, Joseph Nichols, Jacob Mahoney, Anne M Dosey, Shuoxing Jiang, Roger K Sunahara, Georgios Skiniotis, and Nils G Walter. Electron microscopic visualization of protein assemblies on flattened DNA origami. *ACS Nano*, 9(7):7133–7141, 2015.

[80] Yonggang Ke, Luvena L Ong, William M Shih, and Peng Yin. Three-dimensional structures self-assembled from DNA bricks. *Science*, 338(6111):1177–1183, 2012.

[81] John SantaLucia and Donald Hicks. The thermodynamics of DNA structural motifs. *Annu. Rev. Biophys. Biomol. Struct.*, 33:415–440, 2004.

[82] William A Voter and Harold P Erickson. The kinetics of microtubule assembly. evidence for a two-stage nucleation mechanism. *Journal of Biological Chemistry*, 259(16):10430–10438, 1984.

[83] D Kuchnir Fygenson, H Flyvbjerg, K Sneppen, A Libchaber, and S Leibler. Spontaneous nucleation of microtubules. *Physical Review E*, 51(5):5058, 1995.

[84] Henrik Flyvbjerg, Elmar Jobs, and Stanislas Leibler. Kinetics of self-assembling microtubules: an "inverse problem" in biochemistry. *Proceedings of the National Academy of Sciences*, 93(12):5975–5979, 1996.

[85] Johanna Roostalu and Thomas Surrey. Microtubule nucleation: beyond the template. *Nature Reviews Molecular Cell Biology*, 18(11):702, 2017.

[86] Paul W K Rothemund, Axel Ekani-Nkodo, Nick Papadakis, Ashish Kumar, Deborah Kuchnir Fygenson, and Erik Winfree. Design and characterization of programmable DNA nanotubes. *Journal of the American Chemical Society*, 126(50):16344–16352, 2004.

[87] Pedro Fonseca, Flavio Romano, John S Schreck, Thomas E Ouldridge, Jonathan P K Doye, and Ard A Louis. Multi-scale coarse-graining for the study of assembly pathways in DNA-brick self-assembly. *The Journal of Chemical Physics*, 148(13):134910, 2018.

[88] David Yu Zhang and Erik Winfree. Control of DNA strand displacement kinetics using toehold exchange. *Journal of the American Chemical Society*, 131(47):17303–17314, 2009.

[89] Seung Hyeon Ko, Gregg M Gallatin, and J Alexander Liddle. Nanomanufacturing with DNA origami: Factors affecting the kinetics and yield of quantum dot binding. *Advanced Functional Materials*, 22(5):1015–1023, 2012.

[90] David Doty, Trent A Rogers, David Soloveichik, Chris Thachuk, and Damien Woods. Thermodynamic binding networks. In Robert Brijder and Lulu Qian, editors, *DNA23: Proceedings of the 23rd International Conference on DNA Computing and Molecular Programming*, volume 10467 of *LNCS*, pages 249–266. Springer, 2017.

[91] James C Mitchell, J Robin Harris, Jonathan Malo, Jonathan Bath, and Andrew J Turberfield. Self-assembly of chiral DNA nanotubes. *Journal of the American Chemical Society*, 126(50):16342–16343, 2004.

[92] Robert D Barish, Paul W K Rothemund, and Erik Winfree. Two computational primitives for algorithmic self-assembly: Copying and counting. *Nano Letters*, 5(12):2586–2592, 2005.

[93] Kenichi Fujibayashi, Rizal Hariadi, Sung Ha Park, Erik Winfree, and Satoshi Murata. Toward reliable algorithmic self-assembly of DNA tiles: A fixed-width cellular automaton pattern. *Nano Letters*, 8(7):1791–1797, 2008.

[94] Sekhar Babu Mitta, Sungguk Han, Srivithya Vellampatti, Anshula Tandon, Jihoon Shin, Tai Hwan Ha, and Sung Ha Park. Streptavidin-decorated algorithmic DNA lattices constructed by substrate-assisted growth method. *ACS Biomaterials Science & Engineering*, 2018.

[95] Thomas H LaBean, Hao Yan, Jens Kopatsch, Furong Liu, Erik Winfree, John H Reif, Nadrian C Seeman, et al. Construction, analysis, ligation, and self-assembly of DNA triple crossover complexes. *Journal of the American Chemical Society*, 122(9):1848–1860, 2000.

[96] Dage Liu, Sung Ha Park, John H Reif, and Thomas H LaBean. DNA nanotubes self-assembled from triple-crossover tiles as templates for conductive nanowires. *Proceedings of the National Academy of Sciences*, 101(3):717–722, 2004.

[97] Junghoon Kim, Tai Hwan Ha, and Sung Ha Park. Substrate-assisted 2D DNA lattices and algorithmic lattices from single-stranded tiles. *Nanoscale*, 7:12336–12342, 2015.

[98] Jianping Zheng, Jens J Birktoft, Yi Chen, Tong Wang, Ruojie Sha, Pamela E Constantinou, Stephan L Ginell, Chengde Mao, and Nadrian C Seeman. From molecular to macroscopic via the rational design of a self-assembled 3D DNA crystal. *Nature*, 461(7260):74–77, 2009.

[99] Tanguy Chouard. Structural biology: breaking the protein rules. *Nature News*, 471(7337):151–153, 2011.

[100] Hangxia Qiu, John C Dewan, and Nadrian C Seeman. A DNA decamer with a sticky end: the crystal structure of d-CGACGATCGT. *Journal of Molecular Biology*, 267(4):881–898, 1997.

[101] Xiaoping Yang, Lisa A Wenzler, Jing Qi, Xiaojun Li, and Nadrian C Seeman. Ligation of DNA triangles containing double crossover molecules. *Journal of the American Chemical Society*, 120(38):9779–9786, 1998.

[102] Phiset Sa-Ardyen, Alexander V Vologodskii, and Nadrian C Seeman. The flexibility of DNA double crossover molecules. *Biophysical Journal*, 84(6):3829–3837, 2003.

[103] Shuoxing Jiang, Fan Hong, Huiyu Hu, Hao Yan, and Yan Liu. Understanding the elementary steps in DNA tile-based self-assembly. *ACS Nano*, 11(9):9370–9381, 2017.

[104] John H Reif, Sudheer Sahu, and Peng Yin. Compact error-resilient computational DNA tilings. In *Nanotechnology: Science and Computation*, pages 79–103. Springer, 2006.

[105] Nadrian C Seeman. De novo design of sequences for nucleic acid structural engineering. *Journal of Biomolecular Structure and Dynamics*, 8(3):573–581, 1990.

[106] David Nečas and Petr Klapetek. Gwyddion: an open-source software for SPM data analysis. *Open Physics*, 10:181–188, 2012. Gwyddion URL: http://gwyddion.net/.

[107] Constantine G Evans, Rizal F Hariadi, and Erik Winfree. Direct atomic force microscopy observation of DNA tile crystal growth at the single-molecule level. *Journal of the American Chemical Society*, 134(25):10485–10492, 2012.

[108] John von Neumann. Various techniques used in connection with random digits. In A S Householder, G E Forsythe, and H H Germond, editors, *Monte Carlo Method*, pages 36–38. National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 1951.

[109] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[110] Tristan Stérin. Personal communication.

[111] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.