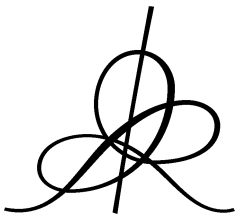
CIRCUITS AND SELF-ASSEMBLING DNA STRUCTURES

Alessandra CARBONE and Nadrian C. SEEMAN



Institut des Hautes Études Scientifiques
35, route de Chartres
91440 – Bures-sur-Yvette (France)

Janvier 2002

IHES/M/02/01

Circuits and Self-Assembling DNA Structures

Alessandra Carbone
Institut des Hautes Études Scientifiques
35, Route de Chartres
F-91440 Bures-sur-Yvette, France
carbone@ihes.fr

and

Nadrian C. Seeman
Department of Chemistry
New York University
New York, NY 10003, USA
ned.seeman@nyu.edu

Abstract: Self-assembly is beginning to be appreciated as a practical vehicle for computation. We investigate how some basic ideas on tiling can be applied to the assembly and evaluation of circuits. We suggest that these procedures can be realized on the molecular scale through the medium of self-assembled DNA tiles. One layer of self-assembled DNA tiles will be used as the program or circuit that leads to the computation of a particular Boolean expression. This layer serves to template the assembly of tiles whose associations then lead to the actual evaluation involving the input data. We describe the DNA motifs that can be used for this purpose, and we also describe how the template layer can be programmed for a particular purpose, much the way that a general-purpose computer can run programs for a variety of applications. The basic molecular system that we describe is fundamentally a pair of two-dimensional layers, but it appears possible to extend this system to a third dimension.

Introduction. The notion of computation by self-assembly goes back at least to Wang tiles (1). The combination of stable branched DNA molecules containing sticky ends to produce multidimensional constructs dates from the early 1980's (2). The notion of using tiles based on branched DNA as the tiling medium is due to Winfree (3), and Reif has commented extensively on this form of computation (4). The assembly of DNA-based tiles into two dimensional periodic arrays has been performed several times, using several different motifs (5-9). In addition, Rothemund has reported aperiodic self-assembly on the macroscopic scale (10). Recently, a one-dimensional example of logical computation using DNA tiles has been realized in the laboratory (11). Here, we point out that the cumulative XOR computation performed in (11) can also be viewed as a circuit that computes the parity of the elements in the input. We extend this concept to two and three dimensional circuit systems based on unusual DNA motifs. The unique approach that we take to molecular computation is that we propose a self-assembled programmable molecular plane that can process a variety of different inputs in a second layer, thereby extending the system into the third dimension. In addition, we point out that three-dimensional multi-layered systems can be programmed in the same fashion.

1. A Self-Assembling Template. A molecular circuit has been realized (11) that, given an entry of *n* bits, outputs *1* if the number of *1*'s in the entry is odd and *0* otherwise. (It computes the so-called *parity function*.) The idea is simple. One produces a *1*-dimensional template (Fig.1a) that represents the input sequence of values *0* and *1* for the circuit, together with an appropriate set of tiles (Fig.1b). The tiles code the truth table of the *XOR* operation (Fig.1c), denoted by the symbol ⊕_.

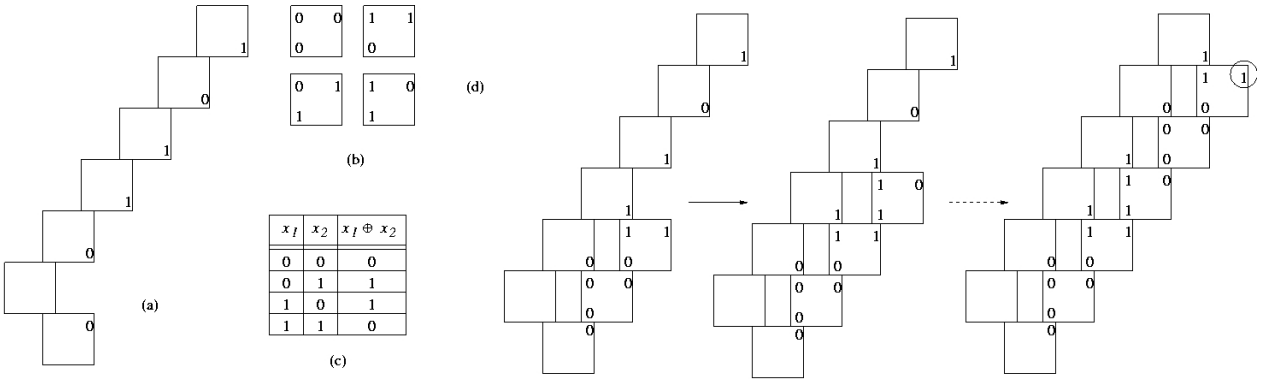


Fig.1: Template (a) and set of tiles (b) that realize the truth table of the ⊕ operation (c). The pairs of values x_1, x_2 on each row of the table correspond to the pair of values on the left hand side of the tiles, and the value $x_1 \oplus x_2$, associated with x_1,x_2 in the table, is recorded on the right hand side of the corresponding tile. Each row of the table corresponds to a tile. (d) Process of self-assembly of tiles on the template (a). The value *1* (circled) obtained at the end of the assembly, corresponds to the fact that the input sequence *001101* contains an odd number of *1*'s.

By adding the tiles to a solution containing the template, one induces the tiles to self-assemble on the template. One after the other, the tiles are "glued" to the template as soon as a double site emerges. The assumption here is that a tile can only attach to the template if there are two positions available for it to bind. The first arrow in Fig.1d illustrates the transition from the second to the third step of the self-assembly process. After five transition steps one obtains the complete structure (on the right), where the value of the last assembled tile is the value of the parity function on the sequence *001101*. It is worth pointing out that the labeled corners used here are logically equivalent to the labeled edges of Wang tiles. Also, the set of tiles is *complete*, i.e. for any possible output there is a tile with the same input.

In Sections 3 and 4 we extend the idea of assembling tiles to a template, and we illustrate a way to compute Boolean expressions; in Section 5 we suggest how to construct more general

programmable 3D devices. For the first application, we need to introduce some classical definitions and remarks of the theory of computation. (See also refs. (12, 13)).

2. Boolean Functions and Boolean Expressions. A *Boolean function* $f(x_1, \ldots, x_n)$ is a mapping from $\{0,1\}^n$ to $\{0,1\}^m$, where n is the number of distinct Boolean variables, and where we assume, for simplicity, $m=1$. The behavior of f is described through a *truth table* that associates a value *0* or *1* with each combination of values *0,1* of $x_1,\ldots, x_n$. For instance, the truth table in the top left of Fig.2, represents a function which answers *1* when exactly two of the three input variables take value *1*.

A *Boolean expression* is either a Boolean variable, or an expression of the form $b(\phi_1,\ldots, \phi_n)$ where b is an elementary Boolean function and $\phi_1,\ldots, \phi_n$ are Boolean expressions. Elementary Boolean functions, also called *logical connectives*, are, for example, $\oplus$ (*XOR*) mentioned in the previous section, $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $NAND$ and NOR. The expressions $\phi_1 \wedge \phi_2$ (conjunction of ϕ_1 and ϕ_2), $\phi_1 \vee \phi_2$ (disjunction), $\neg\phi_1$ (negation), ϕ_1 *NAND* ϕ_2, ϕ_1 *NOR* ϕ_2 are Boolean expressions. The $\oplus$ connective can be defined out of $\wedge$, $\vee$, $\neg$ as follows $x \oplus y \equiv (\neg x \wedge y) \vee (x \wedge \neg y)$; likewise, the *NAND* and *NOR* connectives are defined as x *NAND* $y \equiv \neg (x \wedge y)$ and x *NOR* $y \equiv \neg (x \vee y)$, respectively.

A logical connective has a fixed number of entries and a fixed number of exits. All the Boolean connectives mentioned above have two entries *x, y* and one exit, except for the negation $\neg$ which has only one entry. The truth tables of the elementary functions $\wedge$, $\vee$, $\neg$, *NAND*, *NOR* are given in Fig.2.

x_1	x_2	x_3	$f(x_1,x_2,x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

x_1	$\neg x_1$
0	1
1	0

x_1	x_2	$x_1 \wedge x_2$	$x_1 \vee x_2$	x_1 NAND x_2	x_1 NOR x_2
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Fig.2: truth tables for some Boolean functions on the variables x_1, x_2, x_3.

A*ny* Boolean function can be defined as a Boolean expression using the connectives $\wedge, \vee, \neg$. For instance the Boolean expression representing $f(x_1, x_2, x_3)$ in Fig. 2 (top left) is $(\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3)$. Given a truth table one can always write down the associated Boolean expression: one reads the rows of the truth table where the value of the function equals 1, and writes down a conjunction for each one of such rows. The conjunction describes whether the input variables are negated or not. For instance, at the first row of the truth table of $f(x_1, x_2, x_3)$ with value *1*, we see that x_1 takes value *0* and x_2, x_3 take value *1*. The conjunction $\neg x_1 \wedge x_2 \wedge x_3$ describes this fact. The Boolean expression is then defined as the disjunction of all conjunctions representing values *1* in the truth table of the function.

For practical purposes, one might want to reduce the number of connectives used to define the set of Boolean functions. The pair of connectives $\wedge$, $\neg$ is sufficient to this purpose since $\vee$ is definable from $\wedge$, $\neg$ as follows $x \vee y \equiv \neg (\neg x \wedge \neg y)$. Similarly, the pair of connectives $\vee$, $\neg$ can do it. *One* single connective is also sufficient: by definition, the *NAND* operator is essentially a $\vee$, i.e. $x \; NAND \; y \equiv \neg x \vee \neg y$, and in particular it allows one to define a negation as $x \; NAND \; 1 \equiv \neg (x \wedge 1) \equiv \neg x$. A similar argument shows that the connective *NOR* can be used to represent all Boolean functions.

2.1 Boolean Circuits. Another way to represent Boolean functions is through *Boolean circuits*. One starts with an oriented graph, which is free of non-trivial oriented cycles (simple loops included). Each node in the graph is marked with a label that is either a Boolean variable x_i, a designation of *1* ("true") or *0* ("false"), or an elementary connective. An example of a Boolean circuit is illustrated in Fig.3a.

If a node is marked with a Boolean variable, or with "true" or "false", then there are no incoming edges at that node. We call these nodes *input nodes*. If a node is labeled with a Boolean connective that has *k* inputs and *1* output, then the node should have exactly *k* edges going into it, and *n* edges, where $n \geq 0$, going out from it (to allow multiple uses of the output value). We call a node with no outgoing edges an *output node*.

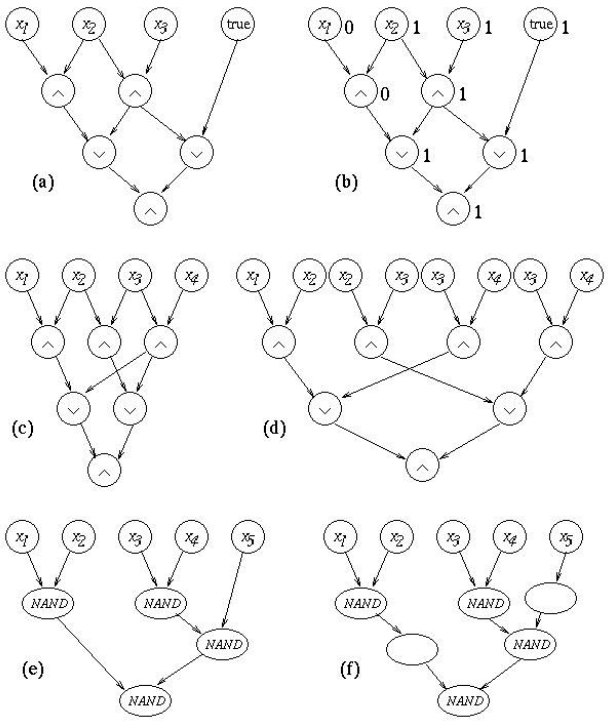


Fig.3: (a) A Boolean circuit. The exits of the gates are used up to two times (see the $\wedge$-gate in the middle of the circuit and the input gate labeled x_2). (b) Evaluation of the circuit (a) on the entries *0,1,1*; the input gate labeled "true" takes value *1*. (c)-(d) Boolean circuits computing the same function. The circuit (d) is tree-like: the output value of every gate is used exactly once; notice that by switching the position of its subcircuits, one can avoid the crossing of the edges. (e) The input x_5 of the circuit is combined with an intermediate result, generated in the second row of the tree, through a *NAND*-gate that lies in the third row. (f) Silent nodes are added to the circuit (e), to make it a tree whose branches pass through all rows.

A circuit represents a Boolean function from $\{0,1\}^n$ to $\{0,1\}$, where n is the number of Boolean variables labeling the input nodes. An assignment of values to the input nodes leads to an assignment to all other nodes of the circuit. This is done by steps: whenever all input nodes $z_1, z_2, \ldots, z_k$ of a given gate are evaluated, then the gate itself is evaluated and its output y is associated with the gate. This procedure is realized recursively on all gates from the inputs to the output of the circuit, and it is well founded because of the absence of oriented cycles in the underlying graph. Suppose we assign the values 0,1,1 to x_1, x_2, x_3, in the circuit of Fig.3a. The first step of the evaluation of the circuit computes $x_1 \wedge x_2$ and $x_2 \wedge x_3$, and it associates values 0 and 1 with the outputs of the respective gates. The second step evaluates $0 \vee 1$ and $1 \vee 1$ (where the gate ``true" takes value 1). The two gates in the third row take value 1 and 1 respectively. After the evaluation of the last conjunction (i.e. $1 \wedge 1$, in the fourth and last row), the circuit outputs 1. (See Fig.3b.)

An example of two *equivalent* Boolean circuits, i.e. computing the same Boolean function, is illustrated in Fig.3c-d. The circuit in Fig.3c *re-uses* several times the same computation (namely, the computation of $x_3 \wedge x_4$ is used twice as well as the inputs x_2 and x_3) while the circuit in Fig.3d uses each intermediate output exactly once. The second circuit forms a *tree*, i.e. for any input node there is exactly one oriented path from it to the output node of the tree. In the language of circuits, Boolean expressions correspond to trees. The reader can easily verify that $((x_1 \wedge x_2) \vee (x_3 \wedge x_4)) \wedge ((x_2 \wedge x_3) \vee (x_3 \wedge x_4))$ is the Boolean expression associated to the tree in Fig.3d. For each circuit there is an equivalent tree-like circuit.

3. Self-Assembling Boolean Expressions: the Parts and the Whole. Based on the same principle of self-assembly of tiles introduced in Section 1, we propose a general schema for the computation of Boolean expressions. The interest is two-fold. We investigate on the one hand *new* self-assembling structures and on the other hand the possibility to compute *arbitrary* Boolean functions.

We construct a 3D structure that corresponds to a computation on a circuit, by assembling two structures one lying on top of the other: the lower layer is a template and the upper one is formed by assembling a set of tiles to the template and to an array of "input" tiles. The *template* is a fixed support that codes a tree-like circuit. Each gate of the circuit takes a fixed position in the template. The *array* of "input" tiles provides the entries to the circuit. The assembly of the tiles depends on the information coded on both the tiles and the template. We first define the different parts playing a role in the construction (i.e. template, tiles, array) and then explain how they assemble together (in Section 3.3).

Before going in the details of the construction, we need to impose a structural assumption on the representation of tree-like circuits. It is due to a technical point that emerges in Section 3.1. We ask that edges in the tree do not cross (see legend of Fig.3d). For the nodes, we ask that the top row of the tree is constituted of input nodes only, that *all* inputs nodes lie in this row, and that a node x lying at row h is connected by edges to nodes occurring in row h-*1*. The nodes of the circuits in Fig.3c-d are connected to nodes lying in adjacent rows, but Fig.3e illustrates a tree where this hypothesis is not satisfied. To overcome this technical point, we allow an extra type of gate that simply passes on information (silently) without computing it. See Fig.3f.

3.1 The Template. A *template* is a discrete triangle on a regular lattice. As illustrated in Fig.4 (top), the nodes of the triangle (black dots), called *pawns*, are labeled in five different ways: *I, N,* T_R, T_L, **.*

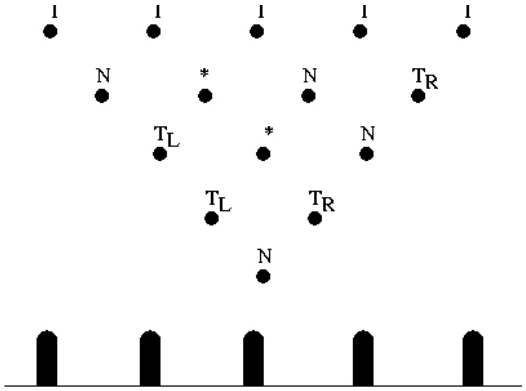


Fig.4: Top: a template of labeled pawns. The number of pawns in the first row (here =*5*) can be arbitrarily large. Bottom: a vertical section of the template. The pawns are "glued" to a surface.

The labeled nodes correspond to gates in a tree-like circuit. The label *I* corresponds to the *input* gates, and only nodes on the first (upper) row of the triangle are labeled this way. The label *N* corresponds to *computational* gates. They can be thought to be *NAND* gates for instance. Any node in the discrete triangle, with the exception of those on the first row, can be labeled this way. The labels T_R/T_L refer to *transmitter* gates. They are silent gates that pass on information without altering the value. The information might be passed on to the node on the left (T_L) or to the node on the right (T_R), when viewed from the input side of the template. The label * corresponds to nodes that are not linked by edges in the tree-like circuit.

Each Boolean expression (or tree-like circuit) is represented by exactly one template. This is straightforward to see. For instance, compare the circuit illustrated in Fig.3f with the template of Fig.4 (top).

Physically, a template is realized as a 3D structure built on a surface with a triangle of pawns sticking out of it (see Fig.4 and Section 3.4). It is worth pointing out that templates need not be triangular. In Sections 4 and 5 we discuss some other applications based on non-triangular templates.

3.2 The Tiles and the Array of Entries. As in (11), a tile is constructed in three parts where information is suitably coded. See Fig.5. The labels *i,j,k,l* are values *0,1* and *A* codifies extra information, that we call *middle label*. The pair of values *i,j* are called *input values* of the tile and *k,l* are called *output values* of the tile. A molecular representation of the tile as a DNA triple crossover (TX) molecule (8) is shown on the right. The values of *i, j, k* and *l* are encrypted in the overhanging sticky ends shown at each of the corners of the tile. The coding of *A* is explained in

Section 3.4; no coding is indicated in the molecular structure shown in Fig.5, where the middle helical domain terminates in hairpin loops.

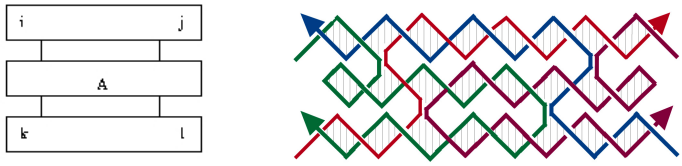


Fig.5: A tile (left) and its molecular representation as a TX molecule (right).

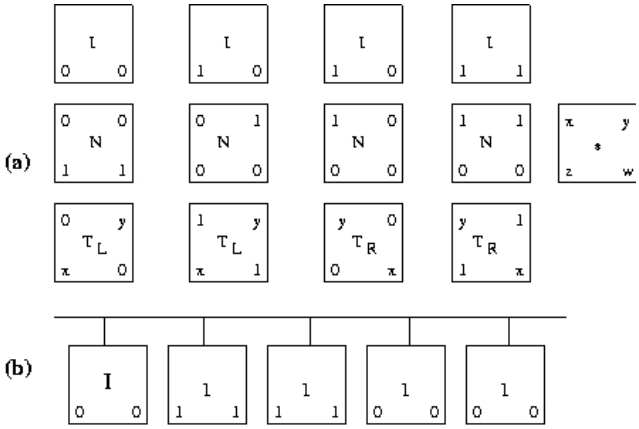


Fig.6: (a): four types of basic tiles, distinguished by their middle label: input (I), computational (N) and transmitter (T_L/T_R) tiles; (b): an array of entries constituted by a bar to which a sequence of input tiles is attached.

We define four types of tiles, each of them corresponding to a different labeling of the pawns of the template (see Fig.6a):

Input tiles, marked with the middle label I, represent the *0/1*-values given as inputs to the circuit. We could consider two tiles instead of four (namely the first and the fourth tile in the figure), but in principle, there is no reason for imposing this restriction.

Computational tiles, marked with the middle label N, represent the truth table for the *NAND* operator. The entries of the truth table are read on the upper side of the tile and the output value is read on the bottom side. The output value appears both on the left and on the right of the tile and this is because it might be combined with some value arriving either from the left or from the right of the circuit.

Transmitter tiles, marked by T_L/T_R, pass the value *0* or *1* on the right (T_R) or on the left (T_L). The letters *x, y* can take values in *0* or *1*, therefore the transmitter tiles are *16* in the whole.

Void tiles, marked by *, do not pass on any information. The letters *x, y, z, w* take the values *0* or *1*. There are *16* different types of void tiles.

A template defines a Boolean expression that accepts *n* input values. In physical terms, to pass on the *n* values to the template, we need to realize an array of entries made out of input tiles, as illustrated in Fig.6b.

3.3 How the Construction Works. Given a fixed template we superimpose to its first row of pawns an array of entries with the same number of entries. (Notice that we are exploiting here the 3D structure.) This is illustrated in Fig.7(left) with a view from the side and from the top: the pawns of the first row of the template match with the tiles of the array of entries. The input tiles of the array match the labels *I* of the pawns of the template and stick to the template through these attaching points.

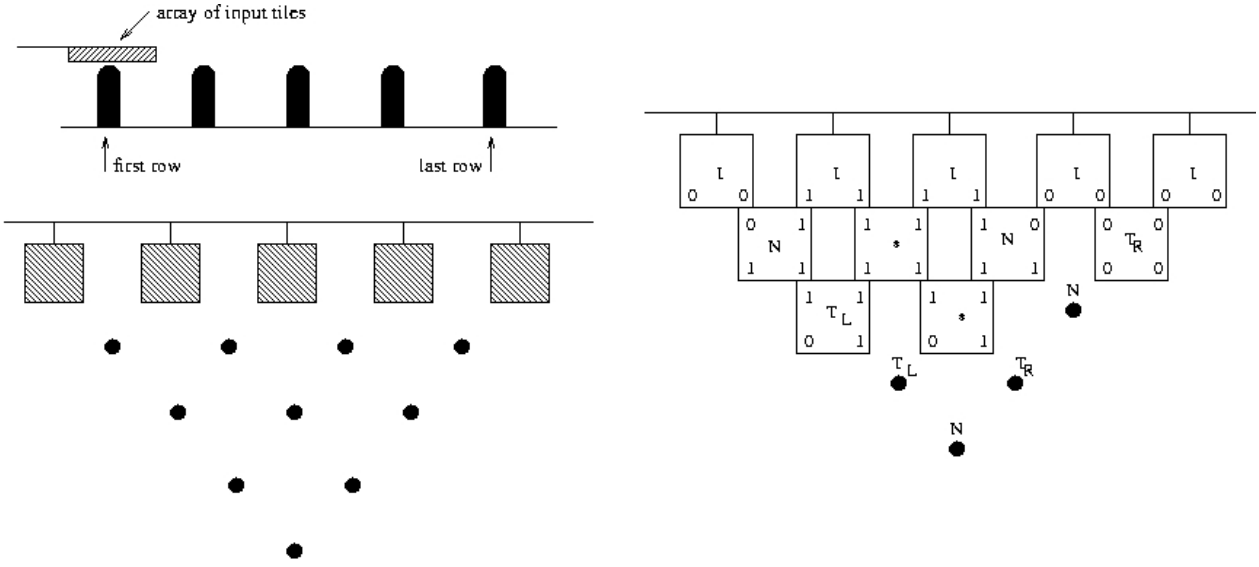


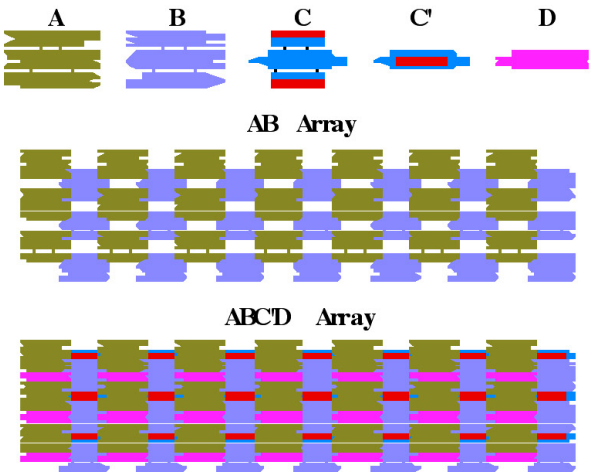
Fig.7: Left: a template and an array of entries viewed from the side and from the top; right: the assembly process, interrupted to an intermediate step, for the template of Fig.4.

In the solution we put template, array of entries, computational tiles, transmitter tiles and void tiles. No input tile is added to the solution, because the array of entries is considered to be pre-assembled. The tiles glue to the template on a second layer, by self-assembly. At the beginning, a number of tiles in the solution attach to the input tiles of the array by matching both the input

values (in the same way as described in Section 1) and the label of the adjacent pawns. This process repeats, row by row, until no more matching is possible. During the assembling process, a tile assembles by gluing its input values to the adjacent tiles *and* its middle label to the pawn. A display of an assembly process is illustrated in Fig.7(right).

Notice that the output values of void tiles (i.e. tiles labeled *) are not a priori determined by the structure. In fact given two input values and the label *, we have tiles in the solution with any pair of output values *k,l*. This freedom does not involve any complication because of the way that void tiles are combined with the rest of the tiles in the structure: void tiles never contribute input values to a computation. A similar consideration holds for transmitter tiles where the values of an opposite pair of corners in the tile are free. Thus, if the void tile in the third row had wound up in the position of the void tile in the second row, a T_L tile with *0*'s on both its non-transmissive corners would have been available to fit on the left of the third row.

3.4 The Chemical Nature of Template and Pawns. The 3D system described in (8) appears suitable for use as the pawn layer of this construction. This system consists of a 2D array that contains helices that protrude from it in one direction or another or both. An example is illustrated in Fig.8: the **AB** array consists of TX tiles connected 1-3, so as to leave gaps large enough for the insertion of a single DNA helix. The **D** component is just such a helix. However, the **C** component is another TX tile. It is rotated (and renamed **C'**) by three nucleotide pairs, ~102˚, to be nearly perpendicular to the array. Its central domain contains sticky ends to bind it to the gaps in the array not occupied by the **D** helix. Attachment in this fashion leads to a helix protruding from the **AB** array in each direction. TX molecules have also been attached to **AB**-like arrays from one of their end helices, rather than their middle helix (F. Liu & NCS, unpublished). A specific set of helices that are the out-of-plane domains of **C'**-like tiles could function as a hard-wired bottom layer of pawns. The sticky ends on the protruding helices could be tuned to select for the proper types of TX tiles (N, T_L, T_R, *), which would be associated with the central helical domain of a TX tile that lay parallel to the **AB** array. The outer domains would correspond to the Boolean input and output values of those tiles.



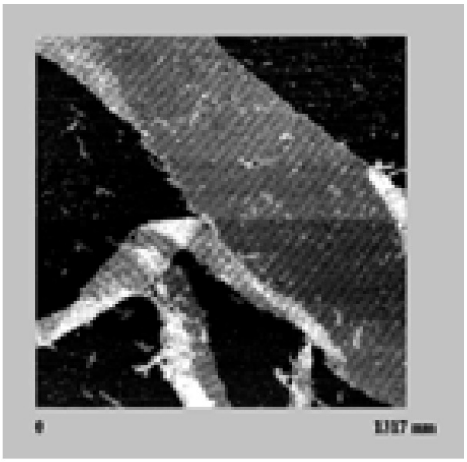


Fig.8. A Two-Dimensional Array Composed of Two TX Molecules, a Rotated TX Molecule, and a Double Helical Molecule. Left: A Schematic Drawing of the Array. The components of the array are shown at the top of the drawing. **A** and **B** are TX molecules containing sticky ends on all three of their domains. The geometrically-represented sticky ends are designed to be complementary so as to produce the **AB** array shown in the drawing. **C** is a third TX molecule, containing only a single pair of sticky ends, in its central domain. When its sticky ends pair with those of **A** and **B**, it is rotated about 103˚ relative to their plane. This rotated molecule is represented as **C'** at the top of the figure. **D** is a conventional double stranded molecule, designed to fill the gaps left in the array. The **ABC'D** array is shown below the **AB** array. Note that the presence of the C' units results in raised stripes that will be visible in the AFM. Right: An AFM Image of the **ABC'D** Array. The stripes of the array caused by the presence of the **C'** TX molecule are visible. The nearly rectangular nature of the array is visible.

4. Programmable 2D DNA-devices. Consider a template of k pawns, and imagine, for a moment, each pawn to be equipped with a specific coding sequence A_i, for $i=1,\ldots,k$. A *controlled* assembly of the array allows knowing the position of each coding A_i in the template, and opens the possibility to address the pawn i by means of its coding address A_i. Imagine also, that each pawn can enter into a bistable state, and that this state could be controlled as well. Then, the template could be programmed and re-used over and over again. It could be employed to induce a controlled assembly of tiles, which can lead to the computation of an arbitrary Boolean function, or to the creation of 2D DNA-layers satisfying specific properties. For instance, one can imagine large layers made out of alternating strips of tiles of width k; fancier designs of 2D DNA-layers and 3D arrays will be addressed in Section 5 and in the *Discussion*. In this section we shall analyze how to construct the units of a programmable 2D DNA-device.

4.1 Programmable Pawns. The pawns define the circuitry on which the tiles do the computation. We have shown above that it is possible to hard-wire a set of pawns by using TX molecules that are rotated out of the plane. However, maximum flexibility would be derived if the pawns themselves were programmable, so that any circuit could be derived from a standard "pegboard" arrangement of pawns. The recent development of the PX-JX_2 device (14) suggests a way to produce programmable pawns. The PX-JX_2 device is programmed by the addition of strands to the solution. Fig.9 shows the operation of the device: panel (a) illustrates the two different motifs, PX and JX_2. The PX motif contains four strands, wherein it appears that two double helices are wrapped around each other. This leads to an arrangement of DNA that contains two parallel double helical domains in which the strands cross over between domains at

every possible position. The JX_2 structure is just like the PX structure, except that two adjacent points of juxtaposition between the helical domains lack crossovers, and the helices are simply juxtaposed there. This topological difference leads to a global structural difference: the JX_2 structure is unwrapped by a half turn, relative to the PX structure. This feature can be seen from the labels associated with the ends of the helical domains: A and B are in the same place in each structure, but C and D are reversed between the PX and the JX_2 structures. Panel (b) shows a modified version of these structures that enables one to convert between them. Two parallel strands near the middle of the PX molecule, one red and one blue, have been interrupted, and the missing DNA is replaced with two green strands. The green strands contain single-stranded extensions on them, used to initiate a branch migratory process with the complete complements to the green strands; as shown by (15), branch migration eventually leads to removal of the strands (process I). The unstructured intermediate shown at the top can be converted to the JX_2 structure by the addition of the purple strands (process II). In like fashion, processes III and IV can convert the JX_2 structure back to the PX structure.

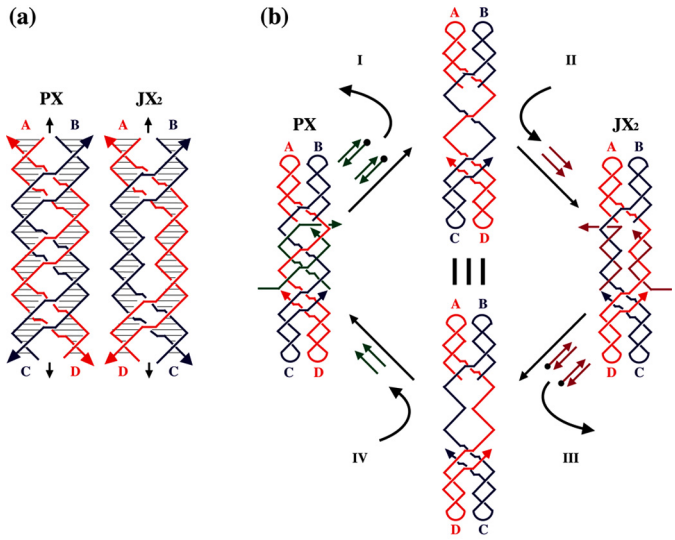


Fig.9. Schematic Drawings of the PX-JX_2 Device. (a) The PX and JX_2 motifs. The PX motif consists of two helical domains formed by four strands that flank a central dyad axis (indicated by the vertical black arrows). Two stands are drawn in red and two in blue, where the arrowheads indicate the 3' ends of the strands. The Watson-Crick base pairing in which every nucleotide participates is indicated by the thin horizontal lines within the two double helical domains. Every possible crossover occurs between the two helical domains. The same conventions apply to the JX_2 domain, which lacks two crossovers in the middle. The letters **A, B, C** and **D**, along with the color coding, show that the bottom of the JX_2 motif (**C** and **D**) are rotated 180˚ relative to the PX motif. (b) Principles of Device Operation. On the left is a PX

molecule. The green set strands are removed by the addition of biotinylated green fuel strands (biotin indicated by black circles) in process I. The unstructured intermediate is converted to the JX_2 motif by the addition of the purple set strands in process II. The JX_2 molecule is converted to the unstructured intermediate by the addition of biotinylated purple fuel strands in process III. The identity of this intermediate and the one above it is indicated by the identity sign between them. The cycle is completed by the addition of green set strands in process IV, restoring the PX device.

It appears possible to merge the PX-JX_2 device into a larger framework motif that can be incorporated into a DNA lattice much like the **C'** tiles above. This framework is shown in Fig.10.

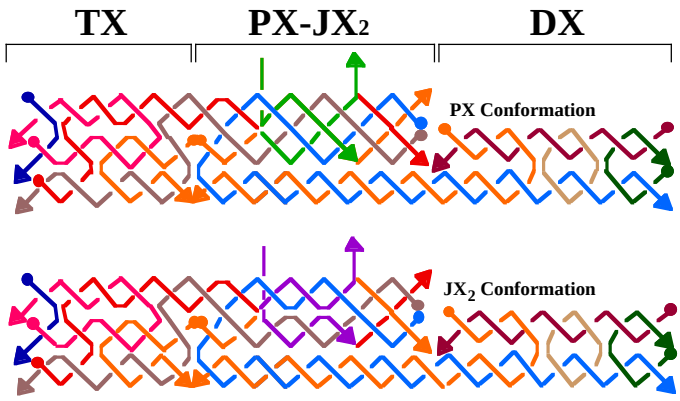


Fig.10: The motifs PX-JX_2 device is embedded into a larger framework motif. The three sections of the device are a TX portion on the left, the PX-JX_2 device in the center and a DX portion on the right. Arrowheads indicate 3' ends and filled circles indicate 5' ends. Note that the long orange strand contains a 5', 5' and a 3', 3' linkage.

The framework contains three portions, a TX motif on the left, a PX-JX_2 device in the center and a DX motif on the right; we term this structure a TPJD motif. The bottom helical domain is designed to be the portion that inserts into a simple 2D array containing gaps, such as the AB array shown in Fig.8. The TX portion is designed to support the PX-JX_2 device in the center. As described above, the PX-JX_2 device can be converted from the PX conformation to the JX_2 conformation by removing the light green strands that force it into the PX conformation and replacing them with the purple strands that put it into the JX_2 conformation. The DX section on the right of the framework acts as a protecting group. Only the helix that extends above it can interact with the tile-containing layer, and the one below is buried. Conversion from PX to JX_2 reverses the accessibility of the helices in the device. A pair of devices could interact with the

middle domain of a TX tile to select whether it should be a tile associated with the N, T_R, T_L or * functions. To select for more functions, other than the four above, a more complex tile than a TX would be needed. Recently, a 6-helix bundle (6HB) tile has been devised (16). See Fig.11. For a single layer application, as described so far, there are no obvious advantages to using such a tile over, say, a 4-helix analog of the planar TX tile. However, this is a system that can be extended to three dimensions, as we describe below.

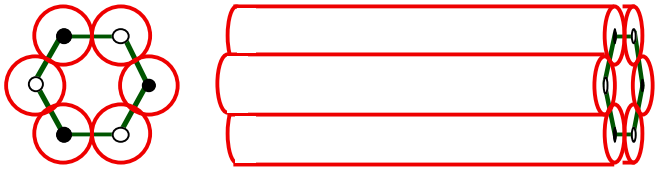


Fig.11: The six-helix bundle tile 6HB. A view down the helix axes is shown on the left and an oblique view is shown on the right. The helices are alternately antiparallel, meaning that they are phased about a half-turn apart. This feature is indicated by the alternating filled and unfilled circles in the drawings.

5. Programmable 3D-arrays. Templates can be used to construct 3D objects. The following basic construction suggests a way to do it. Consider a template whose shape need not be a triangle, for instance let it be a "square", and consider tiles equipped with a pawn, as illustrated in Fig.12a (left). The first layer of assembled tiles (possibly guided by an array of entries) forms a second "template" of pawns that can be used for assembling a second layer of tiles and so on (Fig.12a, right). More sophisticated schemes of constructions can be engineered. For instance, one might like to fill up only partially the board with tiles: by inserting appropriate coding in the pawns, one is able to build "walls" with specified "heights" (see Fig.12b).

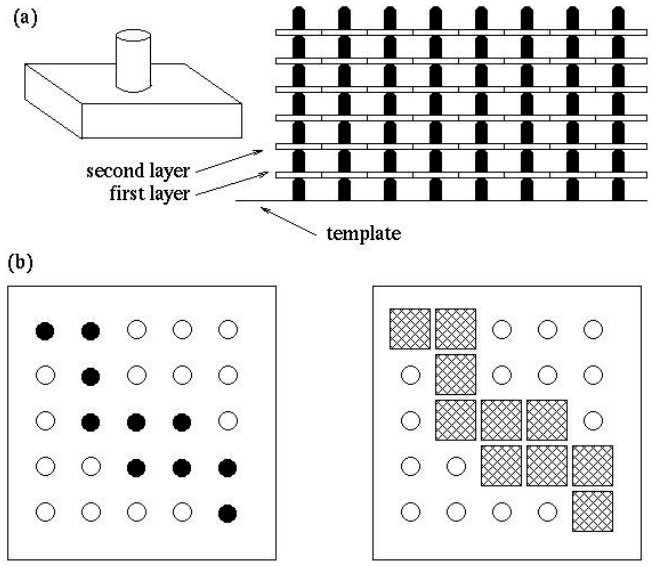


Fig.12: (a) A tile (left) and a multiple layers 3D assembly (right); (b, left): a board built out of two kinds of pawns, an active kind (black pawns) and an inactive one (white pawns); (b, right): view from the top of a wall built out of tiles lying above the active pawns.

Fig.13 illustrates the way that the TPJD and 6HB motifs could be combined to produce a 3D arrangement. Illustrated are layers of 6HB tiles connected by pairs of adjacent TPJD programmable pawns. The bottom layer of 6HB motifs forms the basis for the attachment of the TPJD programmable pawns, which in turn template the assembly of the next layer. With this paired arrangement of pawns, eight different types of 6HB tiles could be used.

Although there may ultimately be very difficult experimental problems with this type of 3D assembly, the components described above appear to be capable of serving to produce the 3D system described above. If the 6-helix bundle is used for the computational tiles and TPJD motifs are used for the pawns, one can build up a 3D structure using the scheme illustrated in

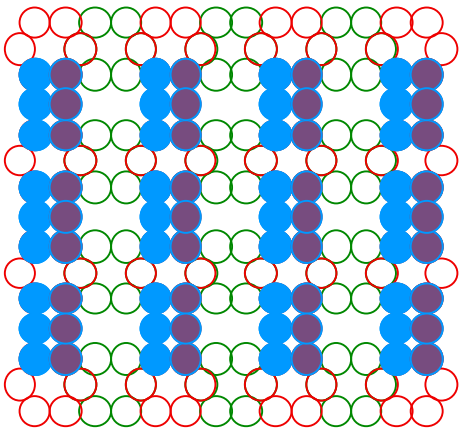
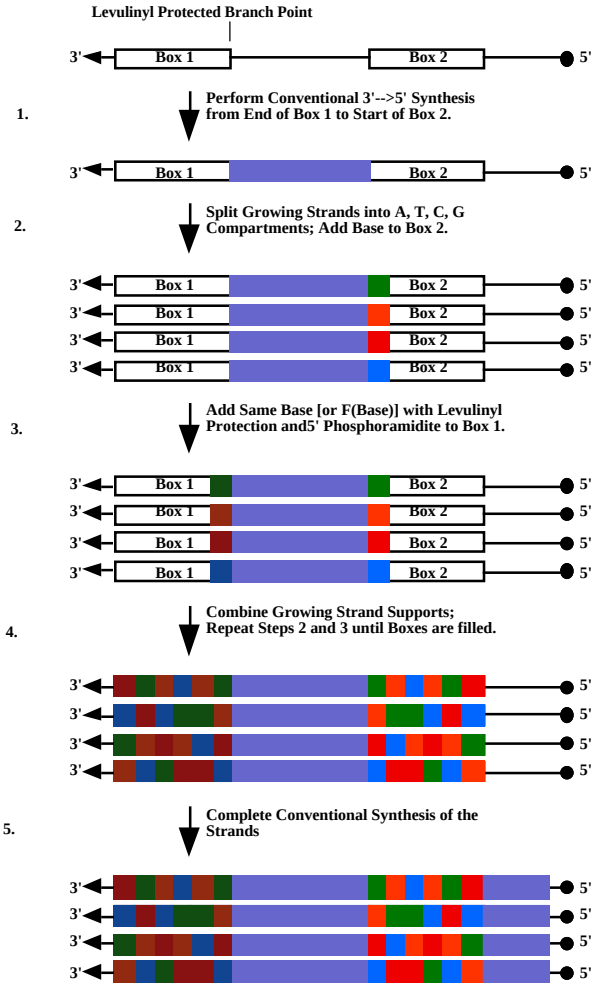


Fig.13.

Fig.13: Three layers of 6HB tiles. The red circles are helices in 6HB tiles that are in a plane closer to the reader than the green circles; the green circles are 6HB tiles that are further from the reader. The features shown as three blue filled circles in a row or three purple filled blue circles are programmable TPJD motifs that interact with the red tiles, and are closer yet to the reader. Two TPJD motifs abut each other to give a selectivity of eight possible tile types in any given layer. No TPJD motifs emanate up from the top layer, or down from the bottom layer, to give a sense of the intra-layer arrangement.

Discussion. We have described a novel system for computation on the molecular scale. We have described the arrangement of a programmable set of pawns that can produce a series of different logical operations in the level above them. Once assembled into a program, input data containing any values at all can be processed by this program. The pawns are predicated on sequence-dependent DNA nanomechanical devices that can be programmed individually. Each device pair can be responsible for generating four different specifications for the type of tile that is inserted in the computational array. The programs could be activated by the use of a device, such as that described in (17) to add specific strands upon an electronic signal, thereby programming the pawns with a conventional computer.

The key cost to operating the system will be generating the variety of strands necessary to label all of the different pawns needed for the system. This cost could be decreased significantly by the use of mix-and-split syntheses (18) that encoded sticky ends on the pawns that specified their location in the array, as well as the sequence of the controlling strands. In the TPJD device described above, it is not accidental that the complements to the set strands are also the same strands that contain the sticky ends. Using a synthesis that entails double protection features (base-sensitive levulinyl groups and acid sensitive dimethoxytrityl groups) and appropriately using 5' and 3' phosphoramidites, it should be possible to encode the same nucleotides (or a well-defined function of them (19)) to complement the set strands that are used in the sticky ends.



This split and mix synthesis is illustrated in Fig.14.

Fig.14. Split-and-Mix Synthesis applied to a strand containing the complement to the set strands of a PX-JX_2 device. The template of the strand is shown at the top of the diagram. It contains a branch point blocked in the leftward direction by a levulinyl group. In the first step, rightward

synthesis is shown to the right of the branch point. The strands are then divided into four groups, and in step 2, an A, T, G or C is added in the rightward direction to constitute the first base complementary to the set strand. Following that, in step 3, a nucleotide is added on the left by basic removal of the levulinyl group, and the same nucleotide or a nucleotide that is a well-defined function of the nucleotide added in step 2 is added with a 5'-phosphoramidite to produce the first base of the sticky end. The strands are mixed and again split, and the cycle is repeated until complete. Finally, the 5' end is added conventionally. The key feature in the strands indicated below step 4 is the mirror symmetry of the colors; after the Nth step, there will be 2^N different strands.

It is clear that two sticky ends could be specified, in addition to one sticky end and the region complementary to a set strand. For example, Box 1 and Box 2 could correspond to the same 5' sticky end, X, on either end of a strand, and of the same polarity, if the levulinyl-blocked branch point were a 3', 3' linkage; once initiated, both syntheses could be trityl blocked. Components generated in this fashion could be held together by complementary molecule, X', 3', 3', X', synthesized independently, but containing the same set of linker sequences as their complements. Using, for example, the TX molecule shown in Fig.5, mixing 2N strands with sticky ends could produce N^2 tiles. The arrangement in Fig.15 illustrates this concept. Thus, on the bottom layer, one would have tiles that only specified location. Were one to use a frame (8) to establish phasing, 6HB tiles formed with random mixtures of sticky ends would be located automatically by their sticky ends, because sticky end content would imply location. Using the split-and-mix synthesis technique, one could apply the same approach to producing the pawns, so they would be located at a particular position, but would also have set strands that were a function of position. The pawns in successive layers would be located as a function of the tiles to which they were attached.

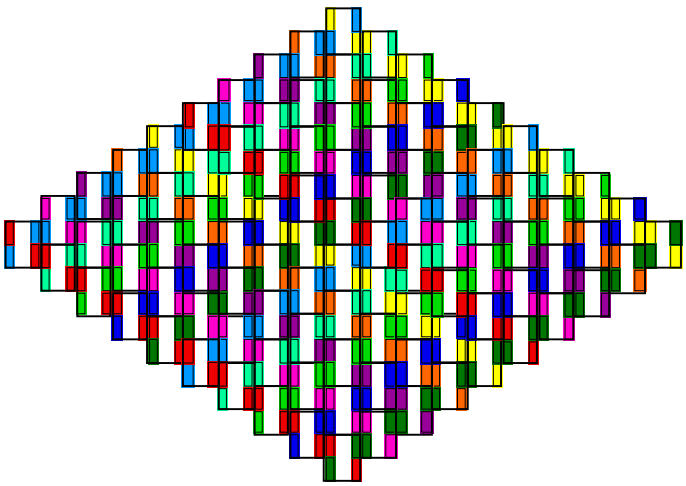


Fig.15: An Inexpensive Route to Diverse Patterns. This array contains 100 tiles, each of which represents a DNA TX molecule. The sticky ends of each tile are color coded, and the two opposite ends of each diagonal are complementary. There are five colors repeated twice in each direction, so the drawing contains four unit cells of the repeat. If each color represents a strand, then 40 strands could make this array from TX components. If diversity to produce TX molecules is taken into account, another 20 strands would be needed, for a total of 60. However, to program this array with unique sticky ends, 150 strands would be needed. For an array of N x M, with S strands needed per tile for diversity, SNM strands are needed for complete specification of sticky ends, but S(N+M) strands are needed for this scheme.

Acknowledgements. This work has been supported by grants GM-29554 from the National Institute of General Medical Sciences, N00014-98-1-0093 from the Office of Naval Research, NSF-CCR-97-25021 from DARPA/National Science Foundation and CTS-9986512, EIA-0086015, CTS-0103002 and DMR-01138790 from the National Science Foundation.

References:

1. Wang, H., (1963) *Proc. Symp. Math. Theory Automata* 23-56, Polytechnic, New York.
2. Seeman, N.C., (1982) *J. Theor. Biol.* **99**, 237-247.
3. Winfree E., (1996) In *DNA Based Computing*, ed. EJ Lipton, EB Baum. 199-219. Am. Math. Soc., Providence.
4. Reif, J.H., (1999) In *DNA Based Computers III*, ed. H Rubin, DH Wood. 217-254. Am. Math. Soc., Providence.
5. Winfree, E., Liu, F., Wenzler, L.A. & Seeman, N.C., (1998) *Nature* **394**, 539-544.
6. Liu, F., Sha, R. & Seeman, N.C., (1999) *J. Am. Chem. Soc.* **121**, 917-922.
7. Mao, C., Sun, W. & Seeman, N.C., (1999) *J. Am. Chem. Soc.* **121**, 5437-5443.
8. LaBean, T., Yan, H., Kopatsch, J., Liu, F., Winfree, E., Reif, J.H. & Seeman, N.C., (2000) *J. Am. Chem. Soc.* **122**, 1848-1860.
9. Sha, R., Liu, F., Millar, D.P. & Seeman, N.C., (2000) *Chem. & Biol.* **7**, 743-751.
10. Rothemund, P.W.K., (2000) *Proc. Nat. Acad. Sci. (USA)* **97**, 984-989.
11. Mao, C., LaBean, T., Reif, J.H. & Seeman, N.C., (2000) *Nature* **407**, 493-496 (2000); *Erratum*: *Nature* **408**, 750-750.
12. Papadimitriou, Ch., (1994) *Computational Complexity*, Addison-Wesley, New York.
13. Sipser, M., (1996) *Introduction to the Theory of Computation*, PWS Publishing Company.
14. Yan, H., Zhang, X., Shen, Z. & Seeman, N.C., (2001) *Nature*, in press.
15. Yurke, B., Turberfield, A.J., Mills, A.P., Jr., Simmel, F.C. & Neumann, J.L., (2000) *Nature* **406**, 605-608.

16. Mathieu, F., Mao, C. and Seeman, N.C., (2002), in preparation.
17. Gurtner, C., Edman, C.F., Formosa, R.E. & Heller, M.J., (2000) *J. Am. Chem. Soc.* **122**, 8589-8594.
18. Ohmeyer, M.H.J., Swanson, R. N., Dillard, L.W., Reader, J.C., Asouline, G., Kobayashi, R., Wigler, M. & Still, W.C. (1993), *Proc. Nat. Acad. Sci. (USA)* **90**, 10922-10926.
19. Seeman, N.C. (1990), *J. Biomol. Struct. & Dyns.* **8**, 573-581.