Title

Simulating 3-Symbol Turing Machines with SIMD||DNA

Author
Ong, Aaron Abram De Guzman

Publication Date
2021

Peer reviewed|Thesis/dissertation

Simulating 3-Symbol Turing Machines with SIMD||DNA

By

AARON ABRAM DE GUZMAN ONG
THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

David Doty

Fereydoun Hormozdiari

David Soloveichik

Committee in Charge

2021

Abstract

SIMD||DNA [11] is a model of DNA strand displacement allowing parallel in-memory computation on DNA storage. We show how to simulate an arbitrary 3-symbol space-bounded Turing machine with a SIMD||DNA program, giving a more direct and efficient route to general-purpose information manipulation on DNA storage than the Rule 110 simulation of Wang, Chalk, and Soloveichik [11]. We also develop software [9] that can simulate SIMD||DNA programs and produce SVG figures.

1 Introduction

DNA storage typically encodes information in the choice of DNA sequences [1,2,6], so that reading and writing require expensive sequencing and synthesis steps. An alternative "nicked storage" scheme of Tabatabaei et al. [10] uses a single long strand called a *register*, with a fixed sequence. Information is stored in the choice of short complementary strands to bind to the register. This gives the potential to process the stored information using DNA strand displacement [8], which reconfigures which DNA strands are bound without changing their sequences. Thus manipulation of the stored information (i.e., computation) can potentially be done *in vitro* with simpler lab steps than DNA sequencing or synthesis.

The SIMD||DNA model of Wang, Chalk, and Soloveichik [11] is an abstract model of such a system. It allows parallel in-memory computation on several copies of the register; each register may store different data. In the experimental implementation, each register strand is attached to a magnetic bead, enabling elution: washing away strands not bound to a register while keeping the registers (and their bound strands) in the solution. This motivates the "multi-stage" SIMD||DNA model of DNA strand displacement, which at a high level works as follows. Each stage is called an *instruction*, consisting of a set of strands to add to the solution. It is assumed that strand displacement reactions proceed until the solution reaches equilibrium, at which point all strands and complexes not attached to a register are washed away. The strands for the next instruction are then added. A key aspect of the model is that the wash step can constrain what strand displacement reactions are possible afterward, compared to "one-pot" strand displacement schemes that mix all strands from the start. This gives the SIMD||DNA model potentially more power than one-pot DNA strand displacement. See section 2 for a formal definition and [11] for more details and motivation for the model.

A major theoretical result of [11] is a SIMD||DNA program that that simulates a space-bounded version of cellular automata Rule 110. When space is unbounded, Rule 110 is known to be efficiently Turing universal, i.e., able to simulate any single-tape Turing machine [3] with only a polynomial-time slowdown [4], though by an awkward indirect construction and encoding with very large constant factors. We show how to simulate an arbitrary 3-symbol space-bounded single-tape Turing machine *directly* with a SIMD||DNA program. Since custom manipulation of bits is much easier to program in a Turing machine than Rule 110, this gives a more direct, efficient, and conceptually simple method of general-purpose information processing on nicked DNA storage. Although we have not worked out the details, it seems likely that the construction can be extended straightforwardly to Turing machines with alphabet sizes larger than 3.

Our construction was designed and tested using software we developed [9] for simulating the SIMD||DNA model. It is able to take a description of an arbitrary SIMD||DNA program: a list of instructions, where each instruction is a set of DNA strands to add. It produces figures indicating visually how the steps work, both with text printed on the command line (for quickly testing ideas) and SVG figures, such as most of those in this paper.

2 Model

In this section we define the model of SIMD||DNA [11].

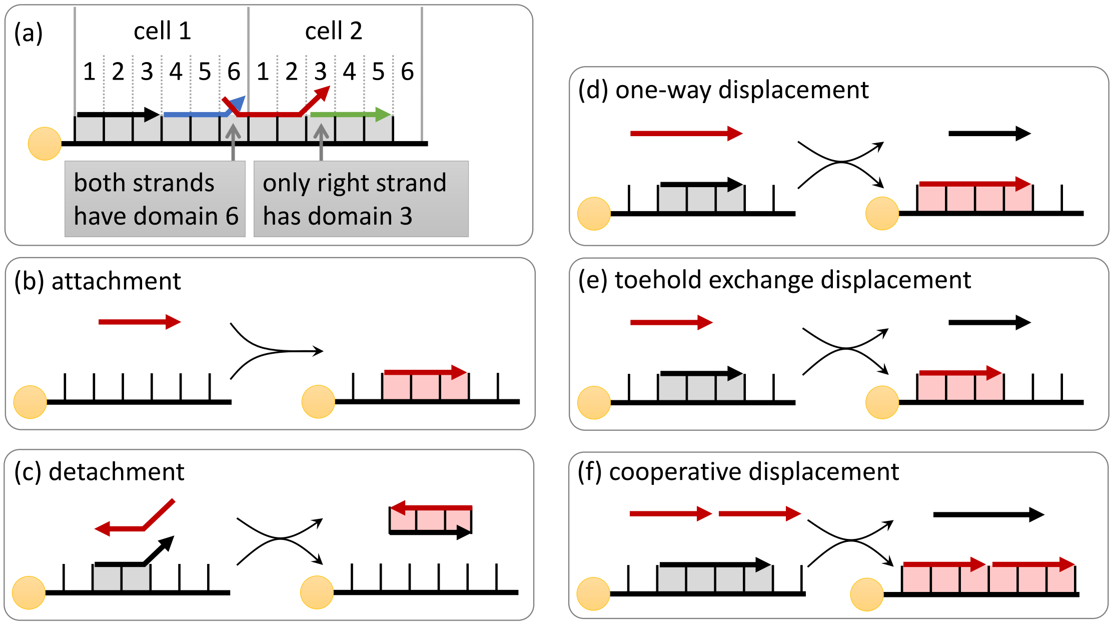
(a)
cell 1
cell 2
1 2 3 4 5 6 1 2 3 4 5 6
both strands have domain 6
only right strand has domain 3
(d) one-way displacement
(b) attachment
(e) toehold exchange displacement
(c) detachment
(f) cooperative displacement

Figure 1: Notational conventions and reactions in the SIMD||DNA model. The register strand is on the bottom in each subfigure, with a yellow round "magnetic bead" depicted on the left (not depicted in subsequent figures). Lightly shaded gray or pink regions denote bonds (double-stranded regions), but later figures omit this and simply draw a forward strand (one with $5'$ end on left and $3'$ end on right) immediately above the domains to which it is bound on the register strand. **(a)** Conventions for domain names of strands. Domains are numbered $1, \ldots, d$ within each cell; $d = 6$ in fig. 1(a) and $d = 18$ in subsequent figures. The register strand has the starred versions of these domains. If a top strand is horizontal over domain i, it has domain i. If it is diagonal over the whole domain, it has an unlabelled domain distinct from all register domains (used as a toehold overhang for detachment, see subfigure (c)). If two strands both partially cover a domain then they both have that domain. **(b)** A forward instruction strand can attach if at least two complementary consecutive domains are unbound on the register. **(c)** Reverse instruction strands can bind to toehold overhangs on forward bound strands to detach them from the register. The fact that the (unlabelled) toeholds are complementary is indicated by a diagonal bend in the reverse strand matching. **(d)** Forward instruction strands can do toehold-mediated strand displacement, one-way if the displacing strand contains all the domains of the displaced strand. **(e)** If the displacing strand is missing the last domain of the displaced, displacement can also happen, known as *toehold exchange*. This is often called "reversible" since it conserves the number of bound domains, but in the SIMD||DNA model, instruction strands are added in large excess over registers, making it effectively irreversible due to the entropic bias toward binding the instruction strand. Thus it is depicted with irreversible arrows in the figure. **(f)** Two forward strands can cooperate to displace a single bound top strand if neither has enough domains to displace on its own.

See fig. 1 for notational conventions in the SIMD||DNA model and an explanation of the basic strand displacement reactions. The register strand is on the bottom in each sub-figure, with a yellow round "magnetic bead" depicted on the left (bead not depicted in subsequent figures). The register strand has its $3'$ on the left and $5'$ end on the right. The model allows multiple registers to be present in solution at once, each possibly configured differently. However, it is assumed that register strands are sufficiently dilute that they do not interact with each other or with strands that have been displaced from other registers. Thus all figures depict only a single register.

The register strand is divided into *cells*, which are further divided into *domains*. Each domain can be thought of as a fixed-length DNA sequence with relatively weak binding (e.g., 5-7 bases). A strand is stably attached to the register strand only if it is bound by at least two domains, but one domain is sufficiently long to act as a "toehold" to help initiate strand displacement (fig. 1(c-f)). Within a cell with d domains, each domain is unique and assumed to be named $1, 2, \ldots, d$. The register strand has the starred version of these domains, e.g., $1^*, 2^*, 3^*, 4^*, 5^*, 6^*, 1^*, 2^*, 3^*, 4^*, 5^*, 6^*$ reading from the register's $3'$ to $5'$ end (left to right) in fig. 1(a). All cells have the same ordered list of domains, so for example in fig. 1(a), domain 5 in cell 1 is the same DNA sequence as domain 5 in cell 2.

An *instruction* is a set of strands that are added to the solution at once. fig. 1(b-f) shows the various reactions that these strands might conduct to change the configuration of strands attached to the register. Multiple reactions can occur in a cascade in a single instruction.[1] Instruction strands can either be *forward* ($3'$ arrow on right) or *reverse* ($3'$ arrow on left). Forward instruction strands can do attachment and displacement reactions (fig. 1(b,d-f)) and reverse instruction strands can detach forward strands previously bound to the register (fig. 1(c)).

Crucially, instruction strands are added in large excess over the register strands. Thus even the toehold exchange displacement, which is often considered reversible due to being enthalpically balanced (same number of domains bound before and after), is actually irreversible in the SIMD||DNA model, due to the entropic bias toward binding the new instruction strand with much larger concentration than the strand it displaces.

The notation of [11] uses dashed lines for reverse strands used for detachment, as a visual reminder that they do not bind to the register. We leave reverse strands as solid lines and rely on the $3'$ arrow to denote that the strand is reversed. We reserve the dashed line notation for later figures to depict *inert* instruction strands: instruction strands that are shown above the register where they would bind if possible, but where no reaction allows them to do so in the current configuration. We also have a slightly different notation for strands with domains mismatching the register: in [11], these are depicted by writing an explicit domain name. In our convention, the drawing of that part of the strand as diagonal and lying entirely above the register domain indicates that the top strand domain and register domain are not complementary (fig. 1(a), cell 2, domain 3). To denote that two adjacent top strands share the same domain, both of which can bind to the register (so they dynamically compete with strand displacement), we draw both strands partially

[1] See for example instruction 39 in fig. 8. In the right cell, an orange instruction strand displaces an orange strand bound to the register via toehold exchange. This opens a toehold for a blue instruction strand to displace the bound blue strand, resulting in the configuration shown at the beginning of instruction 41.

horizontal over the domain, and partially diagonal (fig. 1(a), cell 1, domain 6).

Although these rules allow for nondeterministically competing reactions, our construction is deterministic in the sense that there is only one sequence of reactions possible in any instruction step.

After instruction strands are added and the described reactions go to completion, the *wash* step removes all strands not bound to the register. This includes excess instruction strands that never reacted, as well as strands that were displaced or complexes formed in a detachment reaction.

3 Simulation of Turing machine in SIMD||DNA

In this section we describe how to simulate an arbitrary 3-symbol single-tape Turing machine with SIMD||DNA instructions.

3.1 High-level overview of construction

Since the SIMD||DNA model as defined has no mechanism to grow the register strand, it can only simulate a fixed-space-bound Turing machine (a.k.a., linear-bounded automaton), which starts with s total tape cells and never moves the tape head off of them. A 3-symbol, space-s Turing machine has three tape symbols: $0, 1, \sqcup$. The binary input $x \in \{0,1\}^{<s}$ is represented by string $x\sqcup^{s-|x|}$ on the tape in the initial configuration, i.e. x padded with enough blank symbols to make s total tape cells. We use as a running example the 5-transition Turing machine in fig. 2, which increments a binary number.

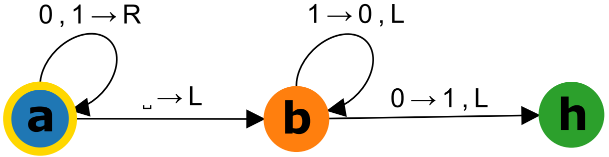

Figure 2: Turing machine (start state a) that increments an integer represented in binary, with the least significant bit on the right. This example is simulated in all subsequent figures.

Each cell of the register represents a tape cell of the Turing machine. If the Turing machine has t total transitions, then each cell uses $d = 2t + 8$ domains.

For each Turing machine, there is a fixed sequence of instructions that, after executing, will update the register to represent the next configuration of the Turing machine.

The cell with the tape head is the only cell with uncovered register domains. Which domains are uncovered (known as a *transition region*) represents both the current state of the Turing machine and the symbol written on that tape cell. For all other cells, a disjoint region (the *symbol region*) represents the symbol on that cell through its pattern of nicks. On the cell with the tape head, the symbol region has no nicks (and represents no symbol) since it is covered by a longer 8-domain strand.

3.2 Representation of Turing machine tape cell as a register cell

In the SIMD||DNA representation of a Turing machine, each register cell represents a single Turing machine tape cell. We represent each Turing machine with tape alphabet $\Gamma = \{0, 1, \sqcup\}$, state set Q, and halt state h, as a set of *transitions*, where each transition $(q, b) \to (r, c, m)$ means that if the Turing machine is in state $q \in Q \setminus \{h\}$ reading symbol $b \in \Gamma$, it changes to state r, writes symbol c, and moves one cell by $m \in \{L, R\}$ (left or right). Since the Turing machine is deterministic, for each state-symbol pair, there is at most one transition with that pair on the left. (But some such pairs could be undefined, e.g., there is no $(b, \sqcup) \to \ldots$ transition in fig. 2.)

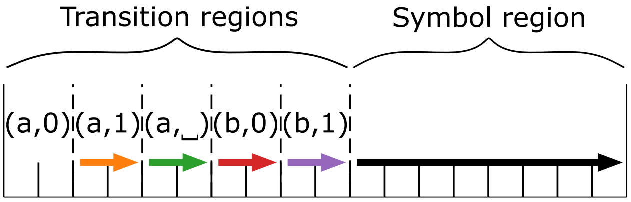

Figure 3: A SIMD||DNA cell where the tape head is presently located. The $(a, 0)$ region is fully exposed, indicating that the Turing machine is in state a and that the cell contains the symbol 0. The other transition regions are fully covered, and the symbol region (rightmost 8 domains of the cell) is covered by a single long strand, not encoding any symbol (which is encoded by the uncovered transition region).

Representation of tape cell with tape head We take every state-symbol pair $(q, \sigma) \in (Q \setminus \{h\}) \times \Gamma$ (each possible left side of a transition) and represent each as two consecutive domains in a SIMD||DNA register cell. See fig. 3. Recall the binary incrementing Turing machine of fig. 2. It has five transitions: $(a, 0) \to (a, 0, R)$, $(a, 1) \to (a, 1, R)$, $(a, \sqcup) \to (b, \sqcup, L)$, $(b, 0) \to (1, h, L)$, $(b, 1) \to (0, b, L)$. We call the pair on the left the *transition input*. Each of the given transition inputs is represented in the SIMD||DNA cell using two domains, requiring ten domains total for our example. Since each register cell represents a cell in M, we must denote the presence of the tape head on one of the cells. If the tape head is present on a given cell and if the current Turing machine configuration has a valid transition, then the two domains that represent that transition will have no top strand attached to them, leaving them exposed. For example, if the tape head is on a cell with the 0 symbol, and the Turing machine is currently in state a, then the region that represents $(a, 0)$ in that cell will be exposed to serve as a toehold for strand displacement. The other transition regions are fully covered by 2-domain strands.

Representation of tape cell without tape head If the tape head is not present on a cell, or if no valid transitions exist for the current configuration,[2] then every transition region is covered by 2-domain strands. Eight additional domains at the rightmost part of the cell, called the **symbol region** represent the current symbol written on that cell.

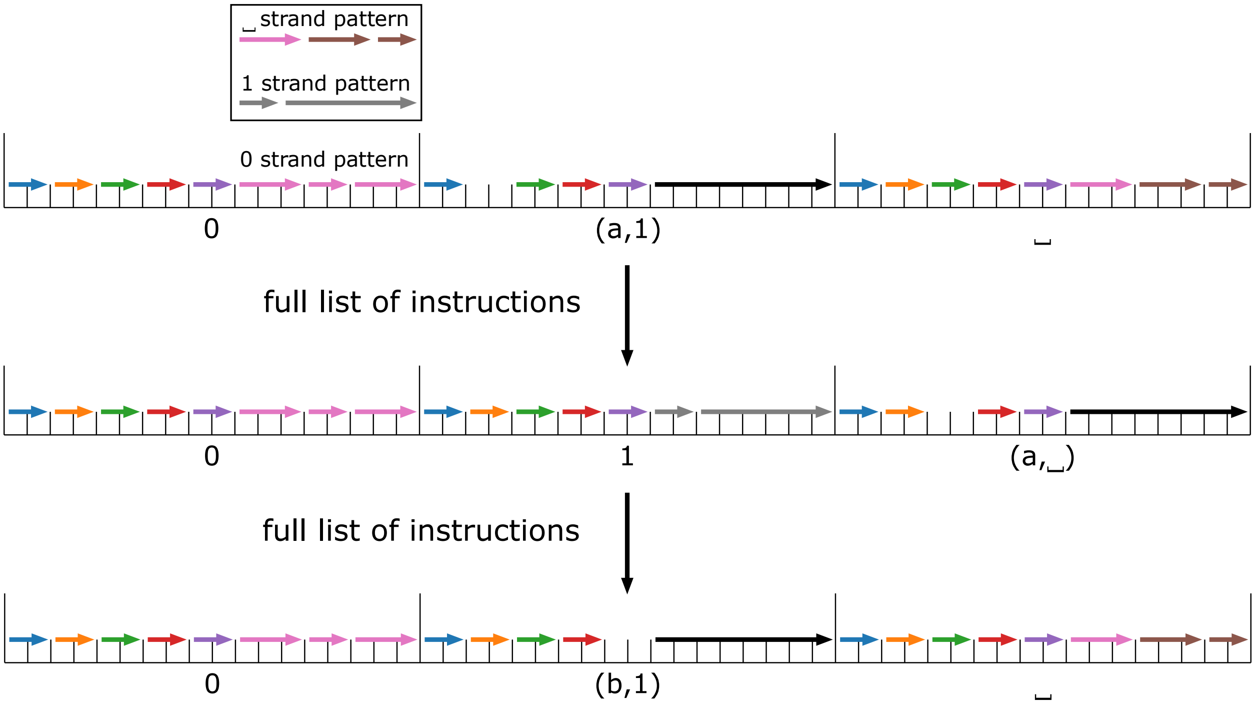

Figure 4: High-level overview of construction. A Turing machine register currently in state a, with the tape head on the second cell. The second cell contains the symbol 1. The leftmost cell contains symbol 0. The inset above shows encodings for 1 and ␣. All the transition regions are fully covered in cells lacking the tape head. After the full list of instructions in the SIMD||DNA program are complete, the register represents the Turing machine configuration with state a and the tape head moved to the rightmost cell with the ␣. The same full list of instructions updates the register again, now representing the Turing machine configuration in state b with the tape head back on the middle cell.

Whenever the tape head is present on a cell, the symbol region is covered by a single 8-domain strand that does not encode any symbol, since the symbol information is already encoded in the transition region with an open toehold.

3.3 Detailed description of SIMD||DNA instructions simulating a Turing machine

We designed an algorithm that converts Turing machine specifications from https://turingmachine.io into SIMD||DNA representations, along with the equivalent instructions. Each transition τ_i has an associated

[2] For example, if the machine has halted; see the bottom register configuration of fig. 9 for a case where the state is non-halting but no valid transition exists.

sublist of instructions L_i, and, not knowing which transition is applicable to the current configuration, we simply add instruction strands in order from $L_1, L_2, \ldots$. For $i \neq j$, to ensure that L_j instructions have no effect when the current applicable transition is τ_i, we "plug" the open domains of other transition regions with a strand and remove the plug strand once it's time to process that transition. Because the SIMD||DNA model allows parallel computation among multiple registers in the same solution, this prevents instructions meant for one configuration from affecting registers currently not in that configuration. In the beginning, all transition regions are plugged, where the order of processing for the transitions is arbitrary.

Pre-plug and post-plug strands to protect instructions for inapplicable transitions from affecting configuration The full list of instructions to simulate a Turing machine transition works as follows. Recall that in the "clean" configurations shown in fig. 4, the only exposed register domains are on the cell representing the tape head. The first instruction in the entire list contains *pre-plug* strands for each transition region. At the end of each instruction sublist L_i, a *post-plug strand* is also placed on the transition region that represents the Turing machine's next applicable transition. Examples of both can be seen in fig. 7. These strands act like a "chemical protecting group" that prevents instruction sublists L_i from modifying the register unless they apply to the intended transition. The difference between the pre-plug and post-plug strands is that pre-plug strands protect the configuration when using instructions strands *before* the applicable transition, whereas post-plug strands protect the configuration when using instructions strands *after* the applicable transition. In the left part of fig. 7, because the register has a pre-plug strand in $(a, \sqcup)$, it means that the instruction sublist $L_{(a,\sqcup)}$ has not been applied to it yet. Instruction sublists for the other transitions $(a, 0) \to \ldots, \quad (a, 1) \to \ldots, \quad (b, 0) \to \ldots, \quad (b, 1) \to \ldots$ will be inert, not affecting the register. The first instruction of $L_{(a,\sqcup)}$ will remove this pre-plug strand so that any register in the $(a, \sqcup)$ configuration can be processed. The instruction sublists will result in a configuration like that of the bottom of figs. 8 and 9. fig. 9 shows instructions that affect the cell where the tape head *was* (right cell), not where it will be *next* (left cell), which is why the left cell is the same in both figs. 8 and 9. This almost represents the next Turing machine configuration, but with the appropriate transition region covered by a *post-plug strand*. The final instruction in the entire list (fig. 7) removes this post-plug strand, restoring the register configuration to be as shown in fig. 4.

The post-plug strand placed on the transition region at the end of simulating a transition has a different purpose from the pre-plug strand placed in instruction 1. Its purpose is to prevent the register from updating its state multiple times in the same instruction iteration. For example, if a register has a post-plug strand on the $(b, 1)$ region (indicating that it has been processed and that its next transition is $(b, 1)$), and the instruction sublist that processes $(b, 1)$ comes after, the register will be unaffected by $(b, 1)$'s deprotecting instruction, keeping it inert throughout. The final instruction in the entire iteration removes these post-plug strands from the registers, as seen in fig. 7, preparing the registers for the next iteration of the instruction set.

Note the duality between pre-plug and post-plug instructions. *All* pre-plug strands are included in the first instruction, though only one of them will bind (the one matching the applicable transition), and instruction strands removing *all* post-plug strands are included as part of the last instruction, though only one will find its complementary post-plug strand to remove. On the other hand, each pre-plug instruction is removed more specifically, by adding a single complementary strand to remove it just prior to the sublist of instructions corresponding to the applicable transition. Similarly, each post-plug strand is added by itself, at the end of the instruction sublist corresponding to the applicable transition.

In the next section, we will describe the details of the instruction sublists that represent the Turing machine transitions.

Sublist of instructions representing a single Turing machine transition *Figure conventions.* For the figures explaining SIMD||DNA instructions that simulate a single transition of the Turing machine (fig. 8 and beyond), we use the following conventions in figures. Several register configurations are shown, but they are not necessarily consecutive. Each is numbered with its absolute index in the list of all 86 instructions implementing the Turing machine of fig. 2. An example with all instructions shown can be found in appendix A. If two adjacent configurations have non-consecutive instruction indices, this means that the instructions not shown are inert: their strands do not affect the register in that configuration. The instruction strands that have just been added are always shown above the register, with a solid line if they will do a reaction as in fig. 1, and with a dashed line if that instruction strand is inert for that configuration. The final configuration in each figure does not show any instruction strands, but for all figures there is a followup figure showing what happens next from that configuration (possibly the followup is fig. 7, the final instruction in the entire program, removing the post-plug strand from the next applicable transition region).

Each Turing machine transition is individually processed by a sublist of instructions. The pre-plug strand is first removed by an instruction containing its complementary strand, so that its corresponding transition region in the cell can be used as a toehold, such as instruction 38 in fig. 8. The next instructions then update the contents of the current cell to encode the symbol that the tape head writes. For example, in fig. 9, the transition $(a, \sqcup) \rightarrow (b, \sqcup, L)$ is represented, and the strand encoding of $\sqcup$ is placed in the right cell after the tape head writes on it and moves left. After that, the instructions check the contents of the tape head's new location and determine the Turing machine's next configuration. In fig. 8, the tape head moves to the left cell and finds a 1, and the Turing machine goes to state b, so it leaves a post-plug strand on $(b, 1)$'s transition region to show that the register has been processed for that instruction iteration, as seen in instruction 46.

Left versus right tape head moves In the SIMD||DNA instructions implementing a single transition, there are two sublists: next-cell instructions and previous-cell instructions. As their names indicate, next-cell instructions update the contents of the tape head's destination, while previous-cell instructions update the contents of the tape head's former location. Other factors such as the symbol to be written on the current cell and the next applicable transition region only introduce minor variations in the instruction strands.

For transitions moving the tape head *left*, the next-cell instructions precede the previous-cell instructions (see figs. 8 to 11). For transitions moving the tape head *right*, this order is reversed (see figs. 12 to 15).

Left tape head moves fig. 8 shows the next-cell instructions for transition $(a, \sqcup) \to (b, \sqcup, L)$, for the special case when the cell to the left of the tape head has the symbol 1. fig. 10 shows the next-cell instructions for the same transition when the cell to the left of the tape head has the symbol 0, and fig. 11 shows the next-cell instructions when the cell to the left has the symbol $\sqcup$. Note that in any given configuration, the same instructions will result in exactly one of the situations depicted in figs. 8, 10 and 11. Once these next-cell instructions are applied, the leftmost domain of the previous cell will serve as a toehold for the previous-cell instructions that follow in fig. 9.

Right tape head moves For right transitions, the first three instructions are the previous-cell instructions, as shown in fig. 12. The next-cell instructions follow, where fig. 13 shows what happens when the cell to the right of the tape head has a $\sqcup$, fig. 14 shows what happens when the cell to the right has a 0, and fig. 15 shows what happens when the cell to the right has a 1. As with the left-cell instructions, the next-cell situations depicted in figs. 13 to 15 are disjoint.

Final deprotecting instruction fig. 7 shows the final deprotecting instruction, which removes the post-plug strand put in place during the last instruction of the non-inert instruction sublists. This puts the register back into a "clean" configuration representing a Turing machine configuration, such as those shown in fig. 4, opening up new toeholds for the next iteration of instructions.

3.4 Complexity of construction

A common metric of "complexity" of DNA systems is the number of unique domains they require. Fewer is better because it is a nontrivial task to design *orthogonal* domains: domains that, if they are not complementary, will have low binding affinity. Low domain complexity is particular important in SIMD||DNA, where each domain is considered "toehold-length": sufficiently short (5-7 bases) that the off-rate of a strand bound by a single domain is large enough to detach in a short amount of time. However, there are only $4^7 = 16384$ DNA sequences of length 7. In practice even fewer are available: half are complementary to the other half, leaving only 8192 available to assign to the unstarred versions of each domain. DNA sequence design heurtistics such as the "3-letter code" (using only A,T,C for forward strands, thus only A,T,G for the register and reverse strands) reduce this number to $3^7 = 2187$. Reasonable design constraints, e.g., avoiding using almost equal domains such as $5'$-AAAAAAT-$3'$ and $5'$-AAAAAAA-$3'$, which both bind almost equally strongly to $3'$-TTTTTTT-$5'$, further limit the set of available domains.

Our construction uses $d = 2t + 8$ total unique DNA domains (which repeat throughout the register), where t is the number of transitions of the simulated Turing machine. While each transition is represented by 2 domains, the construction can be altered to use more domains in order to prevent transition strands from spontaneously dissociating from the register. To simulate a Turing machine with space bound s, the register has s "cells", where each cell is simply a copy of one each of the d domains $1, \dots d$. Thus, if each domain consists of k nucleotides, the register strand has $k \cdot s \cdot (2t+8)$ total nucleotides. There is an 11-state, 3-symbol universal Turing machine (directly simulating another Turing machine) with 32 transitions [5], giving $2 \cdot 32 + 8 = 72$ total domains required in the worst case. However, specialized non-universal Turing machines with a smaller number of transitions (for example the 5-transition binary incrementor of fig. 4) could accomplish many computationally sophisticated tasks.

Each Turing machine transition can be represented by approximately 16 SIMD||DNA instructions. The exact number varies depending on the specific properties of the transition in question, such as the the direction of the tape head, the symbol to be written, and whether any of the possible next configurations are halting or not. This range has constant upper and lower bounds, however, so the total number of instructions is in $O(t)$, where t is the number of transitions in the Turing machine. Because the size of the cell increases in $O(t)$ due to the transition regions, the number of DNA strands present in some instructions also scales by a factor of $O(t)$, most notably the instructions that cause a cascade of toehold exchanges, such as instructions 3 to 7 in figs. 12 and 13.

4 Conclusion

Our construction, like the Rule 110 simulation of Wang, Chalk, and Soloveichik [11], is not Turing universal because it simulates a *space-bounded* Turing machine. Truly universal computation should be possible without knowing in advance how much space will be required. An interesting question (raised also in [11]) is whether a suitable augmentation of the SIMD||DNA model could allow Turing-universal computation. This would require unbounded polymers such as those used in the two-stack machine implementation of Qian, Soloveichik, and Winfree [7]. That paper showed Turing universal computation in the case where only a single copy of certain strands are permitted to exist in solution, simulating only a single stack machine at a time, in contrast to the SIMD||DNA model, where we can operate on many registers, each representing their own Turing machine, in parallel.

Technically the strand displacement reactions of the SIMD||DNA model are currently powerful enough to grow arbitrarily large polymers from a fixed set of strands, as in [7], by alternating top and bottom strands, making a long double-helix with nicks on the top *and* bottom. However, it is difficult to see how to use the ability of SIMD||DNA to exploit this to simulate a Turing machine represented in this way. If there are multiple bottom strands, i.e., there is a nick on the bottom, then any strand displacement of top strands, upon reaching this nick, would separate the polymer into two complexes to the left and right of this nick, and the right polymer would be lost in the wash step. One could imagine, however, augmenting the model to allow, for example, 3-arm junctions, which could be used to do strand displacement that crosses over the boundary between two bottom strands without separating them (since they would be joined to each other by a strong domain representing the third arm "below" the main helix).

Although some of the toehold exchanges in the SIMD||DNA model are reversible based on the principles of DNA strand displacement, we make the assumption that the applied instructions are not undone by the displaced strands. This is based on the assumption that the instruction strands are present in a sufficiently high concentration, such that reversal is possible but unlikely. Because multiple registers can be present in the same solution, another possibility to consider is a displaced DNA strand from Register A binding to an open toehold in Register B, such that the attachment is irreversible even with the presence of a high concentration of instruction strands. One open question is to design a system that factors in these possibilities, reducing the likelihood of unexpected strand displacement results.

Another open question is whether a more domain-efficient encoding exists for the Turing machine construction. Given n transitions, $2n$ domains are required to represent them, which has $O(n)$ complexity. However, given d domains, there are 2^{d-1} possible nick patterns among the attached strands, which makes an $O(\log n)$ domain complexity possible in theory.

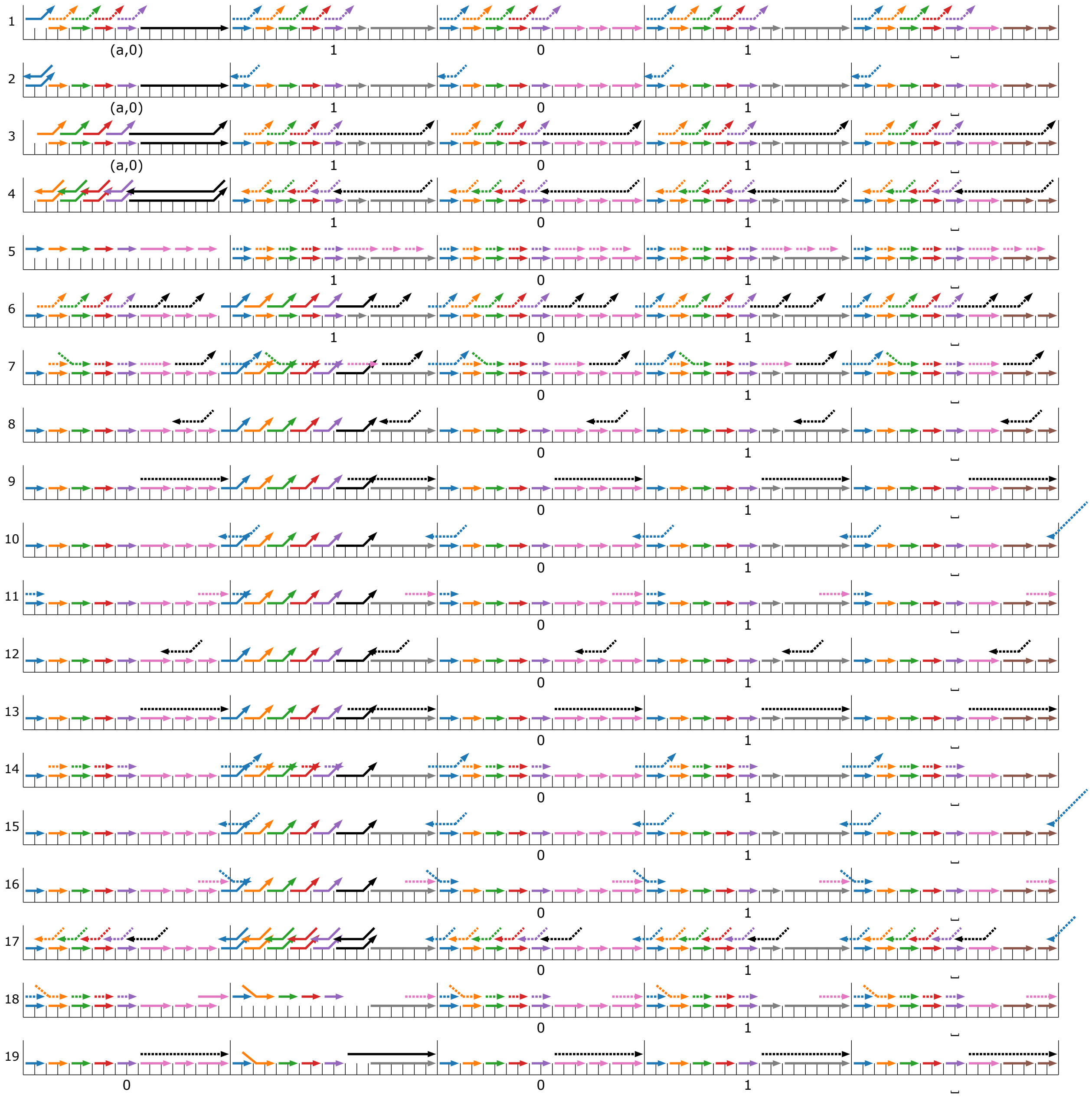

Figure 5: An overview of the first 19 instructions of the construction, which represent the $(a, 0) \to (a, 0, R)$ transition of the Turing machine shown in fig. 2.

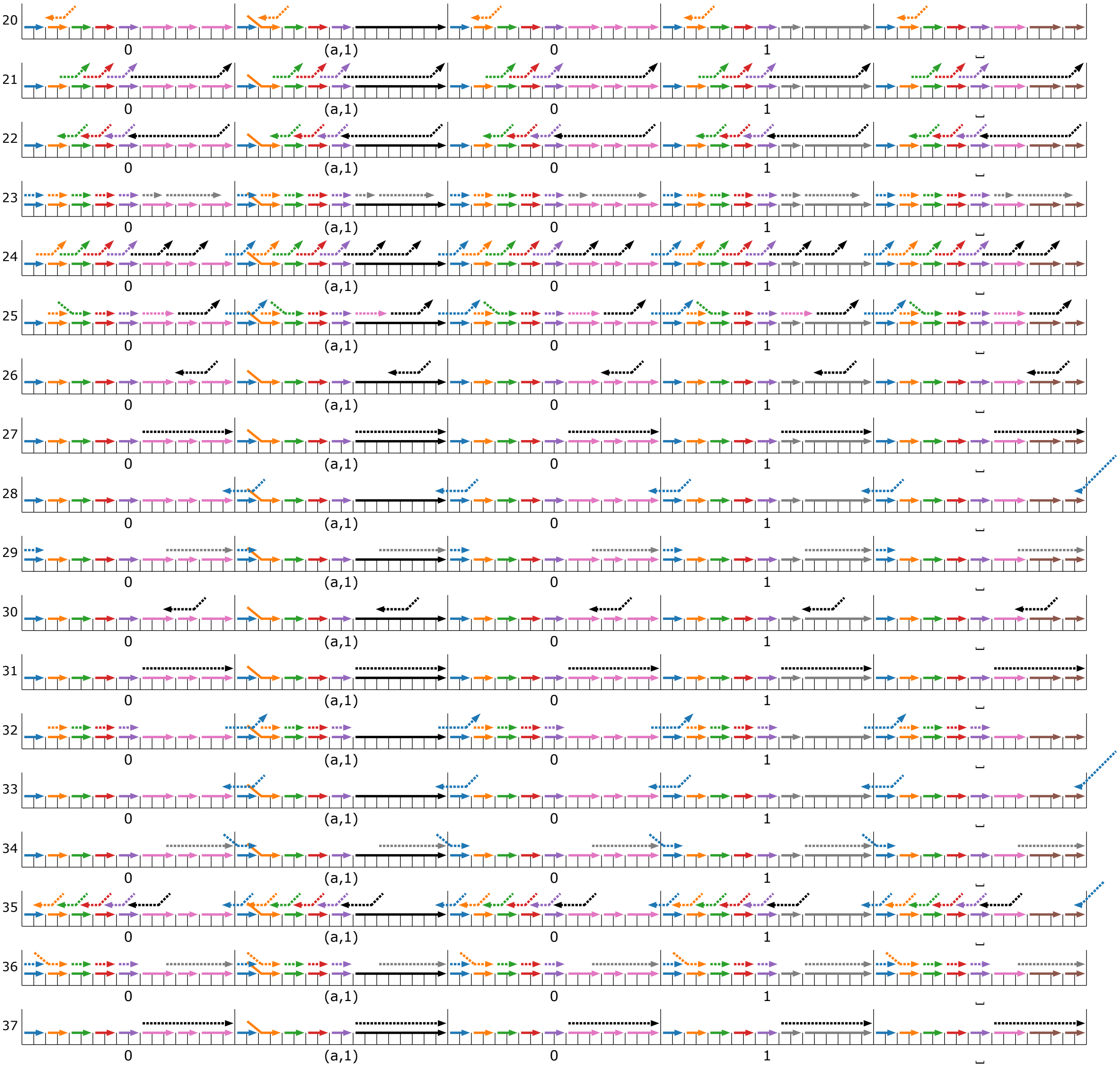

Figure 6: An overview of instructions 20 to 37 of the construction, which represent the $(a, 1) \to (a, 1, R)$ transition of the Turing machine shown in fig. 2. Because the register in this example is not in the $(a, 1)$ configuration, the instructions in this section do not affect the register, as no toehold is present.

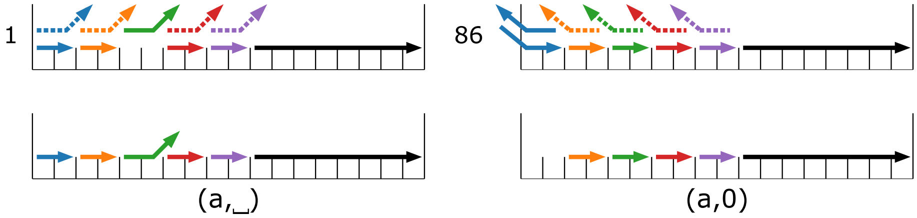

Figure 7: On the left, the first instruction in the whole SIMD||DNA program covers the transition region representing the next applicable transition. The two horizontal rows of strands have the following interpretation: Bottom are strands bound to register, top are instruction strands. Dashed instruction strands will not have an effect on the current cell (but to help verify correctness, they are shown above where they would bind to the register). On the right, the last instruction in the whole SIMD||DNA program, which removes the post-plug strands in each register. In the above example, the post-plug strand covers the $(a, 0)$ transition region, indicating the cell's next Turing machine transition. After this, the entire register is updated to appear as a configuration similar to those in fig. 4.

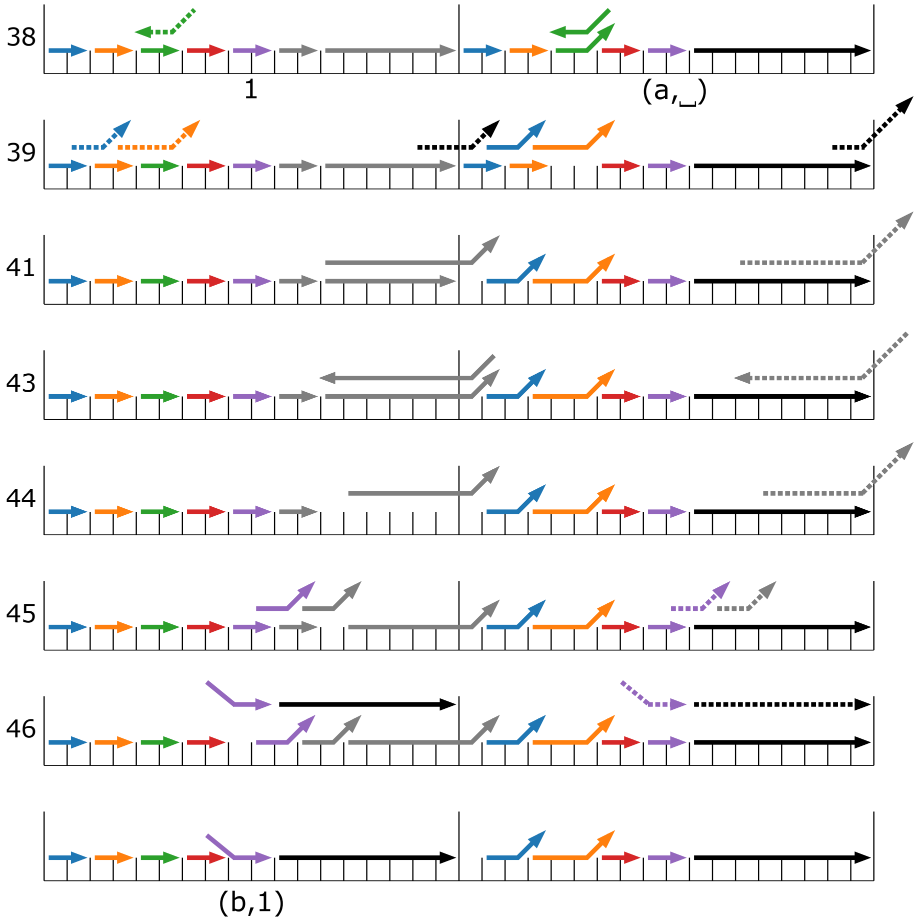

Figure 8: First half of instructions (next-cell instructions) to implement transition $a, \sqcup \rightarrow b, \sqcup, L$, in the case that the cell to the left (where the tape head will move) has a symbol 1. After instruction 46, the left cell (where the tape head will move next) now encodes the next state b and the symbol 1 on the new tape cell. The instructions not shown (40,42,47,48) are inert in this case, but are used when the cell to the left already has a symbol 0 or $\sqcup$ on it instead. figs. 10 and 11 respectively show these cases. fig. 9 shows instructions completing the transition by writing $\sqcup$ over the right cell. In instruction 38, the transition region is first unplugged to expose the toehold. The strands in instruction 39 cascade until the left cell's symbol region, allowing the instructions to branch out depending on the tape content of the left cell. The remaining instructions process the left cell so that it encodes the next configuration. In instruction 46, a special post-plug strand whose leftmost domain is orthogonal is attached to the region that represents the next configuration; this strand will be removed once all transitions have been processed. The angled dashed strands to the right of the register indicate that no cells are present to the right of the rightmost cell; if there were another cell, then one more domain of each of these strands would be horizontal, bound to the leftmost domain of the cell to the right (just as with their solid counterparts to the left).

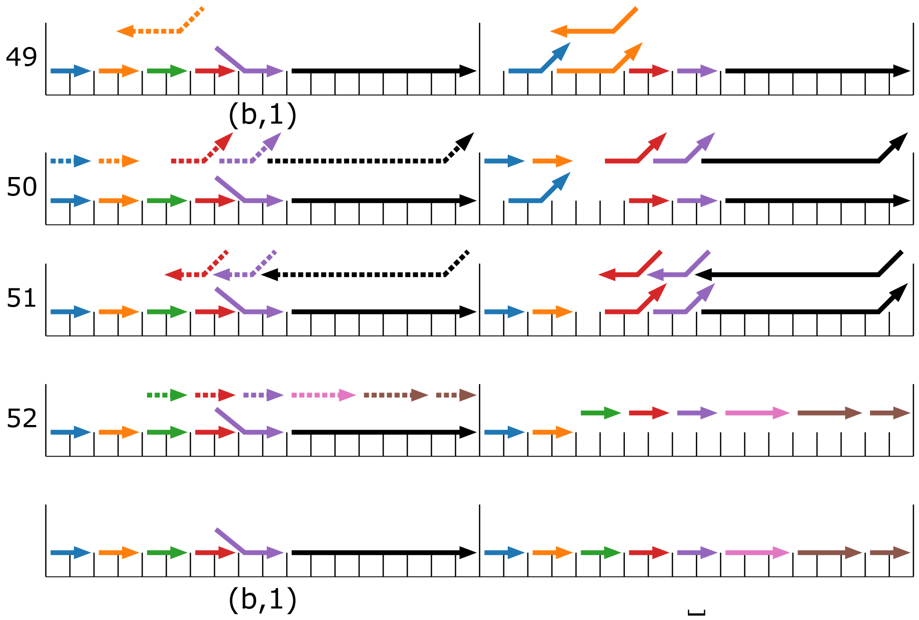

Figure 9: Second half (previous-cell instructions) of transition $a, \sqcup \to b, \sqcup, L$ whose first 11 instructions are shown in fig. 8, fig. 10, and fig. 11; these instructions write $\sqcup$ over the old cell (right of fig. 8) where the tape head was at the start of the transition. Slight variations of instruction 15 write 0 or 1 instead of $\sqcup$.

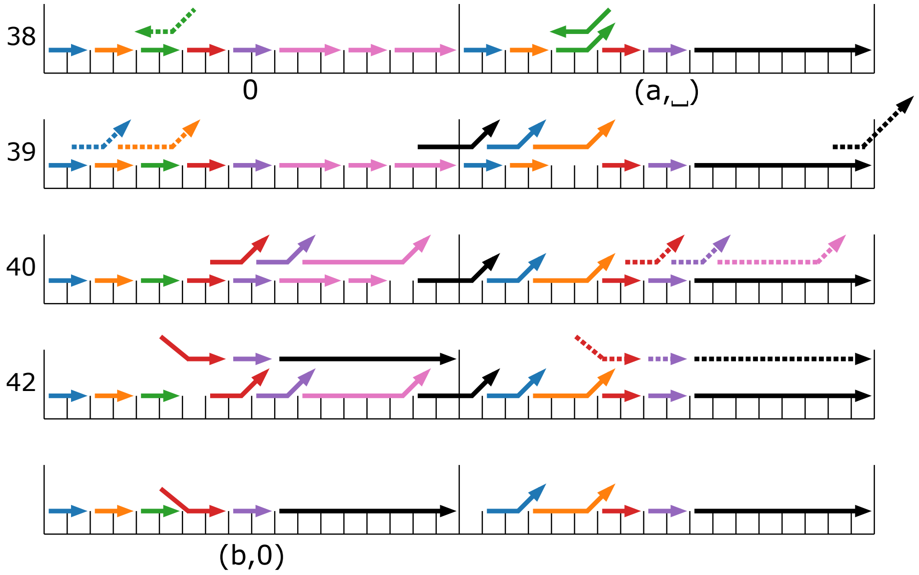

Figure 10: The next-cell instructions of the transition in fig. 8 that are applicable when the cell to the left of the tape head has a 0.

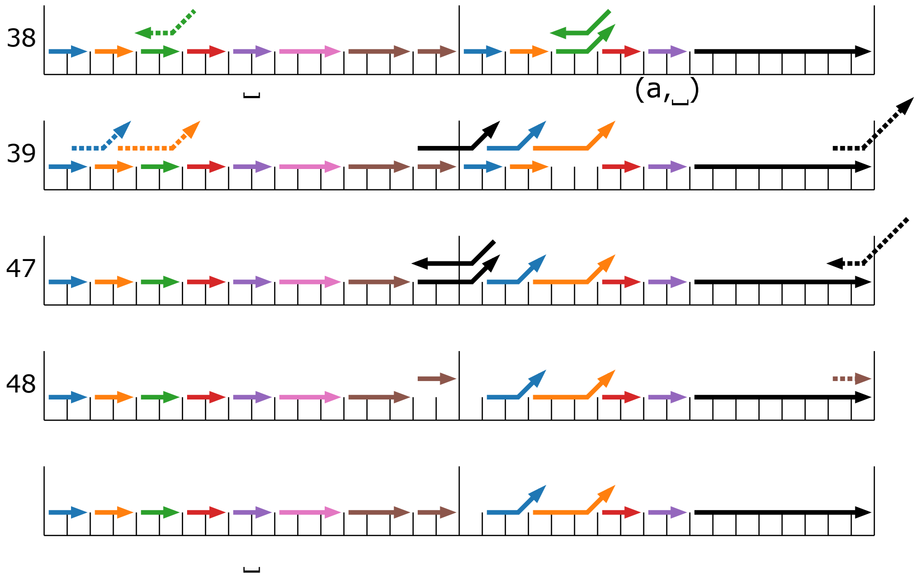

Figure 11: The next-cell instructions of the transition in fig. 8 that are applicable when the cell to the left of the tape head has a ␣. Because no transition exists for the state-symbol pair $(b, ␣)$, the left cell is left unchanged.

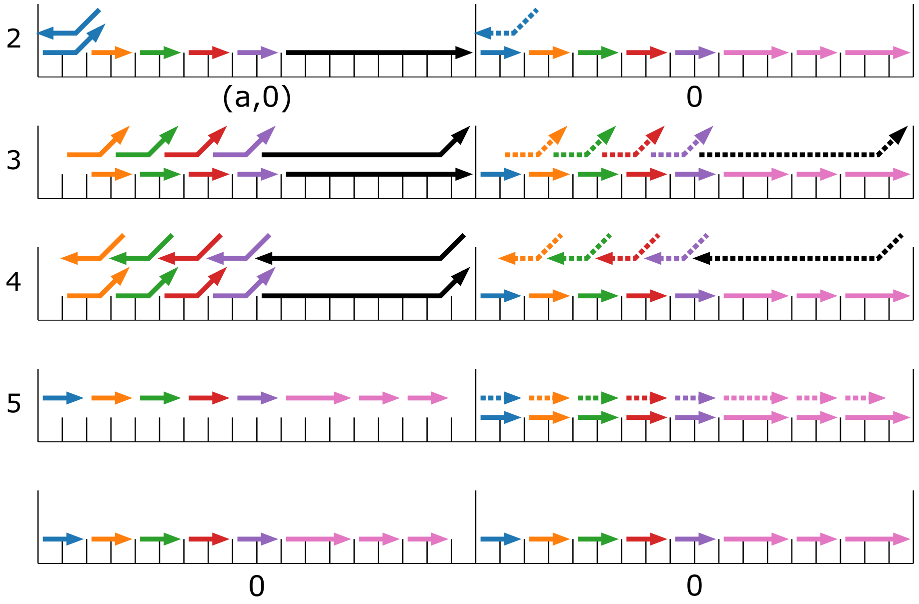

Figure 12: First half (previous-cell instructions) of transition $a, 0 \rightarrow a, 0, R$. Instruction 3's strands cascade to the right side of the current cell, while instruction 4 removes the previously introduced strands. Instruction 5 then covers up all the transition regions and adds the strands of the symbol to be written on the current cell (0 in this example), but leaving the rightmost domain exposed to act as a toehold for the next instructions.

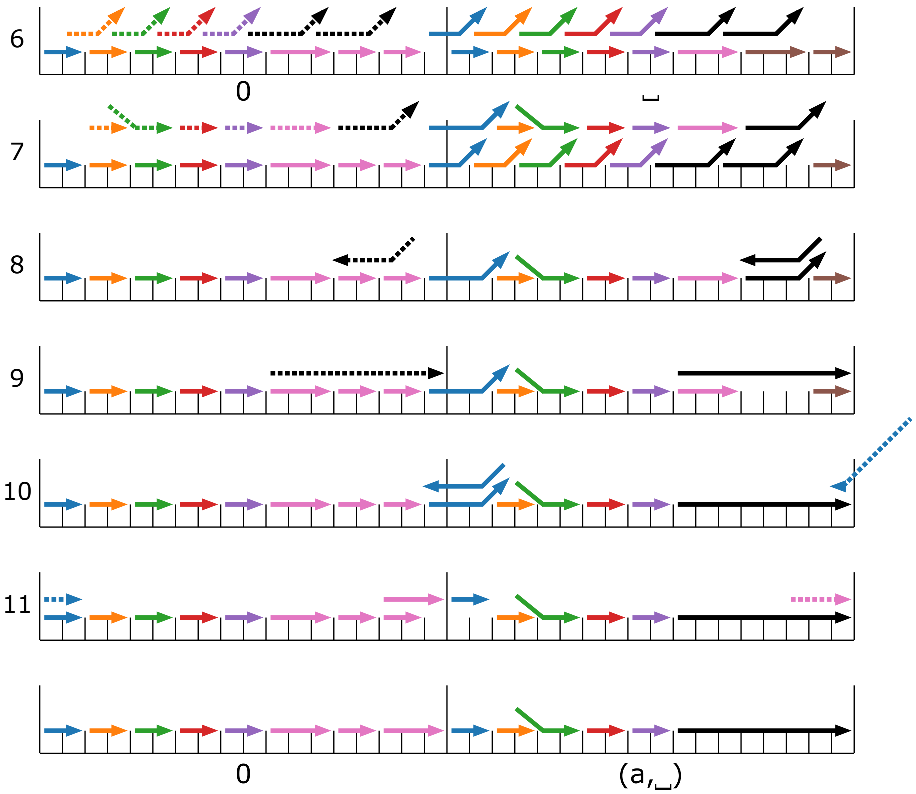

Figure 13: The next-cell instructions for when the right cell contains a blank. The Turing machine enters the $(a, \textvisiblespace)$ configuration afterward.

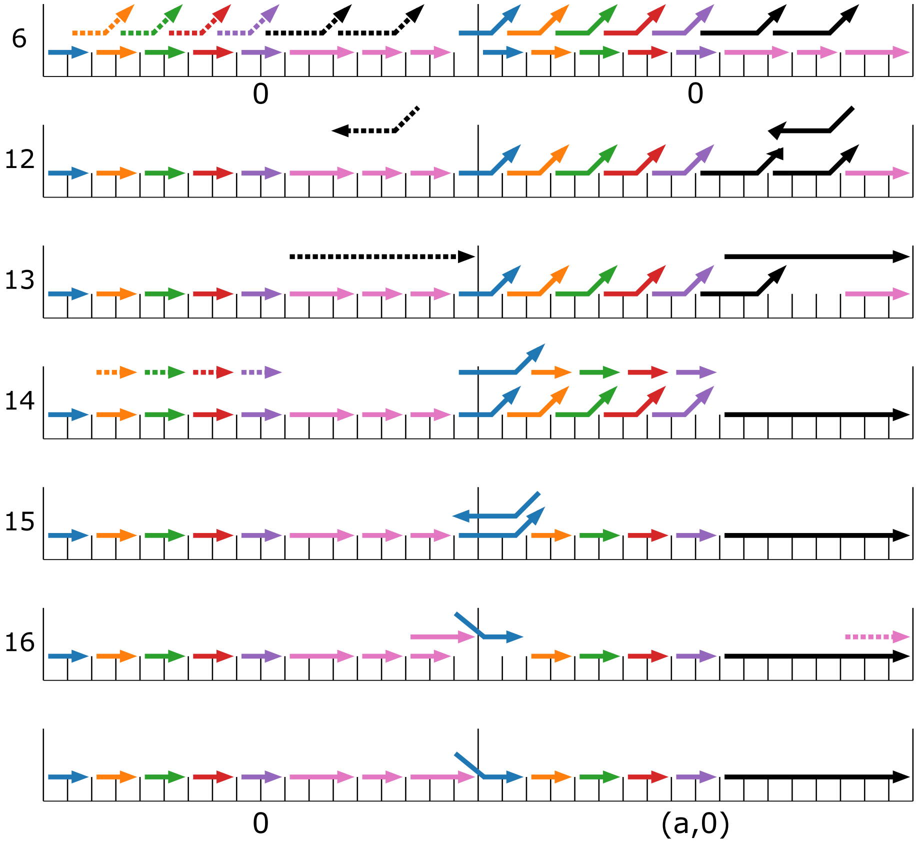

Figure 14: The next-cell instructions for when the right cell contains the symbol 0, entering the $(a, 0)$ configuration.

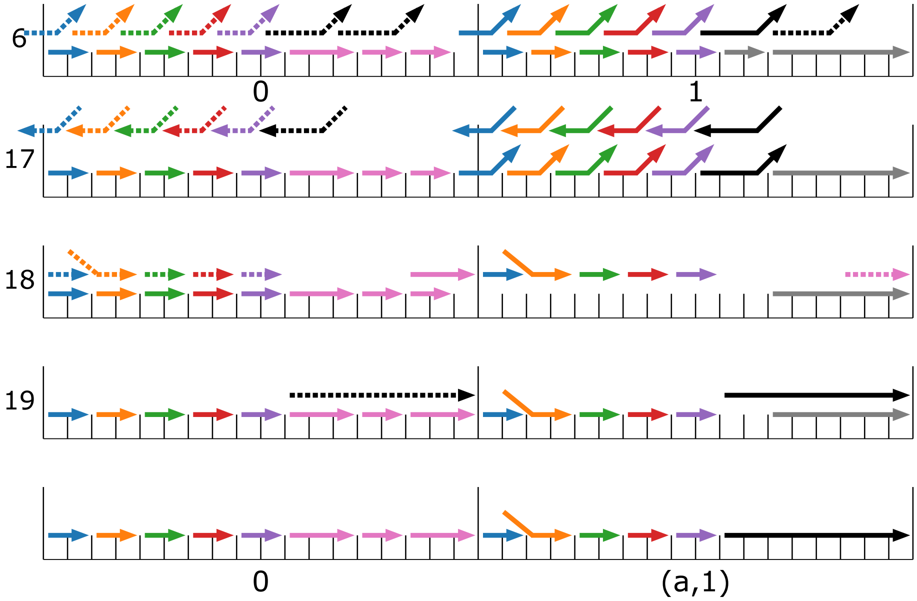

Figure 15: The next-cell instructions for when the right cell contains the symbol 1, entering the $(a, 1)$ configuration.

References

[1] James Bornholt, Randolph Lopez, Douglas M Carmean, Luis Ceze, Georg Seelig, and Karin Strauss. A DNA-based archival storage system. In *ASPLOS 2016: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 637–649, 2016.

[2] George M Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science*, 337(6102):1628–1628, 2012.

[3] Matthew Cook. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.

[4] 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.

[5] Turlough Neary and Damien Woods. Small fast universal Turing machines. *Theoretical Computer Science*, 362(1-3):171–195, 2006.

[6] Lee Organick, Siena Dumas Ang, Yuan-Jyue Chen, Randolph Lopez, Sergey Yekhanin, Konstantin Makarychev, Miklos Z Racz, Govinda Kamath, Parikshit Gopalan, Bichlien Nguyen, Christopher N Takahashi, Sharon Newman, Hsing-Yeh Parker, Cyrus Rashtchian, Kendall Stewart, Gagan Gupta, Robert Carlson, John Mulligan, Douglas Carmean, Georg Seelig, Luis Ceze, and Karin Strauss. Random access in large-scale DNA data storage. *Nature Biotechnology*, 36(3):242, 2018.

[7] Lulu Qian, David Soloveichik, and Erik Winfree. Efficient Turing-universal computation with DNA polymers. In *International Workshop on DNA-Based Computers*, pages 123–140. Springer, 2010.

[8] Georg Seelig, David Soloveichik, David Yu Zhang, and Erik Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, 2006.

[9] SIMD||DNA simulator.
source code: https://github.com/UC-Davis-molecular-computing/simd-dna, 2021.

[10] S Kasra Tabatabaei, Boya Wang, Nagendra Bala Murali Athreya, Behnam Enghiad, Alvaro Gonzalo Hernandez, Christopher J Fields, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards for storing data on native DNA sequences via enzymatic nicking. *Nature Communications*, 11(1):1–10, 2020.

[11] Boya Wang, Cameron Chalk, and David Soloveichik. SIMD||DNA: Single instruction, multiple data computation with DNA strand displacement cascades. In *DNA 2019: International Conference on DNA Computing and Molecular Programming*, pages 219–235, 2019.

A Appendix

The following figures show the full instruction set on a register whose current value is 01␣, where the tape head is on 1 and the current state is a. Because the $(a, 1)$ transition is represented in this configuration, the instruction sublists that represent the other transitions (figs. 16 and 18 to 20) are fully inert. Inert instructions are marked with a red cross underneath the instruction number.

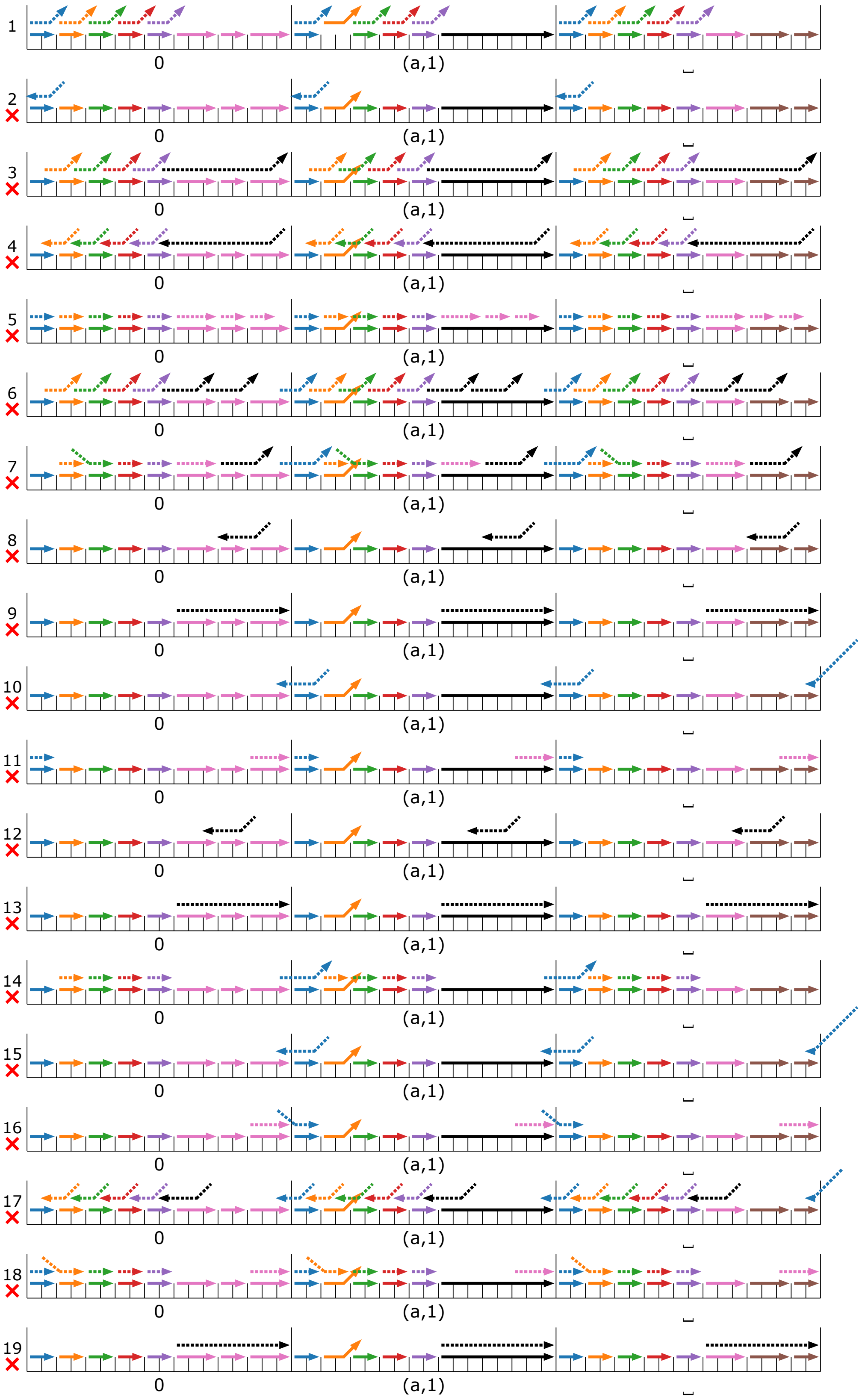

Figure 16: The initial pre-plug instruction, as well as the $(a, 0)$ instruction sublist for a register whose next transition is $(a, 1)$.

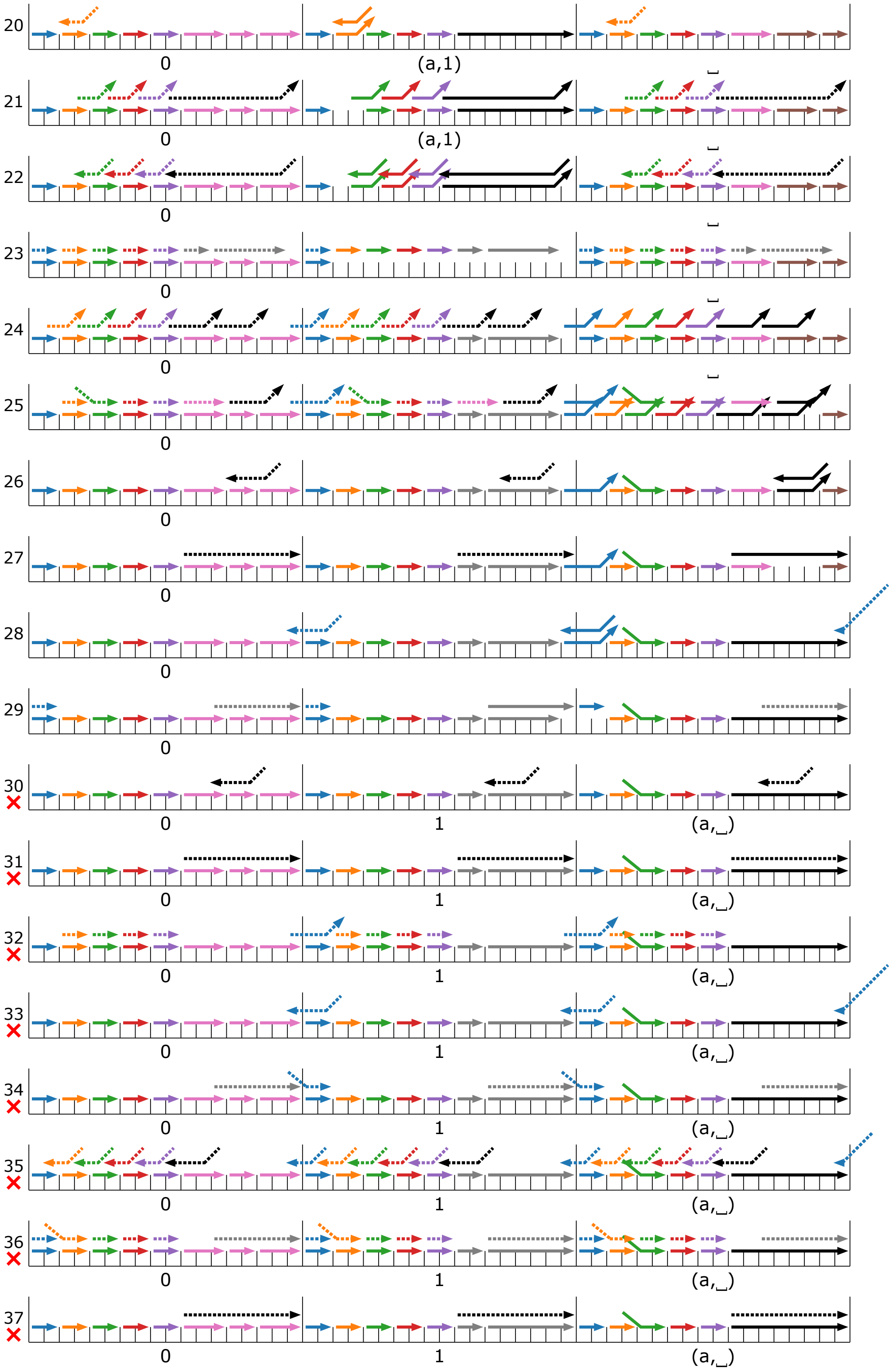

Figure 17: The $(a, 1)$ instruction sublist for a register whose next transition is $(a, 1)$.

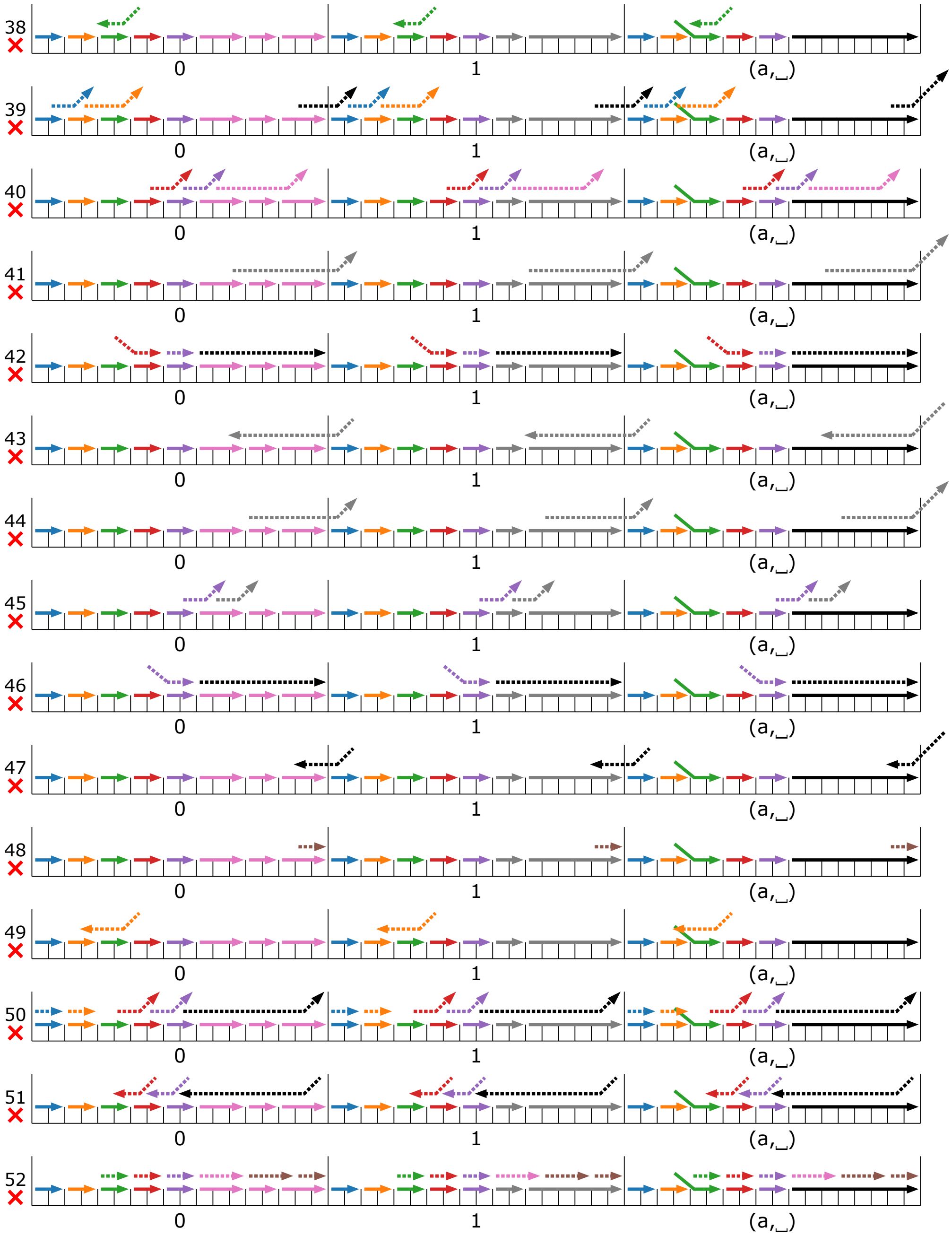

Figure 18: The $(a, \sqcup)$ instruction sublist after the register was processed by the $(a, 1)$ instruction sublist. Even though the register is now in the $(a, \sqcup)$ configuration, the post-plug strand introduced in instruction 25 prevents the register from being updated a second time using the $(a, \sqcup)$ instructions.

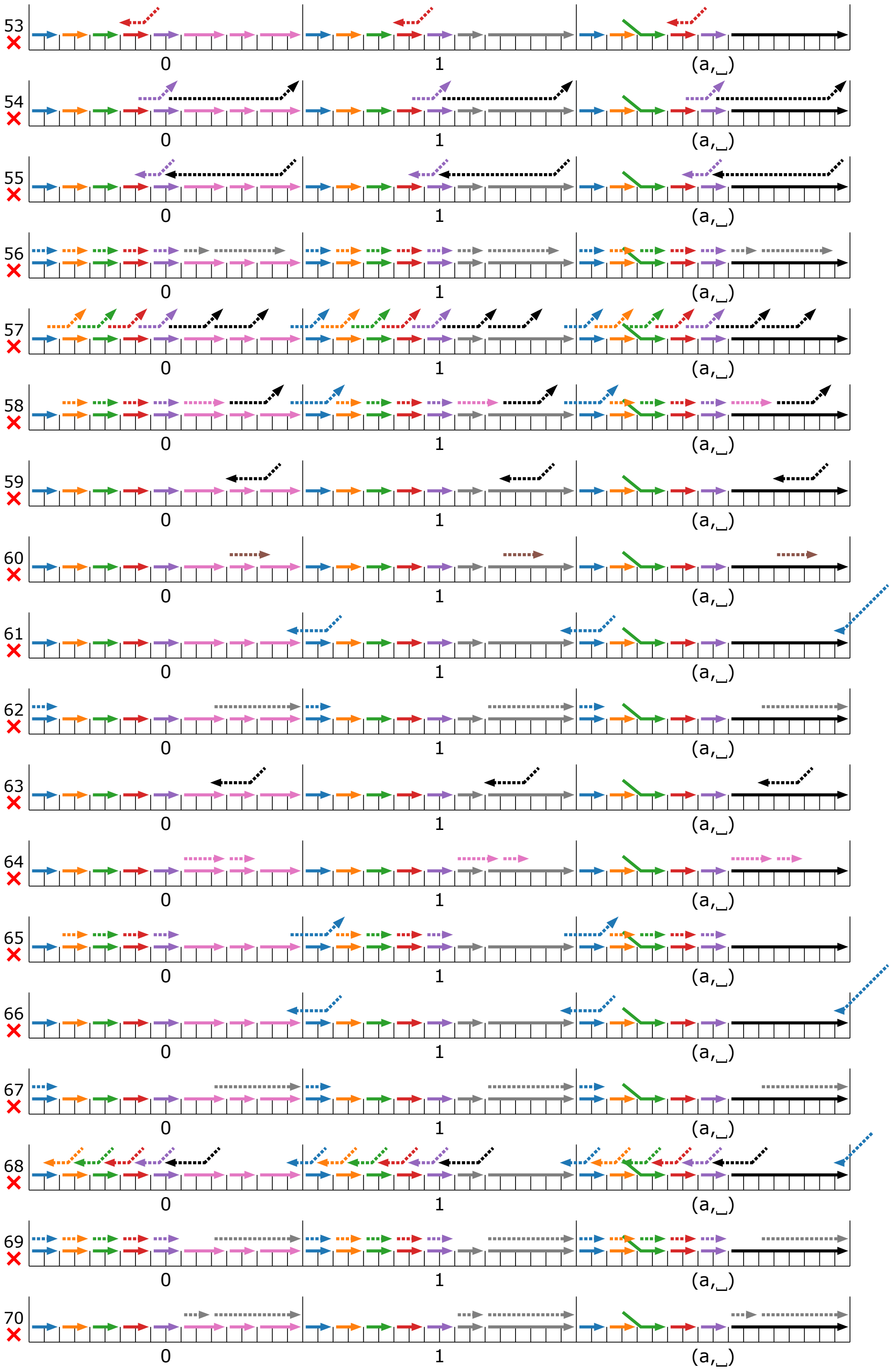

Figure 19: The $(b, 0)$ instruction sublist after the register was processed by the $(a, 1)$ instruction sublist.

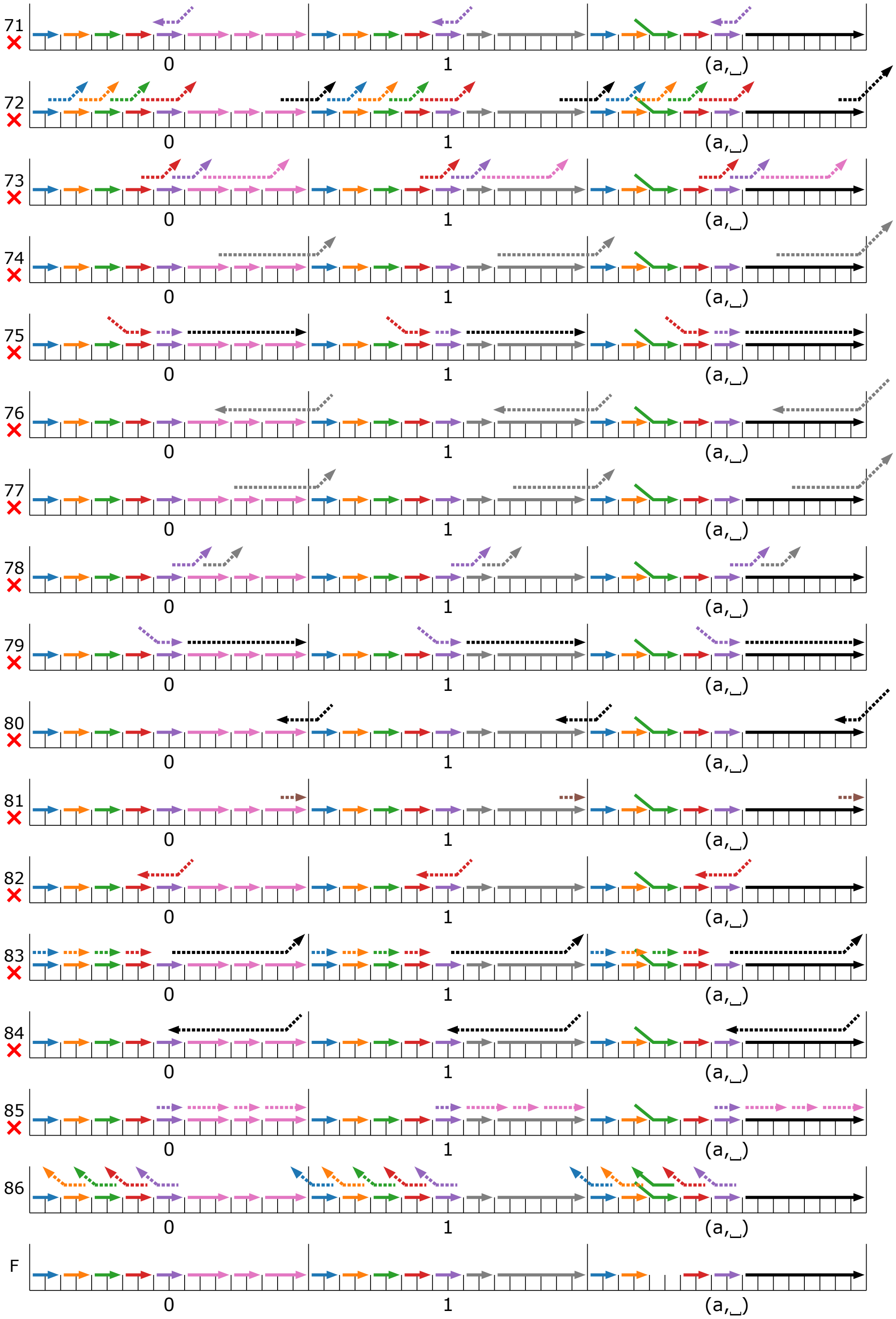

Figure 20: The $(b, 1)$ instruction sublist after the register was processed by the $(a, 1)$ instruction sublist, as well as the final deprotecting instruction that removes the post-plug strands.