Fast Arithmetic in Algorithmic Self-Assembly

Alexandra Keenan[*]   Robert Schweller[*]   Michael Sherman[*]   Xingsi Zhong[*]

Abstract

In this paper we consider the time complexity of adding two n-bit numbers together within the tile self-assembly model. The (abstract) tile assembly model is a mathematical model of self-assembly in which system components are square tiles with different glue types assigned to tile edges. Assembly is driven by the attachment of singleton tiles to a growing seed assembly when the net force of glue attraction for a tile exceeds some fixed threshold. Within this frame work, we examine the time complexity of computing the sum or product of 2 n-bit numbers, where the input numbers are encoded in an initial seed assembly, and the output is encoded in the final, terminal assembly of the system. We show that the problems of addition and multiplication have worst case lower bounds of $\Omega(\sqrt{n})$ in 2D assembly, and $\Omega(\sqrt[3]{n})$ in 3D assembly. In the case of addition, we design algorithms for both 2D and 3D that meet this bound with worst case run times of $O(\sqrt{n})$ and $O(\sqrt[3]{n})$ respectively, which beats the previous best known upper bound of $O(n)$. Further, we consider average case complexity of addition over uniformly distributed n-bit strings and show how to achieve $O(\log n)$ average case time with a simultaneous $O(\sqrt{n})$ worst case run time in 2D. For multiplication, we present an $O(n^{5/6})$ time multiplication algorithm which works in 3D, which beats the previous best known upper bound of $O(n)$. As additional evidence for the speed of our algorithms, we implement our addition algorithms, along with the simpler $O(n)$ time addition algorithm, into a probabilistic run-time simulator and compare the timing results.

Table 1: Summary of results.

	Worst Case		Average Case
	UB	LB	
Addition(2D)	$\Theta(\sqrt{n})$		$O(\log n)$
	(Thm.6.1)	(Thm.B.4)	(Thm.6.1)
Addition(3D)	$\Theta(\sqrt[3]{n})$		$O(\log n)$
	(Thm.7.1)	(Thm. B.4)	(Thm.7.1)
Multiplication (3D)	$O(n^{5/6})$	$\Omega(n^{1/3})$	-
	(Thm. 8.1)	(Thm. B.5)	
Previous Best Addition(2D)	$O(n)$ (See [5])	-	-
Previous Best Multiplication(2D)	$O(n)$ (See [5])	-	-

1   Introduction.

Self-assembly is the process by which systems of simple objects autonomously organize themselves through local interactions into larger, more complex objects. Self-assembly processes are abundant in nature and serve as the basis for biological growth and replication. Understanding how to design and efficiently program molecular self-assembly systems promises to be fundamental for the future of nanotechnology. One particular direction of interest is the design of molecular computing systems for the efficient solution of fundamental computational problems. In this paper we study the complexity of computing arithmetic primitives within a well studied model of algorithmic self-assembly, the abstract tile assembly model.

The abstract tile assembly model (aTAM) models system monomers with four sided Wang tiles with glue types assigned to each edge. Assembly proceeds by tiles attaching, one by one, to a growing initial seed assembly whenever the net glue strength of attachment exceeds some fixed temperature threshold. The aTAM has been shown to be capable of universal computation [18], and research leveraging this computational power has lead to efficient assembly of complex geometric shapes and patterns with a number of recent results in FOCS, SODA, and ICALP $[1, 6, 7, 9–15, 17]$. This universality also allows the model to serve directly as a model for computation in which an input bit string is encoded into an initial assembly. The process of self-assembly and the final produced terminal assembly represent the computation of a function on the given input. Given this framework, it is natural to ask how fast a given function can be computed in this model. Tile assembly systems can be designed to take advantage of massive parallelism when multiple tiles attach at distinct positions in parallel, opening the possibility for faster algorithms than what can be achieved in more traditional computational models. On the other hand, tile assembly algorithms must use up geometric space to perform computation, and must pay substantial time costs when communicating information between two physically distant bits. This creates a host of challenges unique to this physically motivated computational model that warrant careful study.

In this paper we consider the time complexity of adding or multiplying two n-bit numbers within the abstract tile assembly model. We show that addition and multiplication have worst-case lower bounds of $\Omega(\sqrt{n})$ time in 2D and $\Omega(\sqrt[3]{n})$ time in 3D. These lower bounds are derived by a reduction from a simple problem we term the *communication* problem in which two distant bits must compute the AND function between themselves. This general reduction technique can likely be applied to a number of problems and yields key insights into how one might design a sub-linear time solution to such problems. We in turn show that for the problem of addition these lower bounds are matched by corresponding worst case $O(\sqrt{n})$ and $O(\sqrt[3]{n})$ run time algorithms, respectively, which improves upon the previous best known result of $O(n)$ [5]. We then consider the average case complexity of addition given two uniformly generated random n-bit numbers and construct a $O(\log n)$ average case time algorithm that achieves simultaneous worst case run time $O(\sqrt{n})$ in 2D. To the best of our knowledge this is the first tile assembly algorithm proposed for efficient average case adding. Finally, we present a 3D algorithm that achieves $O(n^{5/6})$ time for multiplication which beats the previous fastest algorithm of time $O(n)$ [5] (which works in 2D). We analyze our algorithms under two established timing models described in Section 2.4.

Temperature = 2   Temperature = 2

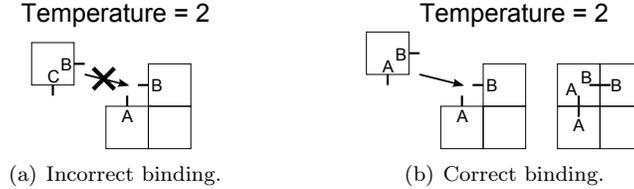(a) Incorrect binding.   (b) Correct binding.

Figure 1: Cooperative tile binding in the aTAM.

Our results are summarized in Table 1. In addition to our analytical results, tile self-assembly software simulations were conducted to visualize the diverse approaches to fast arithmetic presented in this paper, as well as to compare them to previous work. The adder tile constructions described in Sections 4, 5 and 6, and the previous best known algorithm from [5] were simulated using the two timing models described in Section 2.4. These results can be seen in the graphs in Section 9.

2 Definitions

2.1 Basic Notation.

Let $\mathbb{N}_n$ denote the set $\{1, \dots, n\}$ and let $\mathbb{Z}_n$ denote the set $\{0, \dots, n-1\}$. Consider two points $p, q \in \mathbb{Z}^d$, $p = (p_1, \dots p_d)$, $q = (q_1, \dots, q_d)$. Define $\Delta_{p,q} \triangleq \max_{1 \le i \le d} \{|p_i - q_i|\}$.

2.2 Abstract Tile Assembly Model.

Tiles. Consider some alphabet of symbols Π called the *glue types*. A tile is a finite edge polygon (polyhedron in the case of a 3D generalization) with some finite subset of border points each assigned some glue type from Π. Further, each glue type $g \in \Pi$ has some non-negative integer strength $str(g)$. For each tile t we also associate a finite string *label* (typically "0", or "1", or the empty label in this paper), denoted by label(t), which allows the classification of tiles by their labels. In this paper we consider a special class of tiles that are unit squares (or unit cubes in 3D) of the same orientation with at most one glue type per face, with each glue being placed exactly in the center of the tile's face. We denote the *location* of a tile to be the point at the center of the square or cube tile. In this paper we focus on tiles at integer locations.

Assemblies. An assembly is a finite set of tiles whose interiors do not overlap. Further, to simplify formalization in this paper, we further require the center of each tile in an assembly to be an integer coordinate (or integer triplet in 3D). If each tile in A is a translation of some tile in a set of tiles T, we say that A is an assembly over tile set T. For a given assembly Υ, define the *bond graph* G_Υ to be the weighted graph in which each element of Υ is a vertex, and the weight of an edge between two tiles is the strength of the overlapping matching glue points between the two tiles. Note that only overlapping glues that are the same type contribute a non-zero weight, whereas overlapping, non-equal glues always contribute zero weight to the bond graph. The property that only equal glue types interact with each other is referred to as the *diagonal glue function* property and is perhaps more feasible than more general glue functions for experimental implementation. An assembly Υ is said to be τ-*stable* for an integer τ if the min-cut of G_Υ is at least τ.

Tile Attachment. Given a tile t, an integer τ, and a τ-stable assembly A, we say that t may attach to A at temperature τ to form A' if there exists a translation t' of t such that $A' = A \bigcup \{t'\}$, and A' is τ-stable. For a tile set T we use notation $A \to_{T,\tau} A'$ to denote that there exists some $t \in T$ that may attach to A to form A' at temperature τ. When T and τ are implied, we simply say $A \to A'$. Further, we say that $A \to^* A'$ if there exists a finite sequence of assemblies $\langle A_1 \dots A_k \rangle$ such that $A \to A_1 \to \dots \to A_k \to A'$.

2

Tile Systems. A tile system $\Gamma = (T, S, \tau)$ is an ordered triplet consisting of a set of tiles T referred to as the system's *tile set*, a τ-stable assembly S referred to as the system's *seed* assembly, and a positive integer τ referred to as the system's *temperature*. A tile system $\Gamma = (T, S, \tau)$ has an associated set of *producible* assemblies, $\texttt{PROD}_\Gamma$, which define what assemblies can grow from the initial seed S by any sequence of temperature τ tile attachments from T. Formally, $S \in \texttt{PROD}_\Gamma$ as a base case producible assembly. Further, for any $A \in \texttt{PROD}_\Gamma$, if $A \to_{T,\tau} A'$, then $A' \in \texttt{PROD}_\Gamma$. That is, assembly S is producible, and for any producible assembly A, if A can grow into A', then A' is also producible. We further define the set of *terminal* assemblies $\texttt{TERM}_\Gamma$ to be the subset of $\texttt{PROD}_\Gamma$ containing all producible assemblies that have no attachable tile from T at temperature τ. Conceptually, $\texttt{TERM}_\Gamma$ represents the final collection of output assemblies that are built from Γ given enough time for all assemblies to reach a final, terminal state. General tile systems may have terminal assembly sets containing 0, finite, or infinitely many distinct assemblies. Systems with exactly 1 terminal assembly are said to be *deterministic*. For a deterministic tile system Γ, we say Γ *uniquely assembles* assembly A if $\texttt{TERM}_\Gamma = \{A\}$. In this paper, we focus exclusively on deterministic systems. For recent consideration of non-determinism in tile self-assembly see [6, 7, 9, 11, 15].

2.3 Problem Description.

We now formalize what we mean for a tile self-assembly system to compute a function. To do this we present the concept of a *tile assembly computer* (TAC) which consists of a tile set and temperature parameter, along with input and output *templates*. The input template serves as a seed structure with a sequence of *wildcard positions* for which tiles of label "0" and "1" may be placed to construct an initial seed assembly. An output template is a sequence of points denoting locations for which the TAC, when grown from a filled in template, will place tiles with "0" and "1" labels that denote the output bit string. A TAC then is said to compute a function f if for any seed assembly derived by plugging in a bitstring b, the terminal assembly of the system with tile set T and temperature τ will be such that the value of $f(b)$ is encoded in the sequence of tiles placed according to the locations of the output template. We now develop the formal definition of the TAC concept. We note that the formality in the input template is of substantial importance. Simpler definitions which map seeds to input bit strings, and terminal assemblies to output bitstrings, are problematic in that they allow for the possibility of encoding the computation of function f in the seed structure. Even something as innocuous sounding as allowing more than a single type of "0" or "1" tile as an input bit has the subtle issue of allowing pre-computing of f[1].

Input Template. Consider a tile set T containing exactly one tile t_0 with label "0", and one tile t_1 with label "1". An n-bit input template over tile set T is an ordered pair $U = (R, B(i))$, where R is an assembly over $T - \{t_0, t_1\}$, $B : \mathbb{N}_n \to \mathbb{Z}^2$, and $B(i)$ is not the position of any tile in R for any i from 1 to n. The sequence of n coordinates denoted by B conceptually denotes "wildcard" tile positions for which copies of t_0 and t_1 will be filled in for any instance of the template. For notation we define assembly U_b over T, for bit string $b = b_1, \ldots b_n$, to be the assembly consisting of assembly R unioned with a set of n tiles t^i for i from 1 to n, where t^i is equal a translation of tile $t_{b(i)}$ to position $B(i)$. That is, U_b is the assembly R with each position $B(i)$ tiled with either t_0 or t_1 according to the value of b_i.

Output Template. A k-bit output template is simply a sequence of k coordinates denoted by function $C : N_k \to \mathbb{Z}^2$. For an output template V, an assembly A over T is said to represent binary string $c = c_1, \ldots, c_k$ over template V if the tile at position $C(i)$ in A has label c_i for all i from 1 to k. Note that output template solutions are much looser than input templates in that there may be multiple tiles with labels "1" and "0", and there are no restrictions on the assembly outside of the k specified wildcard positions. The strictness for the input template stems from the fact that the input must "look the same" in all ways except for the explicit input bit patterns. If this were not the case, it would likely be possible to encode the solution to the computational problem into the input template, resulting is a trivial solution.

---

[1]This subtle issue seems to exist with some previous formulations of tile assembly computation.

Function Computing Problem. A *tile assembly computer* (TAC) is an ordered quadruple $\Im = (T, U, V, \tau)$ where T is a tile set, U is an n-bit input template, and V is a k-bit output template. A TAC is said to compute function $f : \mathbb{Z}_2^n \to \mathbb{Z}_2^k$ if for any $b \in \mathbb{Z}_2^n$ and $c \in \mathbb{Z}_2^k$ such that $f(b) = c$, then the tile system $\Gamma_{\Im,b} = (T, U_b, \tau)$ uniquely assembles an assembly A which represents c over template V. For a TAC $\Im$ that computes the function $f : Z_2^{2n} \to Z_2^{n+1}$ where $f(r_1 \ldots r_{2n}) = r_1 \ldots r_n + r_{n+1} \ldots r_{2n}$, we say that $\Im$ is an n-bit *adder* TAC with inputs $a = r_1 \ldots r_n$ and $b = r_{n+1} \ldots r_{2n}$. An n-bit *multiplier* TAC is defined similarly.

2.4 Run Time Models

We analyze the complexity of self-assembly arithmetic under two established run time models for tile self-asembly: the *parallel* time model [4,5] and the *continuous* time model [2–4,8]. Informally, the parallel time model simply adds, in parallel, all singleton tiles that are attachable to a given assembly within a single time step. The continuous time model, in contrast, models the time taken for a single tile to attach as an exponentially distributed random variable. The parallelism of the continuous time models stems from the fact that if an assembly has a large number of attachable positions, then the *first* tile to attach will be an exponentially distributed random variable with rate proportional to the number of attachment sites, implying that a larger number of open positions will speed up the expected time for the next attachment. Technical definitions for each run time model are provided in Section A. For a deterministic tile system Γ, we use notation ρ_Γ and ς_Γ to denote the parallel and continuous run time of Γ respectively. When not otherwise specified, we use the term *run time* to refer to parallel run time by default.

3 Lower Bound for Long Distance Communication

To prove lower bounds for addition and multiplication in 2D and 3D, we do the following. First, we consider two identical tile systems with the exception of their respective seed assemblies which differ in exactly one tile location. We show in Lemma B.1 that after Δ time steps, all positions more than Δ distance from the point of initial difference of the assemblies must be identical among the two systems. We then consider the *communication* problem, formally defined in Section B.1, in which we compute the AND function of two input bits under the assumption that the input template for the problem separates the two bits by distance Δ. For such a problem, we know that the output position of the solution bit must be at least distance $\frac{1}{2}\Delta$ from one of the two input bits. As the correct output for the AND function must be a function of both bits, Lemma B.1 implies that at least $\frac{1}{2}\Delta$ steps are required to guarantee a correct solution as argued in Theorem B.2.

With the lower bound of $\frac{1}{2}\Delta$ established for the communication problem, we move on to the problems of addition and multiplication of n-bit numbers. We show how the communication problem can be reduced to these problems, thereby yielding corresponding lower bounds. In particular, consider the addition problem in 2D. As the input template must contain positions for $2n$ bits, in 2D it must be the case that some pair of bits are separated by at least $\Omega(\sqrt{n})$ distance according to Lemma B.3. Focusing on this pair of bit positions in the addition template, we can create a corresponding communication problem template with the same two positions as input. To guarantee the correct output, we hard code the remaining bit positions of the addition template such that the addition algorithm is guaranteed to place the AND of the desired bit pair in a specific position in the output template, thereby constituting a solution to the $\Delta = \Omega(\sqrt{n})$ communication problem, which implies the addition solution cannot finish faster than $\Omega(\sqrt{n})$ in the worst case. A similar reduction can be applied to multiplication. The precise reductions are detailed in Theorems B.4 and B.5. The result statements are as follows.

Theorem 3.1. Any d-dimension n-bit adder TAC has worst case run-time $\Omega(\sqrt[d]{n})$.

Theorem 3.2. Any d-dimension n-bit multiplier TAC has worst case run-time $\Omega(\sqrt[d]{n})$.
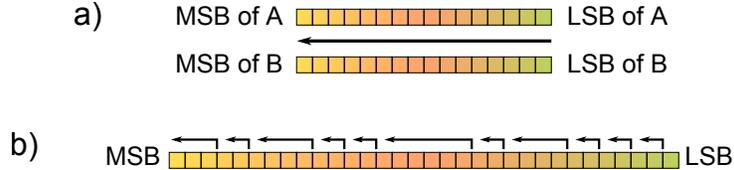
Figure 2: Arrows represent carry origination and propagation direction. a) This schematic represents the previously described $O(n)$ worst case addition for addends A and B [5]. The least significant and most significant bits of A and B are denoted by LSB and MSB, respectively. b) The average case $O(\log n)$ construction described in this paper is shown here. Addends A and B populate the linear assembly with bit A_i immediatly adjacent to B_i. Carry propagation is done in parallel along the length of the assembly.

4 Addition In Average Case Logarithmic Time

We construct an adder TAC that resembles an electronic carry-skip adder in that the carry-out bit for addend pairs where each addend in the pair has the same bit value is generated in a constant number of steps and immediately propagated. When each addend in a pair of addends does not have the same bit value, a carry-out cannot be deduced until the value of the carry-in to the pair of addends is known. When such addends combinations occur in a contiguous sequence, the carry must ripple through the sequence from right-to-left, one step at a time as each position is evaluated. Within these worst-case sequences, our construction resembles an electronic ripple-carry adder. We show that using this approach it is possible to construct an n-bit adder TAC that can perform addition with an average runtime of $O(\log n)$ and a worst-case runtime of $O(n)$.

Lemma 4.1. Consider a non-negative integer N generated uniformly at random from the set $\{0, 1, ..., 2^n-1\}$. The expected length of the longest substring of contiguous ones in the binary expansion of N is $O(\log n)$.

For a detailed discussion of Lemma 4.1 please see Schilling [16].

Theorem 4.2. For any positive integer n, there exists an n-bit adder TAC (tile assembly computer) that has worst case run time $O(n)$ and an average case run time of $O(\log n)$.

The proof of Theorem 4.2 follows from the construction of the adder in Appendices C.1, C.2, and C.3.

5 Optimal $O(\sqrt{n})$ Addition

We show how to construct an adder TAC that achieves a run time of $O(\sqrt{n})$, which matches the lower bound proved in Theorem B.4. This adder TAC closely resembles an electronic carry-select adder in that the addends are divided into sections of size $\sqrt{n}$ and the sum of the addends comprising each is computed for both possible carry-in values. The correct result for the subsection is then selected after a carry-out has been propagated from the previous subsection. Within each subsection, the addition scheme resembles a ripple-carry adder. This construction works well with massive parallelism and allows us to construct an optimal $O(\sqrt{n})$ adder TAC in two dimensions.

Theorem 5.1. There exists a 2D n-bit adder TAC with a worst case run-time of $O(\sqrt{n})$.

The proof of Theorem 5.1 follows from the construction of the tile assembly adder in Section 5.1 and Appendices D.1, D.2, and D.3.

5.1 Construction Overview

Due to space limitations, an overview of the construction is included in this section. A more detailed description of the adder construction (along with the full tile set) can be found in Appendix D.1. Figures

3(a)-3(b) are examples of I/O templates for a 9-bit adder TAC. The inputs to the addition problem in this instance are two 9-bit binary numbers A and B with the least significant bit of A and B represented by A_0 and B_0, respectively. Upon inclusion of the seed assembly (Figure 4a) to the tile set (Figure 18), each pair of bits from A and B are summed (for example, $A_0 + B_0$) (Figure 4b). Each row computes this addition step independently and outputs a carry or no-carry west face glue on the westernmost tile of each row (Figure 4c). As the addition tiles bind to the seed, tiles from the incrementation tile set (Figure 18c) may also begin to attach. The purpose of the incrementation tiles is to determine the sum for each A and B bit pair in the event of a no-carry from the row below and in the event of a carry from the row below (Figure 5). The final step of the addition mechanism presented here propagates carry or no-carry information northwards from the southernmost row of the assembly. When the carry propagation column reaches the top of the assembly, the most significant bit of the sum may be determined and the calculation is complete (Figure 6a-f).

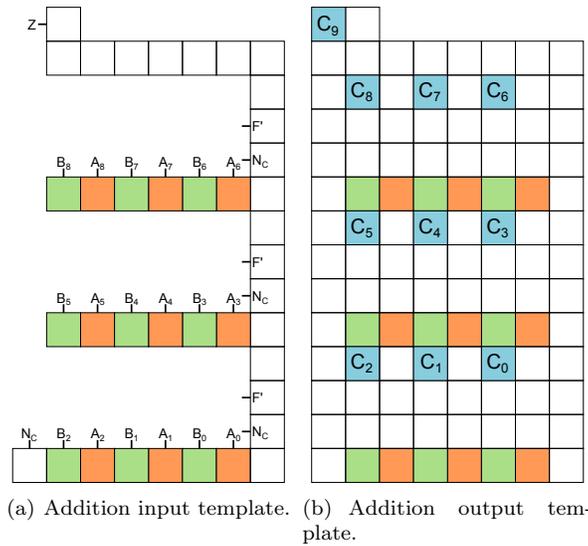(a) Addition input template. (b) Addition output template.

Figure 3: These are example I/O templates for the worst case $O(\sqrt{n})$ time addition introduced in Section 5.
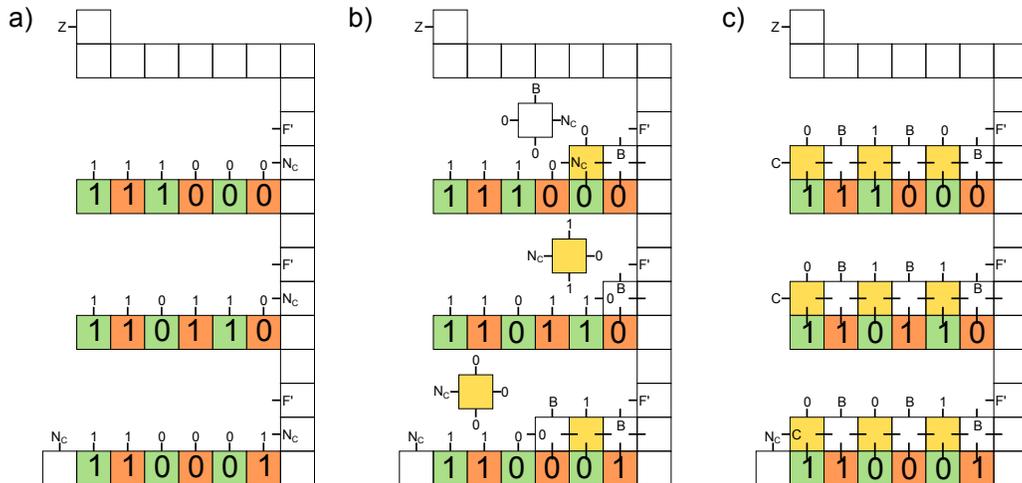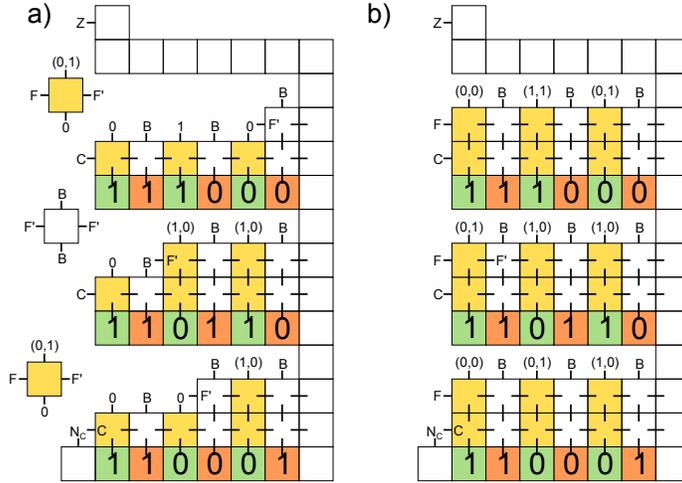

Figure 4: Step 1: Addition.
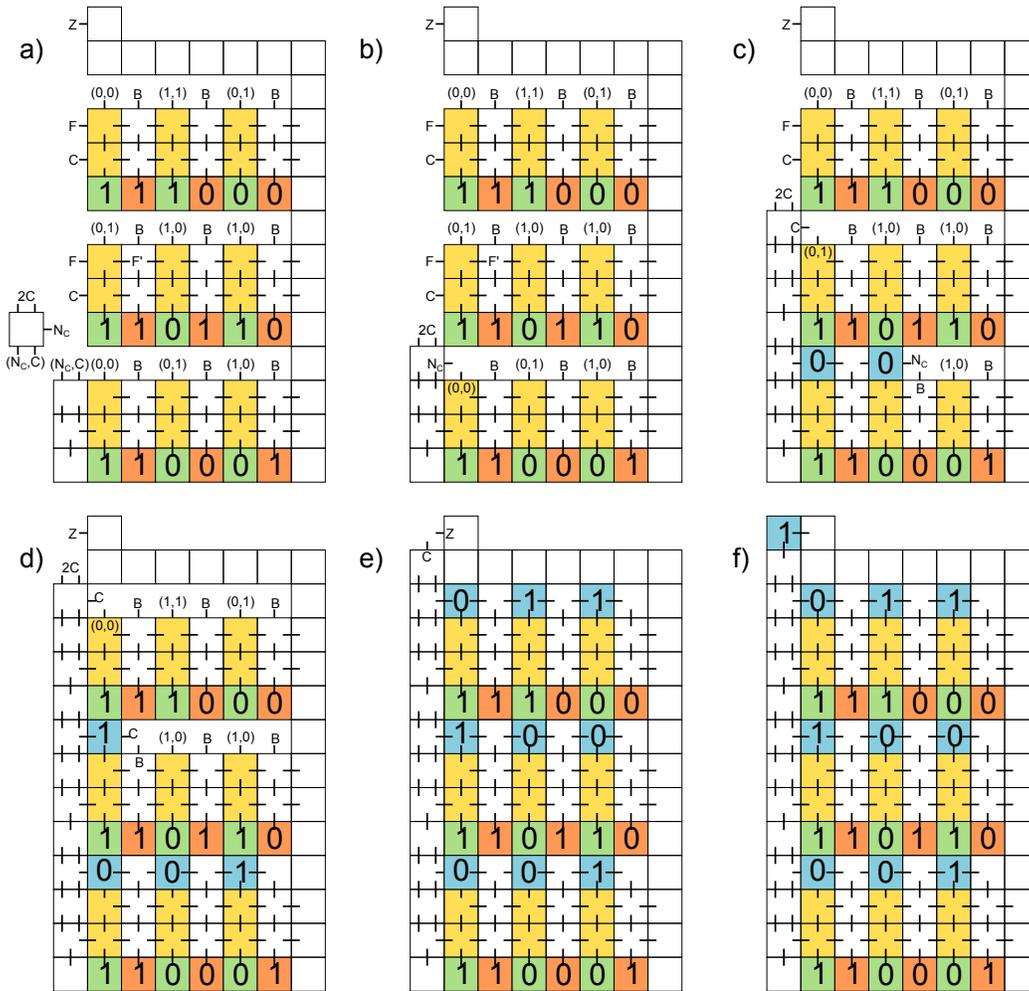
Figure 5: Step 2: Increment.

Figure 6: Step 3: Carry Propagation and Output.
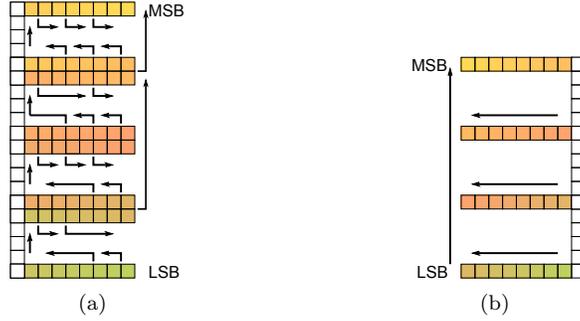
6 Towards Faster Addition

Figure 7: Arrows represent carry origination and direction of propagation for a) the $O(\log n)$ average case, the $O(\sqrt{n})$ worst case combined construction, and b) the $O(\sqrt{n})$ worst case construction.

In this section we combine the approaches described in Sections 4 and 5 in order to achieve both $O(\log n)$ average case addition and $O(\sqrt{n})$ worst case addition. This construction resembles the construction described in Section 5 in that the numbers to be added are divided into sections and values are computed for both possible carry-in bit values. Additionally, the construction described here lowers the average case run time by utilizing the carry-skip mechanism described in Section 4 within each section and between sections.

Theorem 6.1. There exists a 2-dimensional n-bit adder TAC with an average run-time of $O(\log n)$ and a worst case run-time of $O(\sqrt{n})$.

The proof follows from the construction of the adder in Appendices E.1, E.2, and E.3.

7 Addition in Three Dimensions

In this section we extend our adder construction into the third dimension to achieve a $O(\sqrt[3]{n})$ upper bound, which meets the $\Omega(\sqrt[3]{n})$ lower bound from Theorem B.4. Due to space and time constraints a general overview is given. We begin by creating $\sqrt[3]{n}$ total $\sqrt[3]{n} \times \sqrt[3]{n}$ constructions exactly as per Section 6 stacked one atop the other in an alternating fashion such that every odd plate, beginning with the first, has its MSB in the NW corner while the even plate has its LSB in that same corner. We continue by applying the same addition algorithm that was presented in Section 6 to all plates where every lower plate passes its carry-out to the upper plate. This is very similar to how carry out bits are passed between sections in the construction described in Section 6. Finally, every lower plate will pass the appropriate carry to the next lower plate. Please see Figure 8 for a visual overview of this process.

Theorem 7.1. There exists a 3-dimensional n-bit adder TAC with an average run-time of $O(\log n)$ and a worst case run-time of $O(\sqrt[3]{n})$.

We present a high-level sketch of the proof of Theorem 7.1 in Appendix 7.1

7.1 Proof Sketch of Theorem 7.1

We present a high-level sketch of the proof of Theorem 7.1 here. Similar with the $O(\log n)$ average case $O(\sqrt{n})$ worst case combined addition presented in Section 6, the binary numbers to be added are separated into sections each with length $O(\sqrt[3]{n})$. We arrange these numbers on $\sqrt[3]{n}$ scafolds of size $\sqrt[3]{n} \times \sqrt[3]{n}$ 2D. The dashed lines in Figure 8 are carries transmitted between each plate in the third dimension. The lines on the north are carries transmitted from each odd plate to its next even plate. The dashed lines on the southernmost part of Figure 8 are the carries transmitted from the first grouped plates to the last plate. Since the numbers
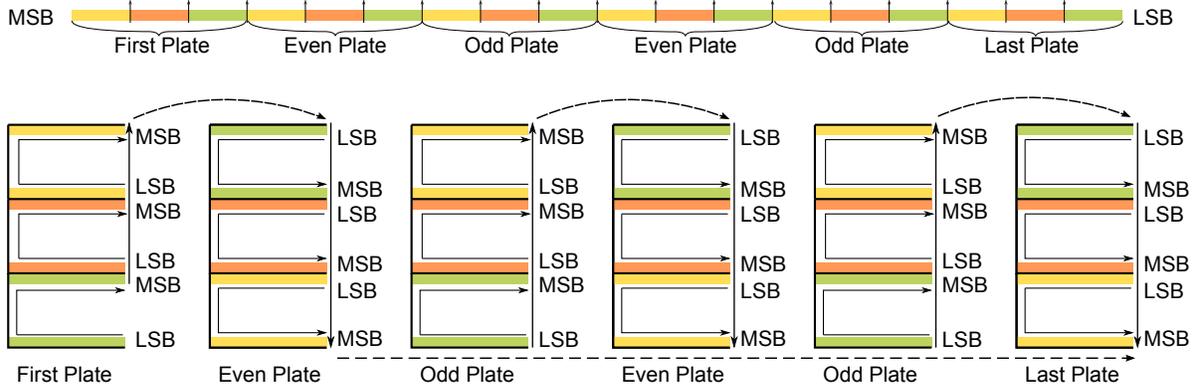
8

Figure 8: An overview of the process to add two number in optimal time in $3D$.

are all fit compactly with only a constant amount space between each plate and a constant amount of space between each section, any one side of the cube is at most $O(\sqrt[3]{n})$. Therefore, this size constraint along with the algorithms previously presented allow us to have an optimal $O(\sqrt[3]{n})$ time complexity with an average case complexity of $O(\log n)$.

8 Multiplication

In this section we sketch our $O(n^{\frac{5}{6}})$ time multiplication algorithm. Let $a = a_{n-1}a_{n-2}\ldots a_0$ and $b = b_{n-1}b_{n-2}\ldots b_0$ be two n-bit numbers to be multiplied. The key idea of our algorithm is to compute the product ab by computing the sum $ab = \sum_{i=0}^{n-1} a \cdot b_i \cdot 2^i$ by repeated application of our $O(\sqrt{n})$ time addition algorithm in a parallelized fashion. The statement of our result is as follows:

Theorem 8.1. There exists a 3D n-bit multiplier TAC with a worst case run-time of $O(n^{\frac{5}{6}})$.

The seed assembly for our construction consists of a 2D $O(\sqrt{n}) \times O(\sqrt{n})$ pad encoding inputs a and b in a "snaked" pattern as is done with the seed in Appendix E.1. Our algorithm expands this seed into a $O(n^{5/6}) \times O(n^{5/6}) \times O(n^{5/6})$ *mega-cube* consisting of n *number blocks* each of dimension $O(n^{1/2}) \times O(n^{1/2}) \times O(n^{1/2})$. Each number cube conceptually is associated with a distinct term from the numbers $a \cdot b_i \cdot 2^i$, and the mega-cube is organized into a $O(n^{1/3}) \times O(n^{1/3}) \times O(n^{1/3})$ array of number cubes (yielding the total dimension of $O(n^{5/6})$).

Assembly of the mega-cube proceeds by initializing the growth of $O(n^{1/3})$ rows of number cubes as shown in Figure 9 (b). The growth that initializes each row extends from the seed assembly at a $O(1)$ rate, thereby initializing all rows in time $O(n^{5/6})$. Each row proceeds by computing the appropriate value $a \cdot b_i \cdot 2^i$ for the given number cube and adding it via our 2D addition algorithm to a running total in $O(\sqrt{n})$ time per number cube. Note that computing a given value $a \cdot b_i \cdot 2^i$ is simply a bit shift rather than a full fledged multiplication and can therefore be derived quickly as shown in Appendix F. Therefore, each row finishes in $O(n^{1/2} \cdot n^{1/3}) = O(n^{5/6})$ time once initialized, yielding a $O(n^{5/6})$ finish time for the entire first level of the mega-cube shown in Figure 9 (b). Further, in addition to initializing the growth of the first plane of the mega-cube, a column is initialized to grow up into the third dimension to seed each of the $O(n^{1/3})$ levels of the mega-cube, implying that all levels finish in time $O(n^{5/6})$. Finally, each sum from the rows of a given level are computed into subtotals for each level as depicted in Figure 9 (c), followed by the summation of these subtotals into a final solution as shown in Figure 9 (d).

The details for our multiplication system are provided in Appendix F.1The full details of our construction are complex and our description in this abstract may be difficult to follow for a general audience. As ongoing work we are developing an expanded write-up and analysis that will be more approachable and we will present it in the final published version of this paper. Further, our run time bound of $O(n^{5/6})$ applies to
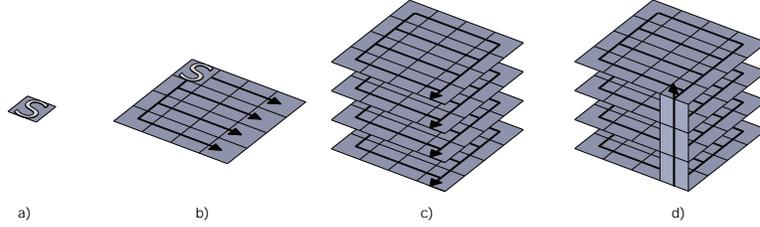
9

Figure 9: High level sketch of multiplication

the parallel time model. We therefore get a $O^*(n^{5/6})$ continuous time bound from Lemma A.1. We believe a tighter analysis should yield an equal asymptotic bound for both run time models.

9 Simulation

Tile self-assembly software simulations were conducted to visualize the diverse approaches to fast arithmetic presented in this paper, as well as to compare them to previous work. The adder tile constructions described in Sections 4-6, and the previous best [5] were simulated using the two timing models described in Sections A.

Figure 10 demonstrates the power of the approaches described in this paper compared to previous work.
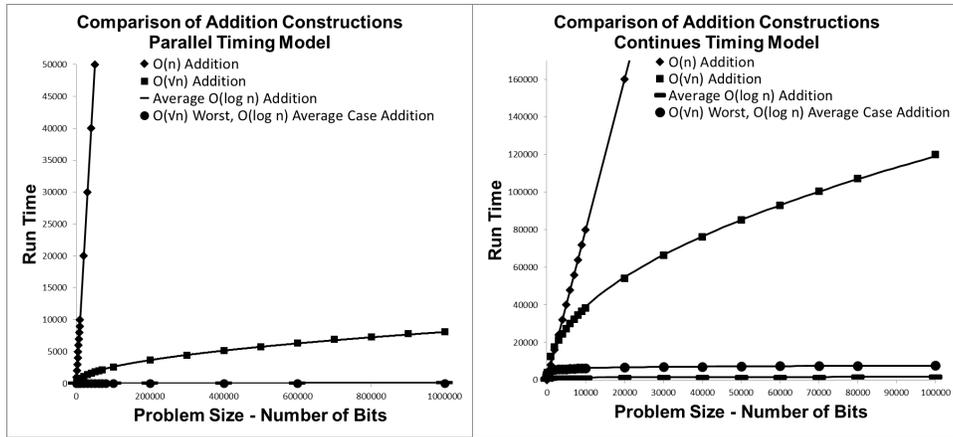

Figure 10: A comparison of adder constructions.

10 Future Work

The results of this paper are just a jumping off point and provide numerous directions for future work. One promising direction is further exploration of the complexity of multiplication. Can $O(n^{1/3})$ or $O(n^{1/2})$ be achieved in 3D? Is sublinear multiplication possible in 2D, or is there a $\Omega(n)$ lower bound? What is the average case complexity of multiplication?

Another direction for future work is the consideration of metrics other than run time. One potentially important metric is the geometric space taken up by the computation. Our intent is that fast function computing systems such as those presented in this paper will be used as building blocks for larger and more complex self-assembly algorithms. For such applications, the area and volume taken up by the computation is clearly an important constraint. Exploring tradeoffs between run time and imposed space limitations for

10

computation may be a promising direction with connections to resource bounded computation theory. Along these lines, another important direction is the general development of design methodologies for creating black box self-assembly algorithms that can be plugged into larger systems with little or no "tweaking".

A final direction focusses on the consideration of non-deterministic tile assembly systems to improve expected run times even for maniacally designed worst case input strings. Is it possible to achieve $O(\log n)$ expected run time for the addition problem regardless of the input bits? If not, are there other problems for which there is a provable gap in achievable assembly time between deterministic and non-deterministic systems?

Acknowledgements

References

[1] Zachary Abel, Nadia Benbernou, Mirela Damian, Erik Demaine, Martin Demaine, Robin Flatland, Scott Kominers, and Robert Schweller, *Shape replication through self-assembly and RNase enzymes*, SODA 2010: Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms (Austin, Texas), Society for Industrial and Applied Mathematics, 2010.

[2] Leonard Adleman, Qi Cheng, Ashish Goel, and Ming-Deh Huang, *Running time and program size for self-assembled squares*, Proceedings of the thirty-third annual ACM Symposium on Theory of Computing (New York, NY, USA), ACM, 2001, pp. 740–748.

[3] Leonard M. Adleman, Qi Cheng, Ashish Goel, Ming-Deh A. Huang, David Kempe, Pablo Moisset de Espanés, and Paul W. K. Rothemund, *Combinatorial optimization problems in self-assembly*, Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing, 2002, pp. 23–32.

[4] Florent Becker, Ivan Rapaport, and Eric Rémila, *Self-assembling classes of shapes with a minimum number of tiles, and in optimal time*, Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 2006, pp. 45–56.

[5] Yuriy Brun, *Arithmetic computation in the tile assembly model: Addition and multiplication*, Theoretical Computer Science **378** (2007), 17–31.

[6] Nathaniel Bryans, Ehsan Chiniforooshan, David Doty, Lila Kari, and Shinnosuke Seki, *The power of nondeterminism in self-assembly*, SODA 2011: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2011, pp. 590–602.

[7] Harish Chandran, Nikhil Gopalkrishnan, and John H. Reif, *The tile complexity of linear assemblies*, 36th International Colloquium on Automata, Languages and Programming, vol. 5555, 2009.

[8] Qi Cheng, Ashish Goel, and Pablo Moisset de Espanés, *Optimal self-assembly of counters at temperature two*, Proceedings of the First Conference on Foundations of Nanoscience: Self-assembled Architectures and Devices, 2004.

[9] Matthew Cook, Yunhui Fu, and Robert T. Schweller, *Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d*, Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011 (Dana Randall, ed.), SIAM, 2011, pp. 570–589.

[10] Erik Demaine, Matt Patitz, Trent Rogers, Robert Schweller, Scott Summers, and Damien Woods, *The two-handed tile assembly model is not intrinsically universal*, Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP 2013), 2013.

[11] David Doty, *Randomized self-assembly for exact shapes*, SIAM Journal on Computing **39** (2010), no. 8, 3521–3552.

[12] David Doty, Jack H. Lutz, Matthew J. Patitz, Robert Schweller, Scott M. Summers, and Damien Woods, *The tile assembly model is intrinsically universal*, FOCS 2012: Proceedings of the 53rd IEEE Conference on Foundations of Computer Science, 2012.

[13] David Doty, Matthew J. Patitz, Dustin Reishus, Robert T. Schweller, and Scott M. Summers, *Strong fault-tolerance for self-assembly with fuzzy temperature*, Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010), 2010, pp. 417–426.

[14] Bin Fu, Matthew J. Patitz, Robert Schweller, and Robert Sheline, *Self-assembly with geometric tiles*, ICALP 2012: Proceedings of the 39th International Colloquium on Automata, Languages and Programming (Warwick, UK), 2012.

[15] Ming-Yang Kao and Robert T. Schweller, *Randomized self-assembly for approximate shapes*, International Colloqium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 5125, Springer, 2008, pp. 370–384.

[16] Mark Schilling, *The longest run of heads*, The College Mathematics Journal **21** (1990), no. 3, 196–207.

[17] Robert Schweller and Michael Sherman, *Fuel efficient computation in passive self-assembly*, Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, 2013.

[18] Erik Winfree, *Algorithmic self-assembly of DNA*, Ph.D. thesis, California Institute of Technology, June 1998.

[19] Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin, *Efficient active self-assembly of shapes*, Manuscript (2012).

A   Run Time Model Definitions

Parallel Run-time.   For a deterministic tile system $\Gamma = (T, S, \tau)$ and assembly $A \in \mathtt{PROD}_\Gamma$, the 1-*step transition* set of assemblies for A is defined to be $STEP_{\Gamma,A} = \{B \in \mathtt{PROD}_\Gamma | A \to_{T,\tau} B\}$. For a given $A \in \mathtt{PROD}_\Gamma$, let $\mathtt{PARALLEL}_{\Gamma,A} = \bigcup_{B \in STEP_{\Gamma,A}} B$, ie, $\mathtt{PARALLEL}_{\Gamma,A}$ is the result of attaching all singleton tiles that can attach directly to A. Note that since Γ is deterministic, $\mathtt{PARALLEL}_{\Gamma,A}$ is guaranteed to not contain overlapping tiles and is therefore an assembly. For an assembly A, we say $A \Rightarrow_\Gamma A'$ if $A' = \mathtt{PARALLEL}_{\Gamma,A}$. We define the *parallel run-time* of a deterministic tile system $\Gamma = (T, S, \tau)$ to be the non-negative integer k such that $A_1 \Rightarrow_\Gamma A_2 \Rightarrow_\Gamma \ldots \Rightarrow_\Gamma A_k$ where $A_1 = S$ and $\{A_k\} = \mathtt{TERM}_\Gamma$. As notation we denote this value for tile system Γ as ρ_Γ. For any assemblies A and B in $\mathtt{PROD}_\Gamma$ such that $A_1 \Rightarrow_\Gamma A_2 \Rightarrow_\Gamma \ldots \Rightarrow_\Gamma A_k$ with $A = A_1$ and $B = A_k$, we say that $A \Rightarrow_\Gamma^k B$. Alternately, we denote B with notation $A \Rightarrow_\Gamma^k$. For a TAC $\Im = (T, U, V, \tau)$ that computes function f, the run time of $\Im$ on input b is defined to be the parallel run-time of tile system $\Gamma_{\Im,b} = (T, U_b, \tau)$. Worst case and average case run time are then defined in terms of the largest run time inducing b and the average run time for a uniformly generated random b.

Continuous Run-time.   In the *continuous* run time model the assembly process is modeled as a continuous time Markov chain and the time spent in each assembly is a random variable with an exponential distribution. The states of the Markov chain are the elements of $PROD_\Gamma$ and the transition rate from A to A' is $\frac{1}{|T|}$ if $A \to A'$ and 0 otherwise. For a given tile assembly system Γ, we use notation Γ_{MC} to denote the corresponding continuous time Markov chain. Given a deterministic tile system Γ with unique assembly U, the run time of Γ is defined to be the expected time of Γ_{MC} to transition from the seed assembly state to the unique sink state U. As notation we denote this value for tile system Γ as ς_Γ. One immediate useful fact that follows from this modeling is that for a producible assembly A of a given deterministic tile system, if $A \to A'$, then the time taken for Γ_{MC} to transition from A to the first A'' that is a superset of A' (through 1 or more tile attachments) is an exponential random variable with rate $1/|T|$. For a more general modeling in which different tile types are assigned different rate parameters see [2, 3, 8]. Our use of rate $1/|T|$ for all tile types corresponds to the assumption that all tile types occur at equal concentration and the expected time for 1 tile (of any type) to hit a given attachable position is normalized to 1 time unit. Note that as our tile sets are constant in size in this paper, the distinction between equal or non-equal tile type concentrations does not affect asymptotic run time. For a TAC $\Im = (T, U, V, \tau)$ that computes function f, the run time of $\Im$ on input b is defined to be the continuous run-time of tile system $\Gamma_{\Im,b} = (T, U_b, \tau)$. Worst case and average case run time are then defined in terms of the largest run time inducing b and the average run time for a uniformly generated random b.

Relating Parallel Time and Continuous Time.   The following Lemma states an upper and lower bound on continuous time with respect to parallel time. Both bounds are straightforward to derive with the lower bound appearing in [4] and the upper bound being derivable in a fashion similar to the proof of Theorem 5.2 in [2]. The most dramatic distinction between parallel and continuous time occurs when the number of tile types, $|T|$, is large, as this slows down assembly in the continuous model but does not affect parallel time. When $|T| = O(1)$, which we adhere to in this paper, the lemma implies that the timing models are very close.

Lemma A.1. Consider a deterministic aTAM system $\Gamma = (T, S, \tau)$ that uniquely assembles (finite) assembly A. Let ρ_Γ denote the parallel time for Γ to assemble A and let ς_Γ denote the continuous time for Γ to assemble A. Then,

- $\varsigma_\Gamma \geq |T| \rho_\Gamma$

- $\varsigma_\Gamma = O(|T| \rho_\Gamma \log(\rho_\Gamma + |S|))$

It is not clear if the $\log(\rho_\Gamma)$ factor in the upper bound for continuous time is necessary, and in fact for the algorithms in this paper the parallel and continuous run times are equal up to constant factors. However, the $\log(|S|)$ factor for general systems is important. Consider a $1 \times |S|$ line of tiles for a seed assembly to

which $|S|$ copies of a single tile type may attach to form a terminal $2 \times |S|$ assembly. This system has a parallel run time of just 1, but $\Theta(\log |S|)$ continuous run time, implying that the $\log |S|$ factor in our bound is tight in the general case. In [4] a comparison between these two run time models is considered in the extension in which tile types may have different concentrations.

B   Lower Bounds

B.1   Communication Problem.

The Δ-communication problem is the problem of computing the function $f(b_1, b_2) = b_1 \wedge b_2$ for bits b_1 and b_2 in the 3D aTAM under the additional constraint that the input template for the solution $U = (R, B(i))$ be such that $\Delta = \max(|B(1)_x - B(2)_x|, |B(1)_y - B(2)_y|, |B(1)_z - B(2)_z|)$.

We first establish a lemma which intuitively states that for any 2 seed assemblies that differ in only a single tile position, all points of distance greater than r from the point of difference will be identically tiled (or empty) after r time steps of parallelized tile attachments:

Lemma B.1. Let $S_{p,t}$ and $S_{p,t'}$ denote two assemblies that are identical except for a single tile t versus t' at position $p = (p_x, p_y, p_z)$ in each assembly. Further, let $\Gamma = (T, S_{p,t}, \tau)$ and $\Gamma' = (T, S_{p,t'}, \tau)$ be two deterministic tile assembly systems such that $S_{p,t} \Rightarrow^r_\Gamma R$ and $S_{p,t'} \Rightarrow^r_{\Gamma'} R'$ for non-negative integer r. Then for any point $q = (q_x, q_y, q_z)$ such that $r < \max(|p_x - q_x|, |p_y - q_y|, |p_z - q_z|)$, it must be that $R_q = R'_q$, ie, R and R' contain the same tile at point q.

The proof for Lemma B.1 can be found in Appendix B.3.

Theorem B.2. Any solution to the Δ-communication problem has run time at least $\frac{1}{2}\Delta$.

The proof for Theorem B.2 may be found in Appendix B.4.

B.2 $\Omega(\sqrt[d]{n})$ Lower Bounds for Addition and Multiplication.

We now show how to reduce instances of the communication problem to the arithmetic problems of addition and multiplication in 2D and 3D to obtain lower bounds of $\Omega(\sqrt{n})$ and $\Omega(\sqrt[3]{n})$ respectively.

We first show the following Lemma which lower bounds the distance of the farthest pair of points in a set of n points. We believe this Lemma is likely well known, or is at least the corollary of an established result. We include it's proof for completeness.

Lemma B.3. For positive integers n and d, consider $A \subset \mathbb{Z}^d$ and $B \subset \mathbb{Z}^d$ such that $A \bigcap B = \emptyset$ and $|A| = |B| = n$. There must exist points $p \in A$ and $q \in B$ such that $\Delta_{p,q} \geq \lceil \frac{1}{2}\lceil \sqrt[d]{2n}\rceil\rceil - 1$.

The proof of Lemma B.3 can be found in Appendix B.5.

Theorem B.4. Any n-bit adder TAC that has a dimension d input template for $d = 1$, $d = 2$, or $d = 3$, has a worst case run time of $\Omega(\sqrt[d]{n})$.

The proof of Theorem B.4 can be found in Appendix B.6.

As the bound of dimension d on the input template of a TAC alone lower bounds the run time of the TAC, we get the following corollary.

We now provide a lower bound for multiplication.

Theorem B.5. Any n-bit multiplier TAC that has a dimension d input template for $d = 1$, $d = 2$, or $d = 3$, has a worst case run time of $\Omega(\sqrt[d]{n})$.

The proof of Theorem B.5 can be found in Appendix B.7.

As with addition, the lower bound implied by the limited dimension of the input template alone yields the general lower bound for d dimensional multiplication TACS.

B.3 Proof of Lemma B.1

We show this by induction on r. As a base case of $r = 0$, we have that $R = S_{p,t}$ and $R' = S_{q,t'}$, and therefore R and R' are identical at any point outside of point p by the definition of $S_{p,t}$ and $S_{p,t'}$.

Inductively, assume that for some integer k we have that for all points w such that $k < \Delta_{p,w} \triangleq \max(|p_x - w_x|, |p_y - w_y|, |p_z - w_z|)$, we have that $R_w^k = R_w^{'k}$, where $S_{p,t} \rightrightarrows_\Gamma^k R^k$, and $S_{p,t'} \rightrightarrows_{\Gamma'}^k R^{'k}$. Now consider some point q such that $k + 1 < \Delta_{p,q} \triangleq \max |p_x - q_x|, |p_y - q_y|, |p_z - q_z|$, along with assemblies R^{k+1} and $R^{'k+1}$ where $S_{p,t} \rightrightarrows^{k+1} R^{k+1}$, and $S_{p,t'} \rightrightarrows^{k+1} R^{'k+1}$. Consider the direct neighbors (6 of them in 3D) of point q. For each neighbor point c, we know that $\Delta_{p,c} > k$. Therefore, by inductive hypothesis, $R_c^k = R_c^{'k}$ where $S_{p,t} \rightrightarrows_\Gamma^k R^k$, and $S_{p,t'} \rightrightarrows_{\Gamma'}^k R^{'k}$. Therefore, as attachment of a tile at a position is only dependent on the tiles in neighboring positions of the point, we know that tile R_q^{k+1} may attach to both R^k and $R^{'k}$ at position q, implying that $R_q^{k+1} = R_q^{'k+1}$ as Γ and Γ' are deterministic.

B.4 Proof of Theorem B.2

Consider a TAC $\Im = (T, U = (R, B(i)), V(i), \tau)$ that computes the Δ-communication problem. First, note that B has domain of 1 and 2, and V has domain of just 1 (the input is 2 bits, the output is 1 bit). We now consider the value Δ_V defined to be the largest distance between the output bit position of V from either of the two input bit positions in B: Let $\Delta_V \triangleq \max(\Delta_{B(1),V(1)}, \Delta_{B(2),V(1)})$. Without loss of generality, assume $\Delta_V = \Delta_{B(1),V(1)}$. Note that $\Delta_V \geq \frac{1}{2}\Delta$.

Now consider the computation of $f(0,1) = 0$ versus the computation of $f(1,1) = 1$ via our TAC $\Im$. Let A^0 denote the terminal assembly of system $\Gamma_0 = (T, U_{0,1}, \tau)$ and let A^1 denote the terminal assembly of system $\Gamma_1 = (T, U_{1,1}, \tau)$. As $\Im$ computes f, we know that $A_{V(1)}^0 \neq A_{V(1)}^1$. Further, from Lemma B.1, we know that for any $r < \Delta_V$, we have that $W_{V(1)}^0 = W_{V(1)}^1$ for any W^0 and W^1 such that $U_{0,1} \rightrightarrows_{\Gamma_0}^r W^0$ and $U_{1,1} \rightrightarrows_{\Gamma_1}^r W^1$. Let $d_\Im$ denote the run time of $\Im$. Then we know that $U_{0,1} \rightrightarrows_{\Gamma_0}^{d_\Im} A^0$, and $U_{1,1} \rightrightarrows_{\Gamma_1}^{d_\Im} A^1$ by the definition of run time. If $d_\Im < \Delta_V$, then Lemma B.1 implies that that $A_{V(1)}^0 = A_{V(1)}^1$, which contradicts the fact that $\Im$ compute f. Therefore, the run time $d_\Im$ is at least $\Delta_V \geq \frac{1}{2}\Delta$.

B.5 Proof of Lemma B.3

To see this, consider a bounding box of all $2n$ points. If all d dimensions of the bounding box were of length strictly less that $\sqrt[d]{2n}$, then the box could not contain all $2n$ points. Therefore, at least one dimension is of length at least $\lceil \sqrt[d]{2n} \rceil$, implying that there are two points of distance at least $\lceil \sqrt[d]{2n} \rceil - 1$ along that particular axis. If these two points are in A and B respectively, then the claim follows. If not, say both are from set A, then there must be a point in B that is at least $\lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1$ from one of these two points in A, implying the claim.

B.6 Proof of Theorem B.4

To show the lower bound, we will reduce the Δ-communication problem for some $\Delta = \Omega(\sqrt[d]{n})$ to the n-bit adder problem with a d-dimension template. Consider some n-bit adder TAC $\Im = (T, U = (F, W), V, \tau)$ such that U is a d-dimension template. The $2n$ sequence of wildcard positions W of this TAC must be contained in d-dimensional space by the definition of a d-dimension template, and therefore by Lemma B.3 there must exist points $W(i)$ for $i \leq n$, and $W(n+j)$ for $j \leq n$, such that $\Delta_{W(i),W(n+j)} \geq \lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1 = \Omega(\sqrt[d]{n})$. Now consider two n-bit inputs $a = a_n \ldots a_1$ and $b = b_n \ldots b_1$ to the adder TAC $\Im$ such that: $a_k = 0$ for any $k > i$ and any $k < j$, and $a_k = 1$ for any k such that $j \leq k < i$. Further, let $b_k = 0$ for all $k \neq j$. The remaining bits a_i and a_j are unassigned variables of value either 0 or 1. Note that the $i + 1$ bit of $a + b$ is 1 if and only if a_i and b_j are both value 1. This setup constitutes our reduction of the Δ-communication problem to the addition problem as the adder TAC template with the specified bits hardcoded in constitutes a template for the Δ-communication problem that produces the AND of the input bit pair. We now specify explicitly how to generate a communication TAC from a given adder TAC.

For given n-bit adder TAC $\Im = (T, U = (F, W), V, \tau)$ with dimension d input template, we derive a Δ-communication TAC $\rho = (T, U^2 = (F^2, W^2), V^2, \tau)$ as follows. First, let $W^2(1) = W(i)$, and $W^2(2) = W(n+j)$. Note that as $\Delta_{W(i), W(n+j)} = \Omega(\sqrt[d]{n})$, W^2 satisfies the requirements for a Δ-communication input template for some $\Delta = \Omega(\sqrt[d]{n})$. Derive the frame of the template F^2 from F by adding tiles to F as follows: For any positive integer $k > i$, or $k < j$, or $k > n$ but not $k = n+j$, add a translation of t_0 (with label "0") translated to position $W(k)$. Additionally, for any k such that $j \leq k < i$, add a translation of t_1 (with label "1") at translation $W(k)$.

Now consider the Δ-communication TAC $\rho = (T, U^2 = (F^2, W^2), V^2, \tau)$ for some $\Delta = \Omega(\sqrt[d]{n})$. As assembly $U^2_{a_i, b_j} = U_{a_1 \ldots a_n, b_1 \ldots b_n}$, we know that the worst case run time of ρ is at most that of the worst case run time of $\Im$. Therefore, by Theorem B.2, we have that $\Im$ has a run time of at least $\Omega(\sqrt[d]{n})$.

B.7 Proof of Theorem B.5

Consider some n-bit multiplier TAC $\Im = (T, U = (F, W), V, \tau)$ with d-dimension input template. By Lemma B.3, some $W(i)$ and $W(n+j)$ must have distance at least $\Delta \geq \lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1$. Now consider input strings $a = a_n \ldots a_1$ and $b = b_n \ldots b_1$ to $\Im$ such that a_i and b_j are of variable value, and all other a_k and b_k have value 0. For such input strings, the $i + j$ bit of the product ab has value 1 if and only if $a_i = b_j = 1$. Thus, we can convert the n-bit multiplier system $\Im$ into a Δ-communication TAC with the same worst case run time in the same fashion as for Theorem B.4, yielding a $\Omega(\sqrt[d]{n})$ lower bound for the worst case run time of $\Im$.

C Average Case $O(\log n)$ Time Addition.
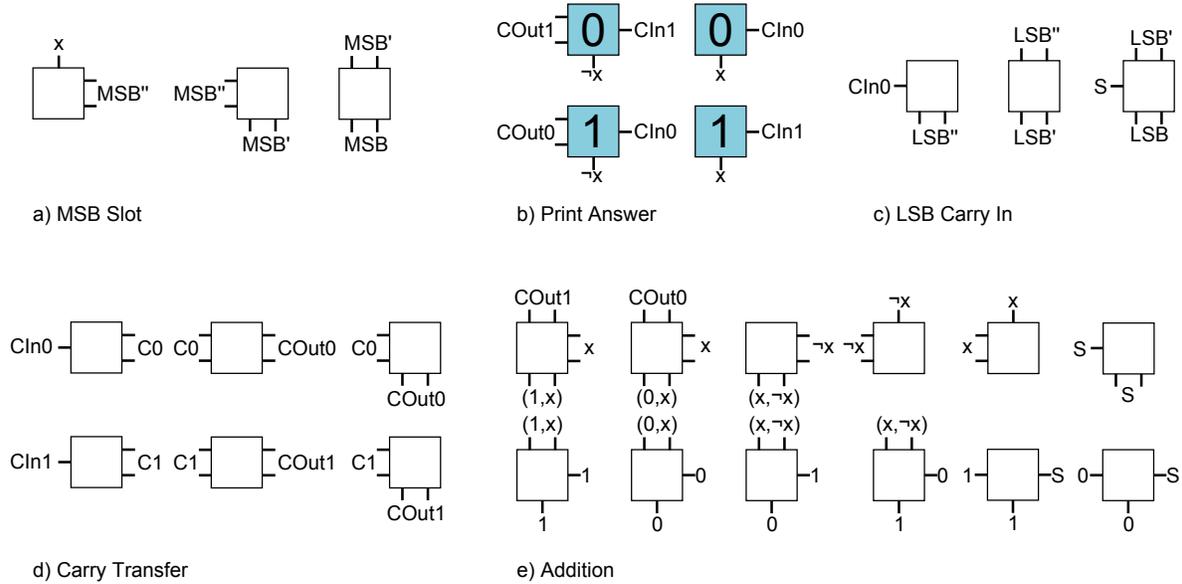
C.1 Construction

Figure 11: This is the complete set of tiles necessary to implement average case $O(\log n)$ and worst case $O(n)$ addition.

We summarize the mechanism of addition presented here in a short example. The complete tile set may be found in Figure 11.

Input Template. The input template, or seed, for the construction of an adder with an $O(\log n)$ average case is shown in Figure 12. This input template is composed of n blocks, each containing three tiles. Within a block, the easternmost tile is the S labeled tile followed by two tiles representing A_k and B_k, the kth bits of A and B respectively. Of these n blocks, the easternmost and westernmost blocks of the template assembly are unique. Instead of an S tile, the block furthest east has an LSB-labeled tile which accompanies the tiles representing the least significant bits of A and B, A_0 and B_0. The westernmost block of the template assembly contains a block labeled MSB instead of the S block and accompanies the most significant bits of A and B, A_{n-1} and B_{n-1}.
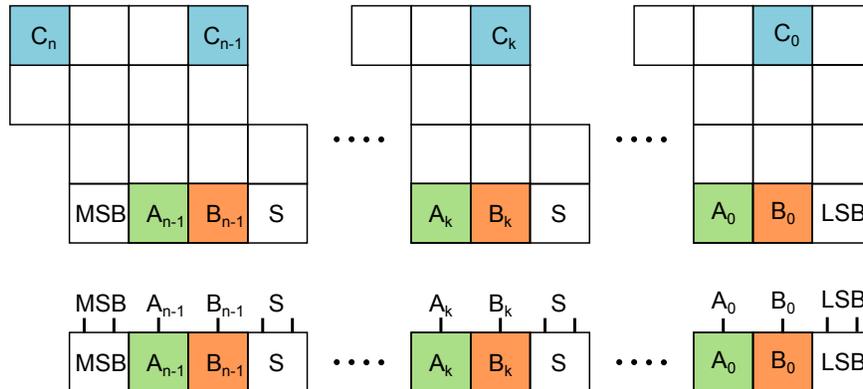

Figure 12: Top: Output template displaying addition result C for $O(\log n)$ average case addition construction. Bottom: Input template composed of n blocks of three tiles each, representing n-bit addends A and B.

Computing Carry Out Bits. For clarity, we demonstrate the mechanism of this adder TAC through an example by selecting two 4-bit binary numbers A and B such that the addends A_i and B_i encompass every possible addend combination. The input template for such an addition is shown in Figure 13a where orange tiles represent bits of A and green tiles represent bits of B. Each block begins the computation in parallel at each S tile. After six parallel steps (Figure 13b-g), all carry out bits, represented by glues $C0$ and $C1$, are determined for addend-pairs where both A_i and B_i are either both 0 or both 1. For addend pairs A_i and B_i where one addend is 0 and one addend is 1, the carry out bit cannot be deduced until a carry out bit has been produced by the previous addend pair, A_{i-1} and B_{i-1}. By step seven, a carry bit has been presented to all addend pairs that are flanked on the east by an addend pair comprised of either both 0s or both 1s, or that are flanked on the east by the LSB start tile, since the carry in to this site is always 0 (Figure 13h). For those addend pairs flanked on the east by a contiguous sequence of size j pairs consisting of one 1 and one 0, $2j$ parallel attachment steps must occur before a carry bit is presented to the pair.

Computing the Sum. Once a carry out bit has been computed and carried into an addend pair A_i and B_i, two parallel tile addition steps are required to compute the sum of the addend pair (Figure 13g-j) .

C.2   Time Complexity.

C.2.1   Parallel Time

$O(n)$ **- worst case.** We first show that this construction has an $O(n)$ worst case run-time under the timing model presented in Section A *Run-time*. Consider a binary sequence T of length $2n$ representing two n-bit binary numbers A and B. A_0 and B_0 represent the least significant bits of A and B, respectively, and A_n and B_n represent the most significant bits of A and B, respectively. The formatting of T is such that $T_i = B_{i/2}$ if i is even, and $T_i = A_{(i-1)/2}$ if i is odd. Sequence T is shown in Figure 15a.
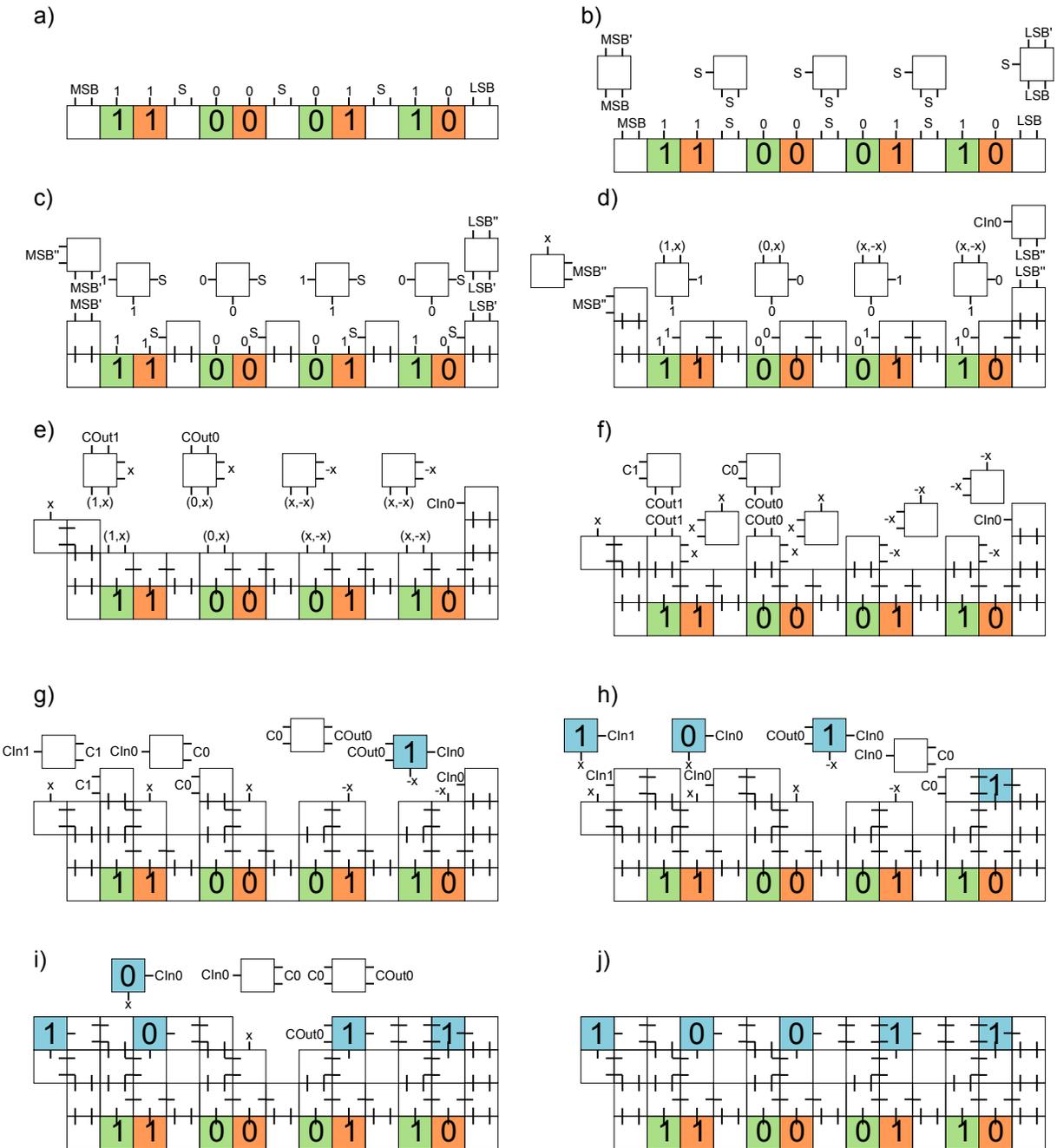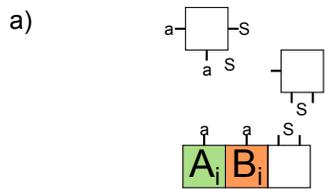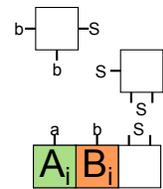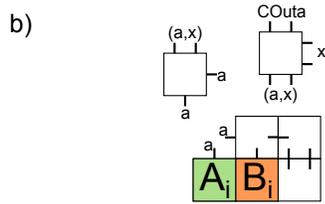
18

Figure 13: Example tile assembly system to compute the sum of 1001 and 1010 using the adder TAC presented in Section 4. a) The template with the two 4-bit binary numbers to be summed. b) The first step is the parallel binding of tiles to the S, LSB, and MSB tiles. c)Tiles bind cooperatively to west-face S glues and glues representing bits of B. The purpose of this step is to propagate bit B_i closer to A_i so that in d) a tile may bind cooperatively, processing information from both A_i and B_i. e) Note that addend-pairs consisting of either both 1s or both 0s have a tile with a north face glue consisting of either $(1,x)$ or $(0,x)$ bound to the A_i tile. This glue represents a carry out of either 1 or 0 from the addend-pair. In (e-g) the carry outs are propagated westward via tile additions for those addend-pairs where the carry out can be determined. Otherwise, spacer tiles bind. h) Tiles representing bits of C (the sum of A and B) begin to bind where a carry in is known. i-j) As carry bits propagate through sequences of $(0,1)$ or $(1,0)$ addend pairs, the final sum is computed.
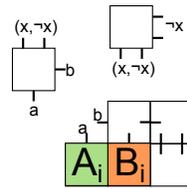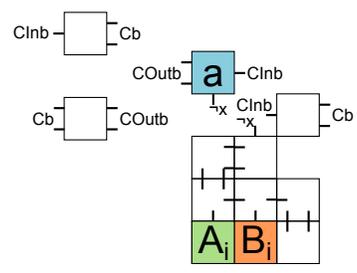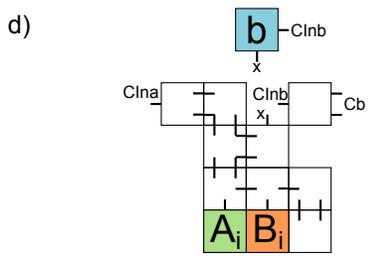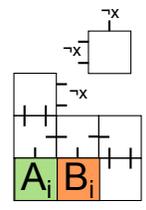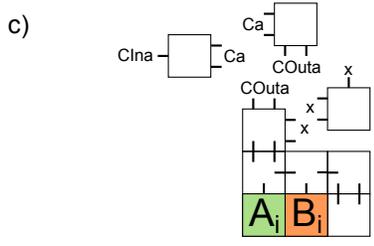
19

a)

A_i and B_i are both 0s or both 1s.

A_i and B_i are different; a=0, b=1 or a=1, b=0.

b)

The (a,x) means that the value of the carry generated for next block is *a* and the result in this block will be the value of the carry-in, x, from previous block.

The (x,¬x) means that the value of the carry which will be propagated to the next block is x, which is same as the carry-in from the previous block. Also, the result in this block will be NOT x.

c)

d)

Here we see that if the input values are equal, the carry that was just generated will be sent to the next block immediately. However, if the input values are not equal, the carry for next block will be propagated after the carry-in is accepted.

e)

$A_i=B_i=a$; carry=b; $A_i+B_i+carry=ab$

$A_i=¬B_i$; a=¬b; carry=b; $Ai+B_i+carry=ba$

Figure 14: The set of figures on the left side show how a carry out may be generated before the *addend-pair* has recieved a carry in. The set of figures on the right side show how a carry out can be dependent upon a carry in having been recieved by the *addend-pair*.

Figure 15: a) Binary sequence T populated by n-bit binary numbers A and B. b) Four possible (A_i, B_i) addend-pair combinations.

T contains n addend-pairs, (A_i, B_i), which are ordered pairs consisting of the ith bit of A and the ith bit of B. The four possible values for each addend-pair are shown in Figure 15b, along with the carry bits they produce upon addition. In T, there exist sequences of various sizes up to k addend-pairs such that every addend-pair in the interval fr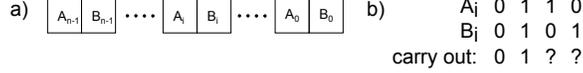om (A_i, B_i) to (A_{i+j-1}, B_{i+j-1}) of a size j addend-pair sequence is $(1,0)$ or $(0,1)$. In the adder TAC outlined in Section 4 and Appendix C.1, the value of the carry bit produced upon addition of $A_i + B_i$ for every $(0,0)$ and $(1,1)$ addend-pair is known and made available to the next addend-pair (A_{i+1}, B_{i+1}) after seven parallel tile addition steps, including the carry bit into (A_0, B_0). Therefore, after a constant number of parallel tile addition steps, the first addend-pair (A_i, B_i) in any j addend-pair-length sequence of $(1,0)$ or $(0,1)$ addend-pairs will be presented with the carry bit from the previous addend-pair (A_{i-1}, B_{i-1}). After $2j$ subsequent parallel tile addition steps, the carry bit from the last addend-pair (A_{i+j-1}, B_{i+j-1}) of the j-size sequence of consecutive $(0,1)$ and $(1,0)$ addend-pairs is presented to (A_{i+j}, B_{i+j}). Once an addend-pair has recieved a carry-in bit, the final sum is computed in one tile addition step. If k is the longest contiguous sequence of $(0,1)$ and $(1,0)$ addend-pairs, then $2k+8$ parallel tile addition steps are required to compute the sum C of A and B (Figure 16). Therefore, the time complexity is $O(k)$. Since the growth is bounded upwards by the longest contiguous sequence k of $(0,1)$ and

Figure 16:

$(1,0)$ addend-pairs, then the worst-case scenario occurs when $k = n$. Thus, the worst-case run time is $O(n)$.

$O(\log n)$ **- average case.** We now show that the average case run-time is $O(\log n)$ under the timing model presented in Section A. In two n-bit randomly generated binary numbers, A and B, the probability of one of the $(1,0)$ $(0,1)$ addend-pair cases occurring at (A_i, B_i) is $1/2$. The sequence of (A_i, B_i) bit pairs can thus be thought of as a Bernoulli process in which the likelihood of occurrence of a $(0,1)$ or $(1,0)$ addend-pair is equal to the occurrence of a $(1,1)$ or $(0,0)$ addend-pair. As shown above, the runtime is bounded by k, the longest contiguous sequence of $(0,1)$ or $(1,0)$ addend-pairs, which might be thought of as the longest sequence of heads in n independent fair coin tosses. Using Lemma 4.1, the expected longest run of heads in n coin tosses is $O(\log n)$ [16]. Therefore, the average case time complexity of the tile addition algorithm described above is $O(\log n)$.

C.2.2 Continuous Time

To analyze the continuous run time of our TAC we first introduce a Lemma C.2. This Lemma is essentially Lemma 5.2 of [19] but slightly generalized and modified to be a statement about exponentially distributed random variables rather than a statement specific to the self-assembly model considered in that paper. To derive this Lemma, we make use of a well known Chernoff bound for the sum of exponentially distributed random variables stated in Lemma C.1.

Lemma C.1. Let $X = \sum_{i=1}^{n} X_i$ be a random variable denoting the sum of n independent exponentially distributed random variables each with respective rate λ_i. Let $\lambda = \min\{\lambda_i | 1 \leq i \leq n\}$. Then $Pr[X > (1+\delta)n/\lambda] < (\frac{\delta+1}{e^\delta})^n$.

Lemma C.2. Let $t_1, \ldots, t_m$ denote m random variables where each $t_i = \sum_{j=1}^{k_i} t_{i,j}$ for some positive integer k_i, and the variables $t_{i,j}$ are independent exponentially distributed random variable each with respective rate $\lambda_{i,j}$. Let $k = \max(k_1, \ldots k_m)$, $\lambda = \min\{\lambda_{i,j} | 1 \leq i \leq m, 1 \leq j \leq k_i\}$, $\lambda' = \max\{\lambda_{i,j} | 1 \leq i \leq m, 1 \leq j \leq k_i\}$, and $T = \max(t_1, \ldots t_m)$. Then $E[T] = O(\frac{k+\log m}{\lambda})$ and $E[T] = \Omega(\frac{k+\log m}{\lambda'})$.

Proof. First we show that $E[T] = O(\frac{k+\log m}{\lambda})$. For each t_i we know that $E[t_i] = \sum_{j=1}^{k_i} \lambda_{i,j} \leq k/\lambda$. Applying Lemma C.1 we get that:

$$Pr[t_i > k/\lambda(1+\delta)] < (\frac{\delta+1}{e^\delta})^k.$$

Applying the union bound we get the following for all $\delta \geq 3$:

$$Pr[T > \frac{(1+\delta)k}{\lambda}] \leq m(\frac{\delta+1}{e^\delta})^k \leq me^{-k\delta/2}, \text{ for all } \delta > 3.$$

Let $\delta_m = \frac{2\ln m}{k}$. By plugging in $\delta_m + x$ for δ in the above inequality we get that:

$$Pr[T > \frac{(1+\delta_m+x)k}{\lambda}] \leq e^{-kx/2}.$$

Therefore we know that:

$$E[T] \leq \frac{k+2\ln m}{\lambda} + \int_{x=0}^{\infty} e^{-kx/2} = O(\frac{k+\log m}{\lambda}).$$

To show that $E[T] = \Omega(\frac{k+\log m}{\lambda'})$, first note that $T \geq \max_{1 \leq i \leq m}\{t_{i,1}\}$ and that $E[\max_{1 \leq i \leq m}\{t_{i,1}\}] = \Omega(\frac{\log m}{\lambda'})$, implying that $E[T] = \Omega(\frac{\log m}{\lambda'})$. Next, observe that $T \geq t_i$ for each i. Since $E[t_i]$ is at least k/λ' for at least one choice of i, we therefore get that $E[T] = \Omega(k/\lambda')$.

$\square$

The next Lemma helps us upper bound continuous run times by stating that if the assembly process is broken up into separate phases, denoted by a sequence of subassembly *waypoints* that must be reached, we may simply analyze the expected time to complete each phase and sum the total for an upper bound on the total run time. The following Lemma follows easily from the fact that a given open tile position of an assembly is filled at a rate at least as high as any smaller subassembly.

Lemma C.3. Let $\Gamma = (T, S, \tau)$ be a deterministic tile assembly system and $A_0 \ldots A_r$ be elements of $PROD_\Gamma$ such that $A_0 = S$, A_r is the unique terminal assembly of Γ, and $A_0 \subseteq A_2 \subseteq \ldots A_r$. Let t_i be a random variable denoting the time for Γ_{MC} to transition from A_{i-1} to the first A_i' such that $A_i \subseteq A_i'$ for i from 1 to r. Then $\varsigma_\Gamma \leq \sum_{i=1}^{r} E[t_i]$.

We now bound the run time of our adder TAC by breaking the assembly process up according to waypoint assemblies $A_0 = S, A_1$, and A_2. We then bound the expected transition time from A_0 to A_1 and A_1 to A_2 to get a bound from Lemma C.3.

Let producible assembly A_1 be the seed line with the additional 4 or 5 tiles (dependant on the type of bit pair) placed atop each bit pair as shown in Figure 13 (g). As each collection of 4 or 5 tiles can assemble independently, the time t_1 to transition from the seed to a superset of A_1 is bounded by the maximum of n sums of at most 5 exponentially distributed random variables with rate $1/|T|$, where T is the tileset for the adder TAC. By Lemma C.2, $E[t_1] = O(\log n)$ time for any input pair of n-bit numbers.

Now consider the time t_2 to transition from A_1 to the unique terminal assembly of the system, i.e, the completion of the third row of the assembly. For this phase we are interested in the time taken for the last

finishing chain of tile placements for each maximal contiguous sequence of $(0,1) - (1,0)$ addend pairs. If the largest of these sequences is length k, then t_2 is bounded by the maximum of at most n sums of at most $3k$ exponentially distributed random variables with rate $1/|T|$. Lemma C.2 therefore implies $E[t_2] = O(k)$. As $k \leq n$ we get a worst case time of $E[t_2] = O(n)$. Further, by Lemma 4.1, we have that the average value of k is $O(\log n)$, yielding an average case time of $E[t_2] = O(\log n)$. Applying Lemma C.3, we get an upper bound for the total runtime of our TAC of $E[t_1] + E[t_2]$ which is $O(n)$ in the worst case and $O(\log n)$ in the average case.

Our $O(\log n)$ average case is the best that can be hoped for in the continuous time model as any adder TAC must place at least $n + 1$ tiles to guarantee correctness. From Lemma C.2, this immediately gives an $\Omega(\log n)$ lower bound in all cases:

Theorem C.4. The continuous run time for any n-bit adder TAC is $\Omega(\log n)$ in all cases.
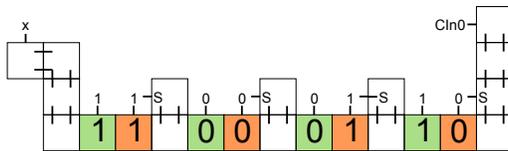
C.3 Correctness.

Figure 17: The scaffolding needed for the computation.

The analysis for the average case $O(\log n)$ worst case $O(n)$ adder TAC will begin from the point of the construction after which all scaffolding is in place (Figure 17).

The first step for any addend-pair in the seed is to perform its addition. This addition step will output either $(1, x)$, $(0, x)$, or $(x, \neg x)$ according to the following formulas:

$$1 + 1 = (1, x)$$

$$0 + 0 = (0, x)$$

$$1 + 0 = (x, \neg x)$$

$$0 + 1 = (x, \neg x)$$

The first value in the output represents the carry-out of the addend-pair and the second value represents the value of the addition. In both cases, x refers to the unknown carry-in which will come from the previous addend-pair's carry out.

Given any addend-pair, we can then determine whether or not we have enough information to generate a carry by looking at the first position of the generated output pair in the first step. The two cases that can generate a carry, $(1, x)$ and $(0, x)$, do just that and immediately propagate a 1 or 0 respectively. The other case, $(x, \neg x)$, simply waits for a carry to come in. No addend-pair can calculate its value until it receives a carry from the previous addend-pair. Once an addend-pair receives a carry-in, it can replace any x or $\neg x$ with the proper value and it can "print" the correct value, i.e. the solution at that particular position. Also, if k_0 was x it can now propagate its carry to the next addend-pair.

D $O(\sqrt{n})$ Time Addition.

D.1 Construction.

Input/Output Template. Figure 19(a) and Figure 19(b) are examples of I/O templates for a 9-bit adder TAC. The inputs to the addition problem in this instance are two 9-bit binary numbers A and B with the
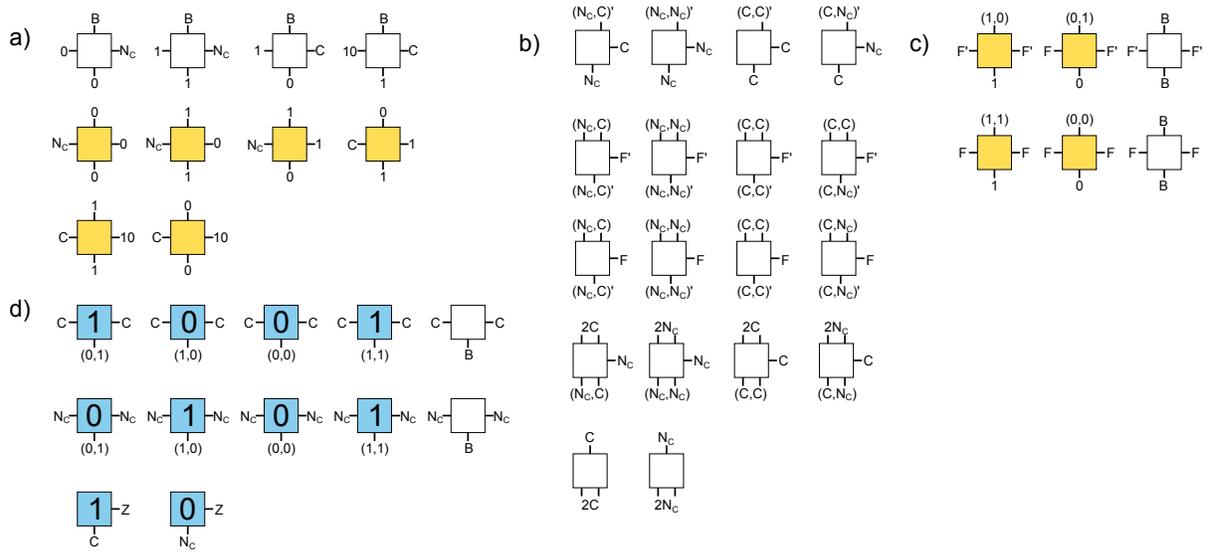
Figure 18: The complete tile set. a) Tiles involved in bit addition. b) Carry propagation tiles. c) Tiles involved in incrementation. d) Tiles that print the answer.

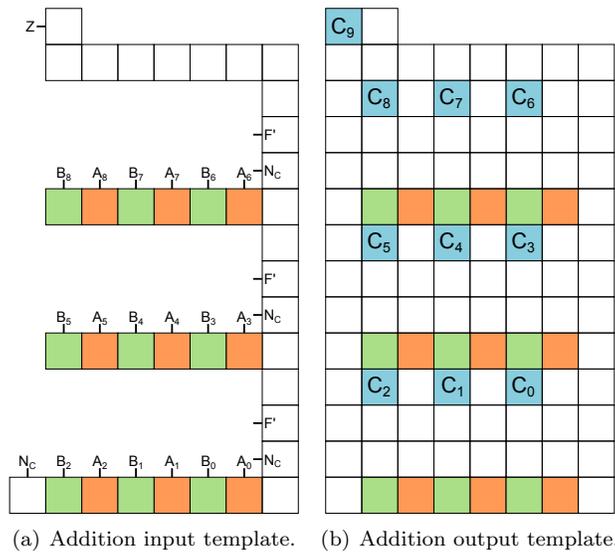(a) Addition input template.　(b) Addition output template.

Figure 19: These are example I/O templates for the worst case $O(\sqrt{n})$ time addition introduced in Section 5.

least significant bit of A and B represented by A_0 and B_0, respectively. The north facing glues in the form of A_i or B_i in the input template must either be a 1 or a 0 depending on the value of the bit in A_i or B_i. The placement for these tiles is shown in Figure 19(a) while a specific example of a possible input template is shown in Figure 20a. The sum of $A + B$, C, is a ten bit binary number where C_0 represents the least significant bit. The placement for the tiles representing the result of the addition is shown in Figure 19(b) while a specific example of an output is shown in Figure 22j.

To construct an n-bit adder in the case that n is a perfect square, split the two n-bit numbers into $\sqrt{n}$ sections each with $\sqrt{n}$ bits. Place the bits for each of these two numbers as per the previous paragraph, except with $\sqrt{n}$ bits per row, making sure to alternate between A and B bits. There will be the same amount of space between each row as seen in the example template 19(a). All Z, N_C, and F', must be placed in the same relative locations. The solution, C, will be in the output template s.t. C_i will be three tile positions above B_i and a total of size $n + 1$.

Below, we use the adder tile set to add two nine-bit numbers: $A = 100110101$ and $B = 110101100$ to demonstrate the three stages in which the adder tile system performs addition.

Step One: Addition. With the inclusion of the seed assembly (Figure 20a) to the tile set (Figure 18), the first subset of tiles able to bind are the addition tiles shown in Figure 18a. These tiles sum each pair of bits from A and B (for example, $A_0 + B_0$) (Figure 20b). Tiles shown in yellow are actively involved in adding and output the sum of each bit pair on the north face glue label. Yellow tiles also output a carry to the next more significant bit pair, if one is needed, as a west face glue label. Spacer tiles (white) output a B glue on the north face and serve only to propagate carry information from one set of A and B bits to the next. Each row computes this addition step independently and outputs a carry or no-carry west face glue on the westernmost tile of each row (Figure 20c). In a later step, this carry or no-carry information will be propagated northwards from the southernmost row in order to determine the sum. Note that immediately after the first addition tile is added to a row of the seed assembly, a second layer may form by the attachment of tiles from the increment tile set (Figure 18)c. While these two layers may form nearly concurrently, we separate them in this example for clarity and instead address the formation of the second layer of tiles in *Step Two: Increment* below.
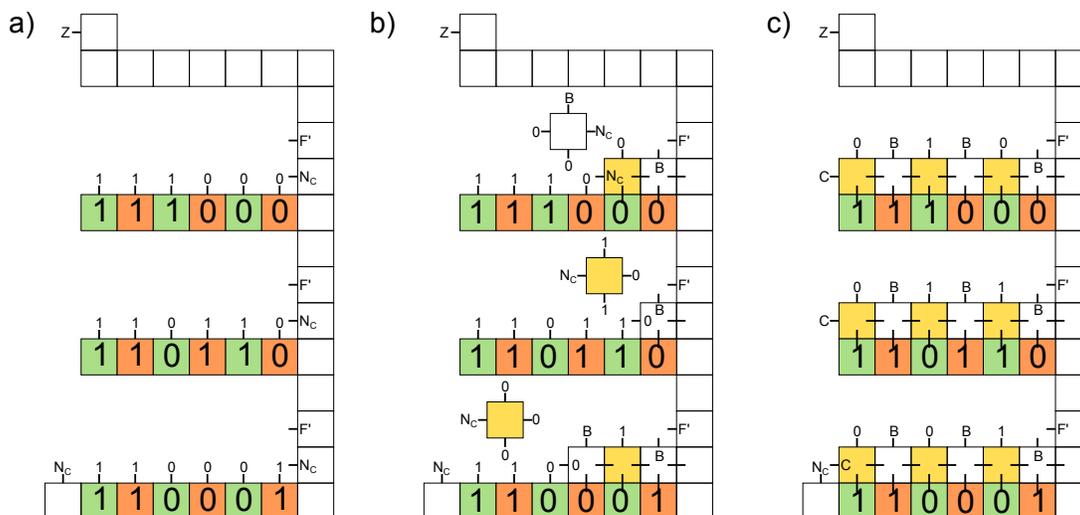

Figure 20: Step 1: Addition.

Step Two: Increment. As the addition tiles bind to the seed, tiles from the incrementation tile set (Figure 18c) may also begin to cooperatively attach. For clarity, we show their attachment following the

completion of the addition layer. The purpose of the incrementation tiles is to determine the sum for each A and B bit pair in the event of a no-carry from the row below and in the event of a carry from the row below (Figure 21).The two possibilities for each bit pair are presented as north facing glues on yellow increment tiles. These north face glues are of the form (x, y) where x represents the value of the sum in the event of no-carry from the row below while y represents the value of the sum in the event of a carry from the row below. White incrementation tiles are used as spacers, with the sole purpose of passing along carry or no-carry information via their east/west face glues F', which represents a no-carry, and F, which represents a carry.
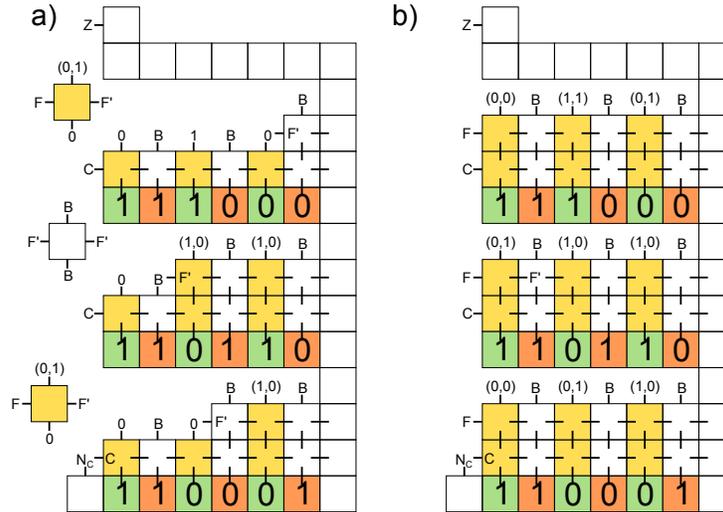

Figure 21: Step 2: Increment.

Step Three: Carry Propagation and Output. The final step of the addition mechanism presented here propagates carry or no-carry information northwards from the southernmost row of the assembly using tiles from the tile set in Figure 18b and then outputs the answer using the tile set in Figure 18d. Following completion of the incrementation layers, tiles may begin to grow up the west side of the assembly as shown in Figure 22a. When the tiles grow to a height such that the empty space above the increment row is presented with a carry or no-carry as in Figure 22b, the output tiles may begin to attach from west to east to print the answer (Figure 22c). As the carry propagation column grows northwards and presents carry or no carry information to each empty space above each increment layer, the sum may be printed for each row Figures 22d-e. When the carry propagation column reaches the top of the assembly, the most significant bit of the sum may be determined and the calculation is complete (Figure 22f).

D.2   Time Complexity.

D.2.1   Parallel Time

Using the runtime model presented in Section A *Run-time* we will show that the addition algorithm presented in this section has a worst case runtime of $O(\sqrt{n})$. In order to ease the analysis we will assume that each logical step of the algorithm happens in a synchronized fashion even though parts of the algorithm are running concurrently.

The first step of the algorithm is the addition of two $O(\sqrt{n})$ numbers co-located on the same row. This first step occurs by way of a linear growth starting from the leftmost bit all the way to the rightmost bit of the row. The growth of a line one tile at a time has a runtime on the order of the length of the line. In the case of our algorithm, the row is of size $O(\sqrt{n})$ and so the runtime for each row is $O(\sqrt{n})$. The addition

Figure 22: Step 3: Carry Propagation and Output.

of each of the $\sqrt{n}$ rows happens independently, in parallel, leading to a $O(\sqrt{n})$ runtime for all rows. Next, we increment each solution in each row of the addition step, keeping both the new and old values. As with the first step, each row can be completed independently in parallel by way of a linear growth across the row leading to a total runtime of $O(\sqrt{n})$ for this step. After we increment our current working solutions we must both generate and propagate the carries for each row. In our algorithm, this simply involves growing a line across the leftmost wall of the rows. The size of the wall is bounded by $O(\sqrt{n})$ and so this step takes $O(\sqrt{n})$ time. Finally, in order to output the result bits into their proper places we simply grow a line atop the line created by the increment step. This step has the same runtime properties as the addition and increment steps. Therefore, the output step has a runtime of $O(\sqrt{n})$ to output all rows.

There are four steps each taking $O(\sqrt{n})$ time leading to a total runtime of $O(\sqrt{n})$ for this algorithm. This upper bound meets the lower bound presented in Corollary 3.1 and the algorithm is therefore optimal.

Choice of $\sqrt{n}$ rows of $\sqrt{n}$ size. The choice for dividing the bits up into a $\sqrt{n} \times \sqrt{n}$ grid is straightforward. Imagine that instead of using $O(\sqrt{n})$ bits per row, a much smaller growing function such as $O(\log n)$ bits per row is used. Then, each row would finish in $O(\log n)$ time. After each row finishes, we would have to

propagate the carries. The length of the west wall would no longer be bound by the slow growing function $O(\sqrt{n})$ but would now be bound by the much faster growing function $O(\frac{n}{\log n})$. Therefore, there is a distinct trade off between the time necessary to add each row and the time necessary to propagate the carry with this scheme. The runtime of this algorithm can be viewed as the $max(row_{size}, westwall_{size})$. The best way to minimize this function is to divide the rows such that we have the same number of rows as columns, i.e. the smallest partition into the smallest sets. The best way to partition the bits is therefore into $\sqrt{n}$ rows of $\sqrt{n}$ bits.

D.2.2 Continuous Time

To bound the continuous run time of our TAC for any pair of length n-bit input strings we break the assembly process up into 4 phases. Let the phases be defined by 5 producible assemblies $S = A_0, A_1, A_2, A_3, A_4 = F$, where F is the final terminal assembly of our TAC. Let t_i denote a random variable representing the time taken to grow from A_{i-1} to the first assembly A' such that A' is a superset of A_i. Note that $E[t_1] + E[t_2] + E[t_3] + E[t_4]$ is an upper bound for the continuous time of our TAC according to Lemma C.3. We now specify the four phase assemblies and bound the time to transition between each phase to achieve a bound on the continuous run time of our TAC for any input.

Let A_1 be the seed assembly plus the attachment of one layer of tiles above each of the input bits (see Figure 20 (c) for an example assembly). Let A_2 be assembly A_1 with the second layer of tiles above the input bits placed (see Figure 21 (b)). Let A_3 be assembly A_2 with the added vertical chain of tiles at the western most edge of the completed assembly, all the way to the top ending with the final blue output tile. Finally, let A_4 be the final terminal assembly of the system, which is the assembly A_3 with the added third layer of tiles attached above the input bits.

Time t_1 is the maximum of time taken to complete the attachment of the 1st row of tiles above input bits for each of the $\sqrt{n}$ rows. As each row can begin independently, and each row grows from right to left, with each tile position waiting for the previous tile position, each row completes in time dictated by a random variable which is the sum of $\sqrt{n}$ independent exponentially distributed random variables with rate $1/|T|$, where T is the tileset of our adder TAC. Therefore, according to Lemma C.2, we have that $t_1 = \Theta(\sqrt{n} + \log \sqrt{n}) = \Theta(\sqrt{n})$. The same analysis applies to get $t_2 = \Theta(\sqrt{n})$. The value t_3 is simply the value of the sum of $4\sqrt{n} + 2$ exponentially distributed random variables and thus has expected value $\Theta(\sqrt{n})$. Finally, $t_4 = \Theta(\sqrt{n})$ by the same analysis given for t_1 and t_2. Therefore, by Lemma C.3, our TAC has worst case $O(\sqrt{n})$ continuous run time.

D.3 Correctness.

The first two steps of the algorithm are addition followed by incrementation. It is important to note that this incrementation step not only outputs both the original (addition result) and incremented value but also whether or not the original value contained a zero. This is important because it will allow us to later use this information to decide whether or not a row will contain a carry. Now, these two steps of the algorithm rely on nothing but the data in the current row and are completed in parallel across all rows. Therefore, with the given tile set in Figure 18, these two steps are straightforward to verify.

The next step is for every row to select a solution from the two that were generated as well as propagate its carry information. We will, for the moment, assume that some row has not received the information of the previous carry and will concentrate on this row. At this step we know if the current row generated a carry $C \in \{T, F\}$, if the rows sum contains a zero $Z \in \{T, F\}$, and two possible row values A and B. A represents the sum if the row receives a carry-in of 0 and B the opposite. In order to continue from this step an answer must be selected and a carry or no-carry must be propagated. If $Z = T$, then we immediately know that we will propagate whatever C may be. If $Z = F$ and $C = T$, we also know that we must propagate a carry. The only situation in which we do not know whether we will propagate a carry is when $Z = F$ and $C = F$. When we encounter this situation we propagate whatever the carry was in the previous row. Also, in order to decide whether we will select A or B we need only the previous carry. Therefore, assuming we have the

correct previous carry we can correctly select both the proper value for this row as well as propagate the correct carry to the next row.

Finally, because we know the initial carry is correct (it is part of the seed), we know that the first row can select the correct result as well as propagate the correct carry. Then, because we know that the first row's carry is correct, we know that the second row can select the correct result as well as propagate the carry. This chain continues until it reaches the last row leading to selecting all the correct values as well as propagating all the correct carries.

The last step is to select the most significant bits value which is solely based on the last row's carry. If the last row propagates a carry it is a 1, otherwise it is a 0. Since we know that the last carry is correct we know that the last value is selected properly.

E $O(\log n)$ Average Case, $O(\sqrt[2]{n})$ Worst Case Addition.

E.1 Construction

This construction combines the two-dimensional scaffold principle of the simple worst-case addition construction (Section 5) with the principle that certain addend-pairs can compute a carry out before they get a carry in, which was shown to reduce the average case run-time to $O(\log n)$ in Section C.2. Contrary to the addition construction in Section D.1, the directionality of adjacent rows is antiparallel in the construction described here. Every odd row beginning with row one, which is the southernmost row, has the least significant bit on the east and the most significant bit on the west. Every even row has bits in the opposite order, as shown in Figure 25a. Each odd row, along with the even row above, should be considered as a single section in which the addition mechanism is nearly identical to the $O(\log n)$ average case adder TAC. Within each section, carry outs are propagated east to west on the odd row, up in constant time from the most significant bit on the odd row (OMSB) to the least significant bit on the even row (ELSB), and from west to east on the even row. Adjacent rows are anti-parallel so that the distance between the most significant bit (MSB) of each section is a constant distance from the least significant bit (LSB) of the section above. This modification allows us to apply the $O(\log n)$ average case addition between each pair of bits, and at the same time apply the carry propagation mechanism of the $O(\sqrt{n})$ worst case addition construction between the MSB of each even row.

Carry Passing and Prediction of Results Within Sections. Within each section, or odd/even row pair, carry outs are propagated as stated in the above paragraph (Figure 26a). In addition, each row will present "predicted" results, that is, the result of the computation as if there were no carry from the MSB of the previous section. These results are shown as yellow tiles in Figure 26b-c. Note the blue tiles in Figure 26d. These blue tiles represent the final sum of the addend pairs. They may be computed without carry information propagated from a section below because the addend pairs are either 1)located in the southernmost section, where it is immediately known that the carry-in is 0 or 2)are flanked by a less significant addend pair for which the carry out is immediately known. The north face glues on yellow tiles in Figure 26d which contain a star * rely on a carry from a previous section. The mechanism of carry propagation within a section can be seen in Figure 27a-b. Vertical columns grow up along the west side of each section to propagate the carry from the odd row to the even row.

Carry Propagation Between Sections. Figure 27c shows a vertical column growing between the southernmost section and the section above. This column is propagating a carry from the most significant bit of the southernmost section A to the least significant bit of the section B to the north. This information continues to the most significant bit of section B at which point a carry bit is computed to propagate to the section north of B (Figure 27d, Figure28a. The terminal tile assembly with the computed sum is presented in Figure 28b.
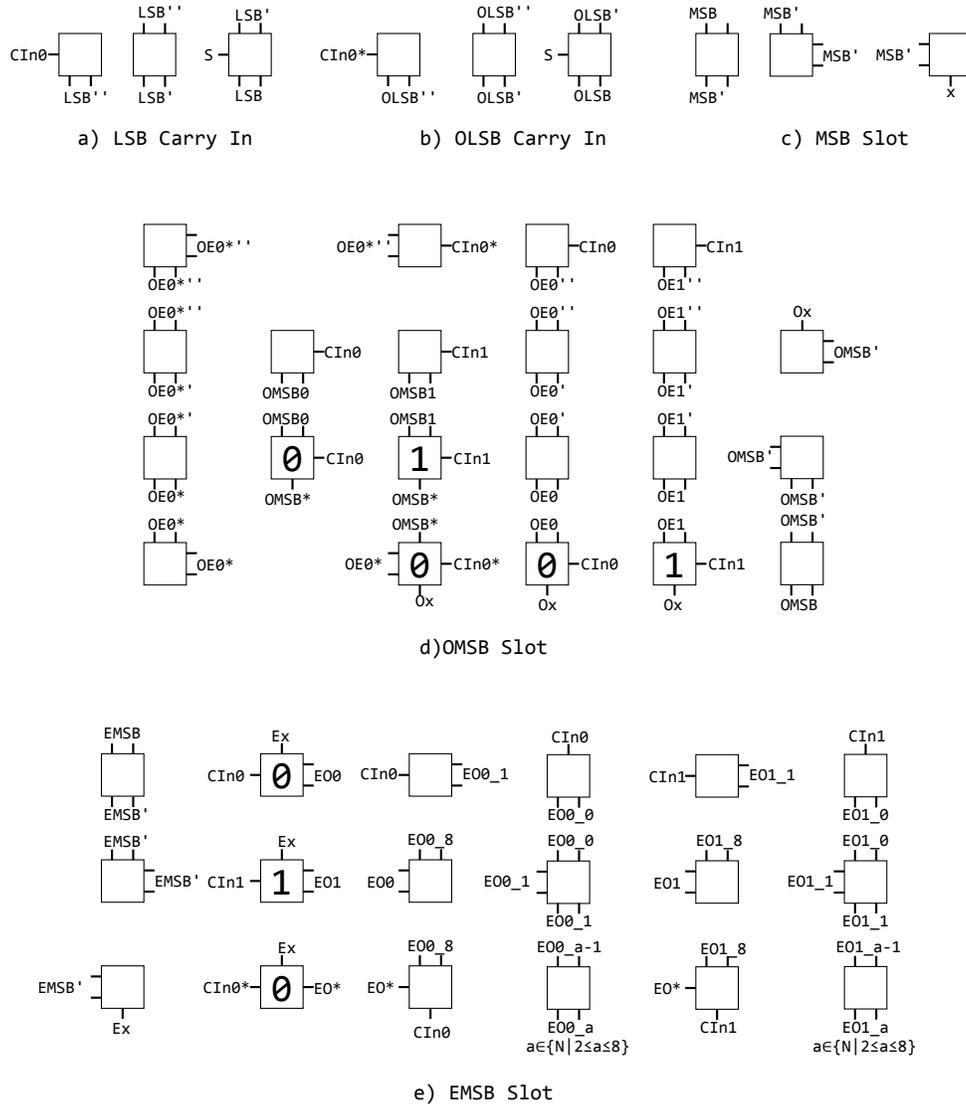
Figure 23: Partial set of tiles necessary to implement $O(\log n)$ average case, $O(\sqrt{n})$ worst case combined addition (see also next figure).
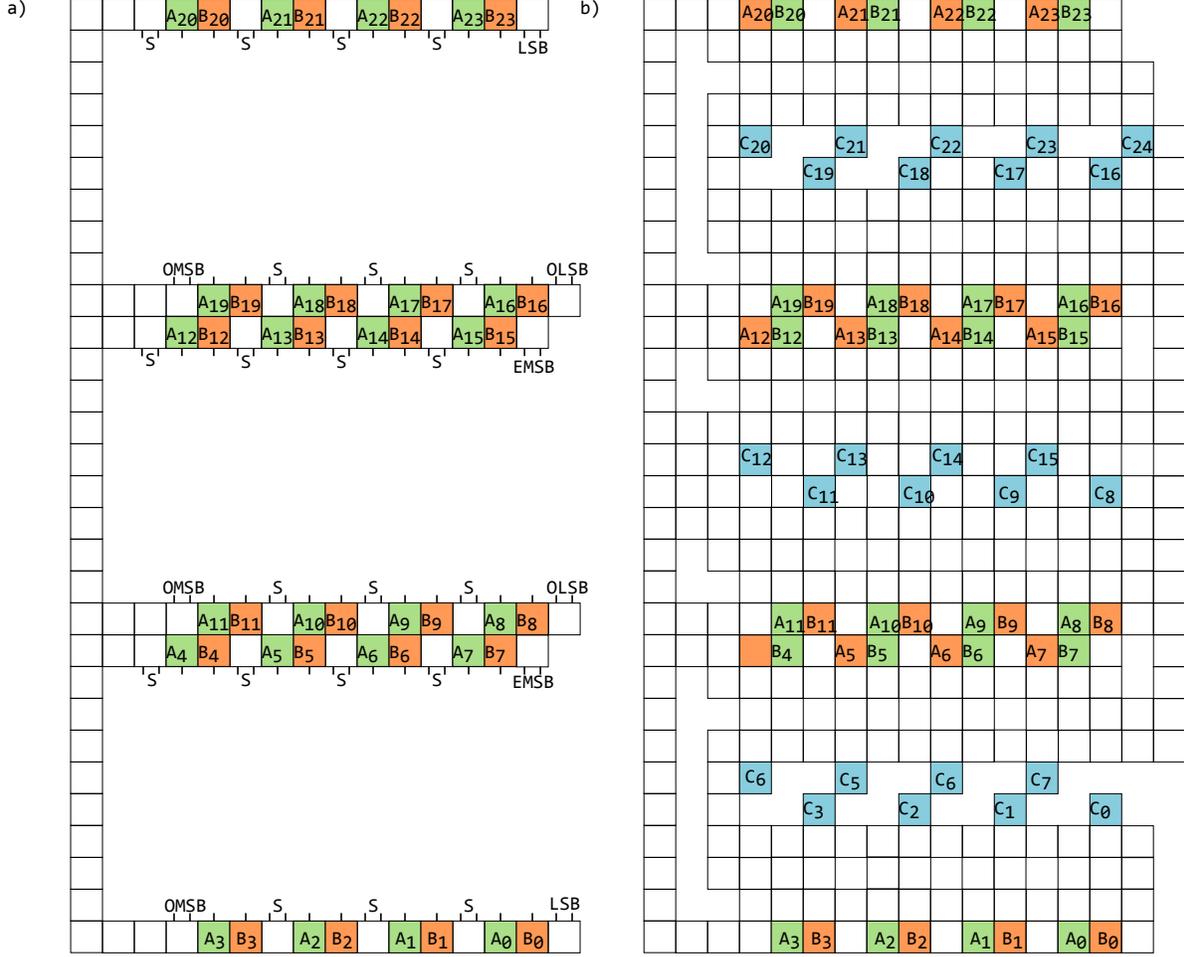
30

Figure 24: Tiles necessary to implement $O(\log n)$ average case, $O(\sqrt{n})$ worst case combined addition (continued from previous figure).

Figure 25: a) The input template for addition of two n-bit binary numbers A and B. b)The output template for the addition of two n-bit binary numbers A and B.

E.2 Time Complexity.

$O(\sqrt{n})$ **Worst Case and** $O(\log n)$ **Average Case Run-Time**   The construction presented in this section represents a combination of the constructions presented in Sections 4 and 5. Thus, this runtime analysis combines elements from the time complexity proofs in C.2 and D.2. In the construction described here, each section on the scaffold precomputes the sum as if there were a carry from the previous section and also as if there were no carry from the previous section. In some cases, whether a carry bit is passed in from the section below is irrelevant and parts of the final sum may be generated before this information is available. If the most significant addend-pair bits (MSB) of a given section can immediately generate a carry out for propagation to the next section, they do so. When a carry in arrives from the previous section, final values are selected from the precompution step, if necessary. In this analysis of time complexity, we first analyze the run time of the precomputataion step, and then consider the run time of propagating carry bits from section to section up the scaffold.

Consider the binary sequence T of length $2n$ composed of n-bit binary numbers A and B as defined in Section C.2. For this construction, divide T into $\frac{\sqrt{n}}{2}$ smaller sequences, or sections, $S_0, S_1, \cdots, S_{\frac{\sqrt{n}}{2}-1}$, with each section containing $2\sqrt{n}$ addend-pairs. These sections are arranged on a scaffold as depicted in Figure 29. Note that the distance between the bottom half and the top half of a given section is constant and requires

Figure 26: a) Carry out bits are computed for each addend pair where a carry out can be deduced immediately. b-c) "Predicted" results (yellow tiles) begin to attach. d) Blue tiles represent final result.
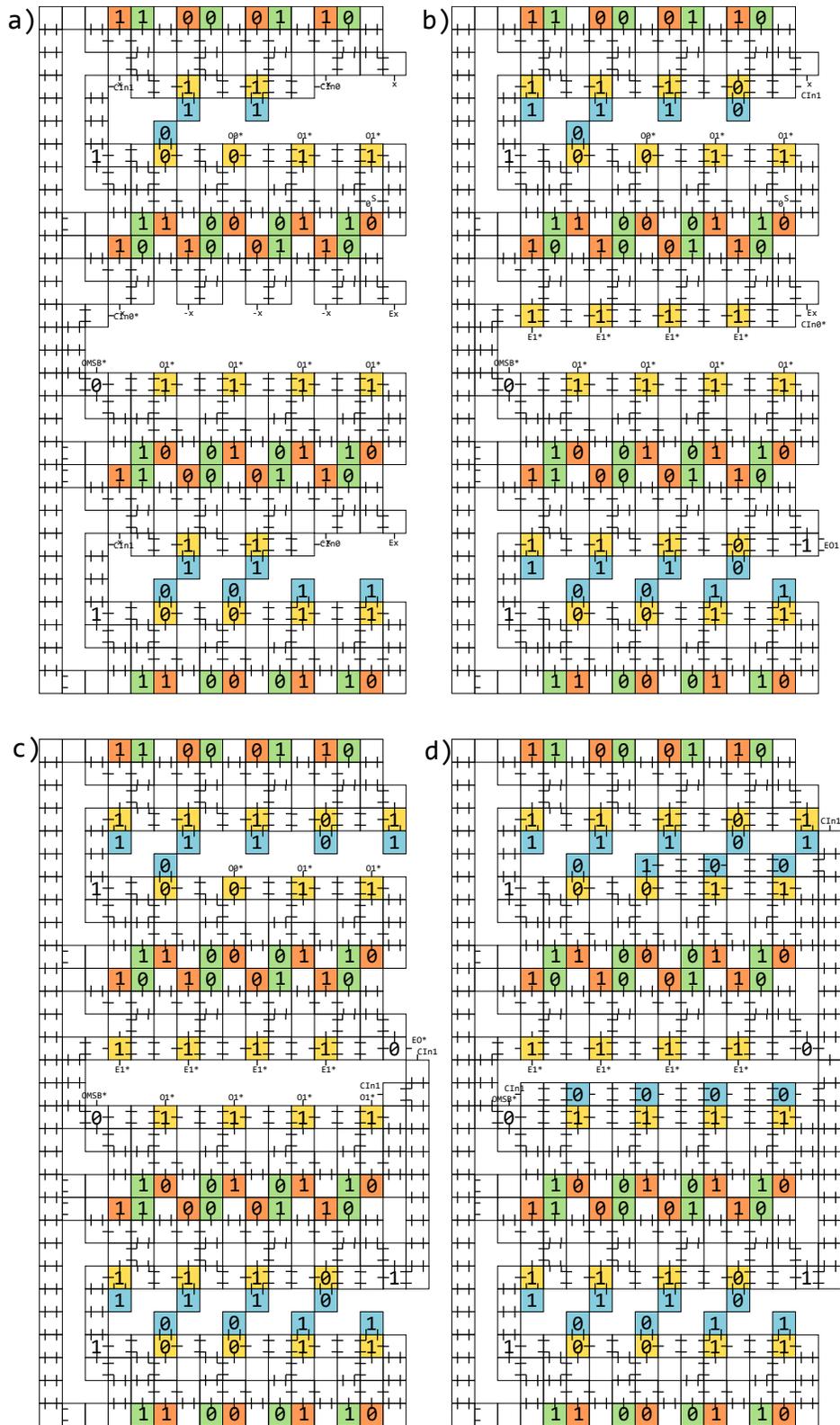
Figure 27: a-b) Carry bits are propagated within sections from the MSB of the odd row to the LSB of the even row using vertical columns. c-d)Carry bits are propagated between sections via vertical columns on the east side of the assembly.
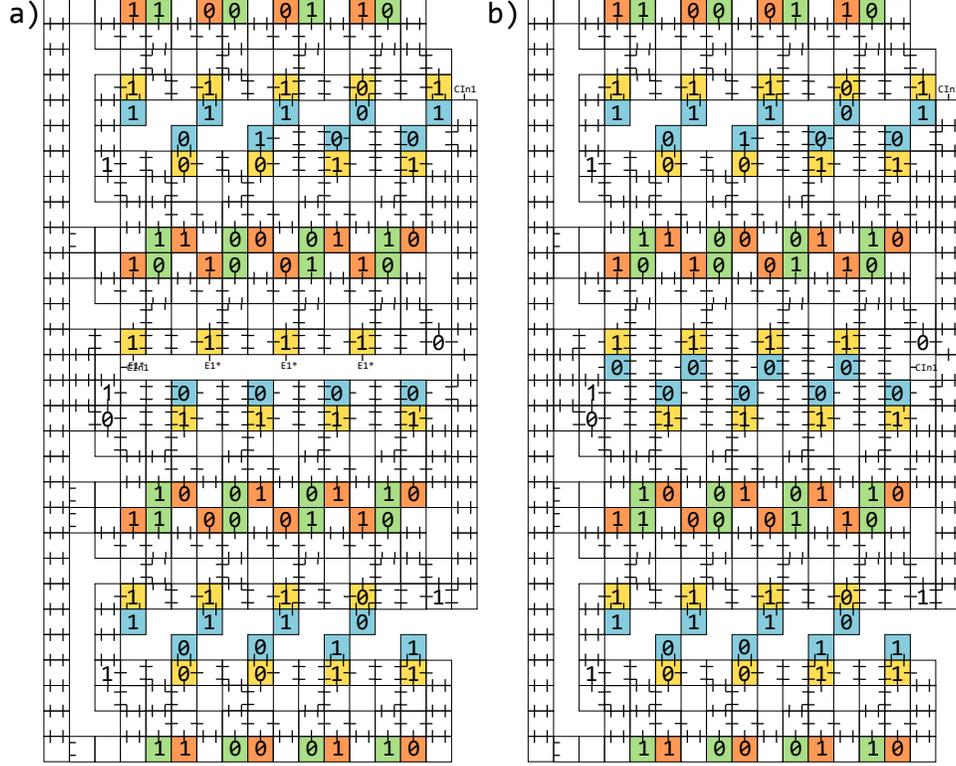
34

Figure 28: a) A carry bit from the first section propagates through the middle section in this assembly. b) The terminal assembly displaying the output C.

a constant number of steps to traverse. We first treat each section, S_i, as an independent addition problem without regard for a carry in from the previous section. Define k as the longest contiguous sequence of $(0,1)$ and $(0,1)$ addend-pairs in S_i. It follows from the proof in Section C.2 that the run-time for S_i is bounded upwards by the length of k. The worst-case runtime for addition over the sequence S_i would thus occur when $k = 2\sqrt{n}$. Therefore, the worst case run time for addition *within* a section is $O(\sqrt{n})$. It also follows, as described in Section C.2, that the expected length of k is $O(\log n)$. Therefore, the average run time for each addition within each section is $O(\log n)$. The precomputation within each of the $\frac{\sqrt{n}}{2}$ sections occurs independently in parallel, leading to a worst case $O(\sqrt{n})$, average case $O(\log n)$ precomputation run-time for all sections.

After performing this precomputation within each section, we must propagate carries between each section. The distance between two neighboring sections is constant and may be traversed by a column of tiles in a constant number of steps. Therefore, a carry out bit, once generated, can be propagated from one section to the next in constant time. A carry out bit may be propagated to the next section, S_{i+1}, immediately if the most significant addend pair of section S_i consists of $(0,0)$ or $(0,1)$. Otherwise, this most significant addend pair of S_i must wait for a carry in before generating a carry out. Therefore, the composition of the most significant addend-pairs of the sections acts as a limiting factor to the speed with which carries may be propagated across all of the sections. Consider the binary sequence P which is comprised of the most significant addend pairs of each section and has a length of $\frac{\sqrt{n}}{2}$ addend-pairs. Let k be the longest contiguous sequence of $(1,0)$ and $(0,1)$ addend-pairs in P. The propagation of carry bits through P is bounded upwards by the length of k, with the worst-case being when $k = \frac{\sqrt{n}}{2}$ and an average case being when $k = O(\log n)$. Thus, the propagation of carry bits between each section up to the most significant addend-pair of T is bounded upwards by $O(\sqrt{n})$ and has an average run-time of $O(\log n)$. Therefore, this addition algorithm
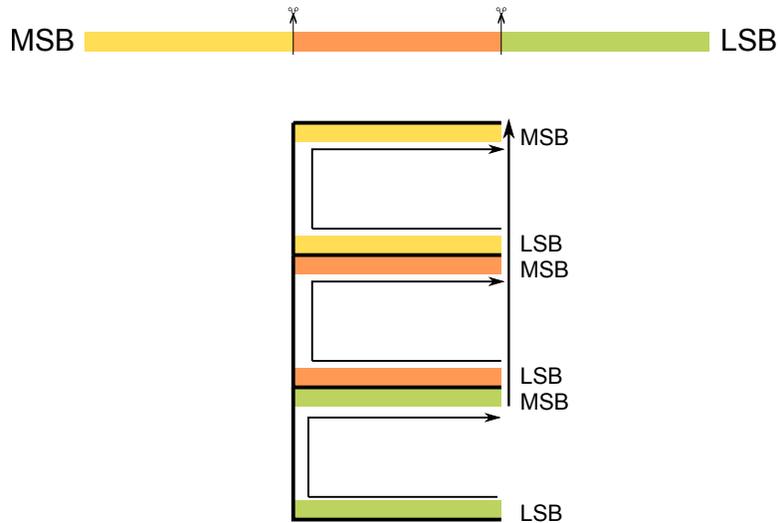
35

Figure 29: Top: T is separated into sections. Bottom: Each section of T is arranged on a scaffold. Arrows represent the general direction of carry bit propagation. MSB denotes the most significant bit of a section, and LSB denotes the least significant bit of a section.

has an upper bound of $O(\sqrt{n})$ and an average run-time of $O(\log n)$.

The parallel run time analysis presented above can be extended to the continuous run time model through application of Lemma C.2 and Lemma C.3 similar to what is done in Sections C.2.2 and D.2.2, yielding worst case $O(\sqrt{n})$ and average case $O(\log n)$ continuous run time bounds.

E.3   Correctness.

Every row with north facing glues performs its addition in the exact same way as described in Section C.1 assuming an incoming carry of 0. As such we use the proof of correctness from Section C.3 to show that this step is correct. Also, every row with south facing glues performs this same addition except with an incoming carry from the addition of the lower north facing glue side. Thus, the same proof also applies to this side. Considering both of these additions as the first step, we can say that the first step correctly adds a section with a carry-in of 0.

Since the first section has no sections before it, the addition of the first section has an input carry of 0 allowing the copying of its value up to the final "display" row. Once an addend-pair pair knows its carry-in it can display its results. This combined with the fact that we proved the addition of any section with an input carry of 0 proves the correctness for the first section.

Once a section finishes the first step, there are three possible carry-outs from the MSB addend-pair in the section: 0, 1, and 0∗. If the section's MSB addend-pair propagates a 0 or 1 then somewhere in the section some addend-pair generated its carry without depending on a carry from the previous section leading to a correct propagation. If the section's MSB addend-pair propagates a 0∗, then no addend-pair in the section was able to generate a carry, meaning that no addend-pair in the section contained equal bits. Therefore, the section will correctly propagate whatever carry it received to the next section. These two clauses cover all cases in terms of carry propagation from one section to next section.

Every section other than the first one performs its addition with an input carry of 0∗ which indicates an unknown carry with a value of 0. In other words, it continues the calculation assuming a 0 input carry but cannot copy any undetermined values, due to the unknown carry, into the "display" row until it receives the real carry from the previous section. This carry only gets propagated until it reaches an addend-pair with equal bits because at this point the addend-pair would have already generated its carry regardless of any incoming input carry and propagated it. One can see that the beginning of any section, other than

the first, is essentially calculated by the algorithm as if it was in the center of some series of addend-pairs with unequal bits. When the correct carry propagates from the previous section to the current section, the correct values may then be copied into the "display" row. If a 0 is carried in, then it is a simple copy up of the value into the "display" row. If a 1 is carried in, then it is the inverse of what was previously calculated. Assuming some section receives the correct carry from the previous section, that section will "display" the correct result. We have shown that the first section propagates a carry correctly, and therefore all subsequent sections propagate their correct carry out. We have also shown that if each section propagates the correct carry-out to the next section, then the addition of addend-pairs within each section is performed correctly, producing the correct sum of A and B, C.

F Multiplication

F.1 Construction.

This TAC uses a shift-and-add algorithm for multiplication. Let A and B be two n-bit binary numbers, and let a_i and b_i represent the ith bits of A and B respectively. Note that if $B = \sum_i b_i 2^i$ for $b_i \in 0, 1$, then the product $AB = \sum_i Ab_i 2^i$. In this algorithm we perform the multiplication by multiplying A by each of the powers of two in B. Each of these partial products is added to the running sum as soon as the partial product is available.

    To describe our construction we focus on the case of multiplying two 64-bit numbers.

Vector Label And Tile Types. To permit a high-level description of the multiplication TAC, we use a *Vector Label* technique to designate the binding possibilities between tiles. Rather than showing the glue types and their placement on each edge of a tile, we label a tile with a vector which describes the rules by which that tile may bind to adjacent tiles. A tile with such a vector actually describes a group of tiles that contains a certain element in the vector that could attach to a certain assembly. The number of tile types described by a Vector Labeled tileset is at most $O(l^d)$, where l is the number of label types and d is the vector dimension. An example set of Vector Labeled tiles may be found in Figure 30.

Figure 30: a) A pair of Vector Labeled tiles describes a binding rule. Dimensions a and b must be the same on both tiles for the tiles to bind, and tiles containing r_1 in the third dimension may bind only to tiles containing r_0 in the third dimension. b)Such a binding rule can describe 8 tile types with 4 binding possibilities.

Input/Output Template. The input template encoding the multiplicand and multiplier, A and B, is a rectangular assembly of size $2n$ tiles. The product of two n-bit numbers will have at most $2n$ bits. This extra n bits of space comprises the west half of the assembly, as shown in Figure 31(a), while the east half encodes A and B via vector labeled tiles. The first and second elements of the vector on each tile are a given bit of A and B. The final assembly for 64-bit inputs is shown in Figure 31(b). The product AB is encoded on the top surface of the final assembly, marked as *Output* in Figure 31(b).

    Before going into details of the TAC implementation, we first give a brief overview of the different parts of the output structure (Fig. 31(b)). Given two n-bit inputs, A and B, our multiplication algorithm involves a

summation of n numbers, $x_0 + x_1 + ... + x_{n-1}$. For any number x_k within these n numbers, $x_k = Ab_k 2^k$. The output assembly (Fig. 31(b)) has four horizontal layers, one stacked on top of the next, each of which sums 16 of the 64 numbers to be summed for the 64-bit example. More generally, there are $\sqrt[3]{n}$ layers, $l_0, l_1, ..., l_{\sqrt[3]{n}-1}$, and each layer l_k is responsible for summing $n^{2/3}$ of the n numbers, $x_{k\sqrt[3]{n}} + x_{k\sqrt[3]{n}+1} + ... + x_{(k+1)\sqrt[3]{n}-1}$. On each layer, there are $\sqrt[3]{n}$ columns running west to east, each of which sums $n^{1/3}$ numbers of the $n^{2/3}$ numbers which are summed within each layer.

– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{56} / b_{56}	a_{55} / b_{55}*	a_{40} / b_{40}	a_{39} / b_{39}*	a_{24} / b_{24}	a_{23} / b_{23}*	a_8 / b_8	a_7 / b_7*
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{57} / b_{57}	a_{54} / b_{54}	a_{41} / b_{41}	a_{38} / b_{38}	a_{25} / b_{25}	a_{22} / b_{22}	a_9 / b_9	a_6 / b_6
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{58} / b_{58}	a_{53} / b_{53}	a_{42} / b_{42}	a_{37} / b_{37}	a_{26} / b_{26}	a_{21} / b_{21}	a_{10} / b_{10}	a_5 / b_5
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{59} / b_{59}*	a_{52} / b_{52}	a_{43} / b_{43}*	a_{36} / b_{36}	a_{27} / b_{27}*	a_{20} / b_{20}	a_{11} / b_{11}*	a_4 / b_4
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{60} / b_{60}	a_{51} / b_{51}*	a_{44} / b_{44}	a_{35} / b_{35}*	a_{28} / b_{28}	a_{19} / b_{19}*	a_{12} / b_{12}	a_3 / b_3*
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{61} / b_{61}	a_{50} / b_{50}	a_{45} / b_{45}	a_{34} / b_{34}	a_{29} / b_{29}	a_{18} / b_{18}	a_{13} / b_{13}	a_2 / b_2
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{62} / b_{62}	a_{49} / b_{49}	a_{46} / b_{46}	a_{33} / b_{33}	a_{30} / b_{30}	a_{17} / b_{17}	a_{14} / b_{14}	a_1 / b_1
– / –	– / –	– / –	– / –	– / –	– / –	– / –	– / –	a_{63} / b_{63}*	a_{48} / b_{48}	a_{47} / b_{47}*	a_{32} / b_{32}	a_{31} / b_{31}*	a_{16} / b_{16}	a_{15} / b_{15}*	a_0 / b_0

(a) Multiplication input template assembly.

(b) Multiplication output assembly.

Figure 31: The input seed and the output assembly for 64-bit inputs, A and B.

Part One: Deployment. An important step in the multiplication process is deploying the n numbers to be summed, $x_0 + x_1 + ... + x_{n-1}$. These numbers must be moved into different positions for addition to take place. First, the seed (Fig. 32(a)) is copied both up and south, as is depicted in Figure 32(b). Figure 37 demonstrates that for any 1-dimensional input, it is possible to create a tile set to send the input information into two directions. This same principle can be used to copy the 2-dimensional input area both up and south into three dimensions as is shown in Figure 32(b).

The copy of the seed to the south will then be copied to the south again and to the east, as is shown in Figure 32(c). This copy to the east will head the formation of a column. Columns run west to east in each layer, and each layer contains $\sqrt[3]{n}$ columns. In this particular 64-bit input example, there will be four columns to each layer, each of length $\sqrt[3]{n}$ blocks. The input information will continue to be propagated south and east until all columns for a given layer are formed. The input information that was copied up (Fig. 32(c)) will act as a seed for the next layer, which is formed in the same way that the first layer was formed.

As the input information is copied in different directions, different functions are applied. For the input information copied up, $n^{\frac{2}{3}}$ bits of zero are attached after the least significant bit of A, and the least significant $n^{\frac{2}{3}}$ bits of B are removed. Let $p(x) = x * 2$ and $q(x) = [x/2]$. Then in figure 32(c), $A = p(A)^{16}$ means attach 16 zeros after A and $B = q(B)^{16}$ means remove the 16 least significant bits from B. We add and remove these bits by shifting numbers. Figure 38(a) demonstrates that it is possible to create a tile set to shift any input.

The copy of the input information to the east marks the head of the first column. As this column is propagated from west to east, the least significant bits of B are removed one by one, and bits of 0 are appended to the least significant end of A one by one, as is seen in Figure 33(a). At the end of the column, a total of $n^{1/3}$ 0 bits will have been appended to A and $n^{1/3}$ of the least significant bits of B will have been removed.

The copy of the input to the south propagates further to the south and then east to form the next column (Figure 33(b)-33(c)). During this southward propagation, the least significant $n^{1/3}$ bits of B are removed,

and $n^{1/3}$ bits of zero are appended to the end of A. This southward propagation continues, and each time $n^{1/3}$ additional bits of zero are appended to the least significant end of A and $n^{1/3}$ least significant bits are removed from B, until the heads of each of the $\sqrt[3]{n}$ columns have formed on the east side. These columns are then free to grow eastward (Figure 34(a)).

The copy of the seed that is sent upward has the least significant $n^{2/3}$ bits removed from B and $n^{2/3}$ bits of 0 appended to the least significant end of A. This is the beginning of the next layer. As the information is propagated upward, it may continue to form layers, which form in parallel (Figure 33(c)). Each time the information is propagated upwards to form the next layer, another $n^{2/3}$ bits of 0 are appended to the least significant end of A, and the least significant $n^{2/3}$ bits are removed from B.
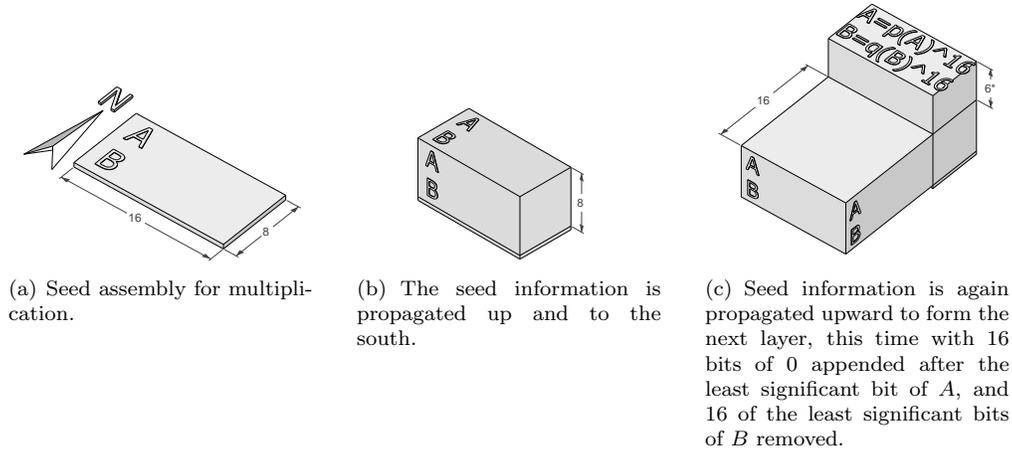

(a) Seed assembly for multiplication.

(b) The seed information is propagated up and to the south.

(c) Seed information is again propagated upward to form the next layer, this time with 16 bits of 0 appended after the least significant bit of A, and 16 of the least significant bits of B removed.

Figure 32: This series shows the propagation of information south for use in the formation of columns and up to form a new layer.

(a) This partial assembly shows the input information being modified and propagated upward for the next layer, eastward for the formation of a new column, and southward to generate more columns.

(b) This partial assembly shows the beginning of the formation of the next layer.

(c) This partial assembly shows the continued formation of new layers, and the columns within those layers.
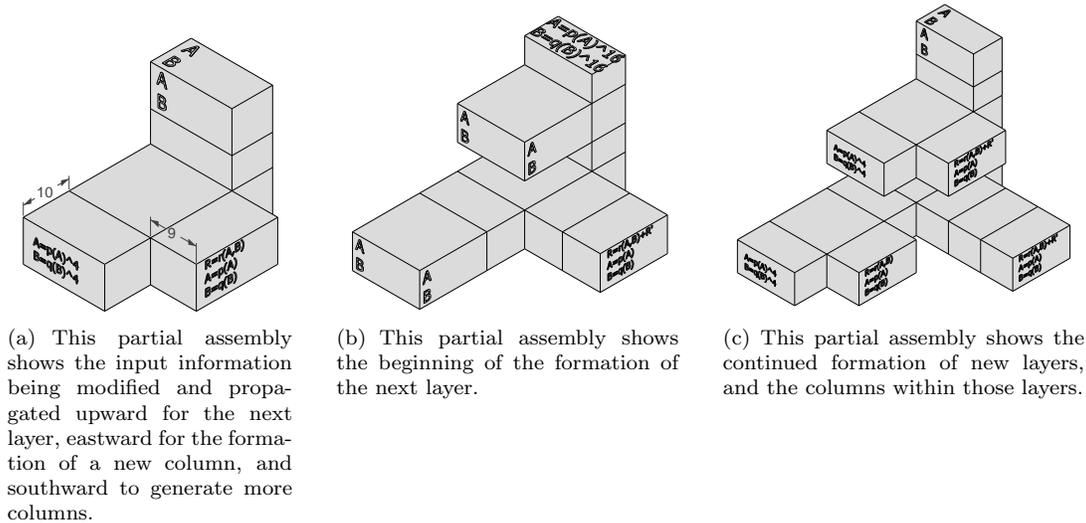
Figure 33: This series shows the process of forming new columns and layers.

Part Two: Addition. While shifting bits between each layer, and between and within each column, the tile set simultaneously performs the additions. The addition algorithm here is based on Section 5. Recall that our multiplication algorithm involves a summation of n numbers, $x_0 + x_1 + ... + x_{n-1}$. For any number x_k within these n numbers, $x_k = Ab_k 2^k$. An example of a two-number multiplication, which is required to attain x_k, is shown in Figure 36. The five elements in the vector label are the current bit of A, the current bit of B, the predicted result without a carry in, the predicted result with a carry in, and the current bit result. The numbers are arranged in zigzag fashion.

After obtaining the result of each column, shown in Figure 34(a), we copy the result of the first column, rotate this result to the south, and send it to its neighbor (Figure 34(b)). Note that we don't need A and B any more; we only copy the result. Using the reversed design of the rotate and copy box (Figure 37), we merge the two inputs from two sides into one single output. Using the same principle shown in Section 5 and Figure 36 we then sum all results from each column together, as shown in Figure 34(c) and Figure 34(d).

(a) Sums within each column have been calculated.

(b) The sum of the numbers within each column is propagated south.

(c) The sums of the numbers within each column are totaled.

(d) The sum of all the numbers within the layer is presented on the south side of the layer.
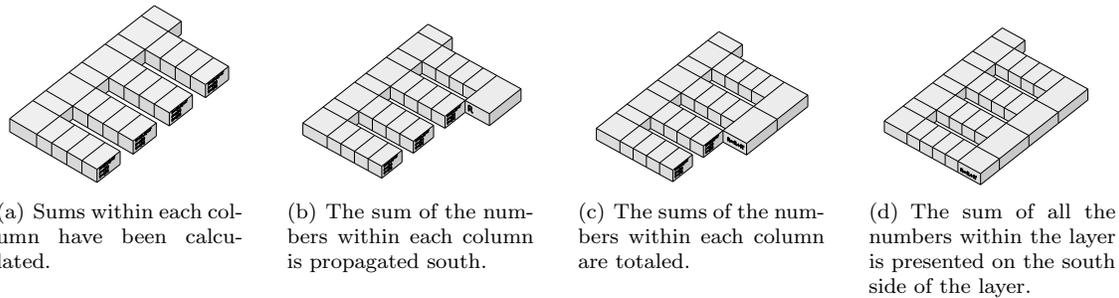
Figure 34: This series shows the process of summing all numbers within each layer.

The result of the summation in the first layer will be copied up to the second layer (Figure 35(c)), where it is added to the result from the second layer. This process is repeated until the last layer is reached and a final result is obtained, as shown in Figure 35(b) and Figure 35(c).

(a) Completed formation of the first layer of the assembly. Note that the other layers may form in parallel.

(b) All layers within the assembly for the 64-bit inputs are completed.

(c) The sum of the numbers within each layer is propagated northward and added to a running total of the sums of each layer.
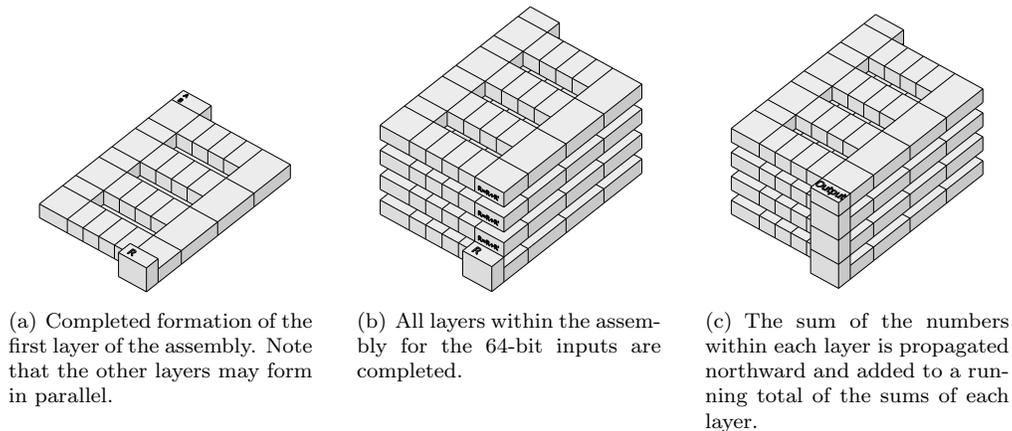
Figure 35: After summing the results of each layer, the multiplication result is displayed on the top surface of the assembly.

F.2   Time Complexity.

$O(n^{\frac{5}{6}})$ **Worst Case Run-Time.**   Our multiplication construction consists of copy (Figure 37), shift (Figure 38(a)), and addition (Figure 36) operations. The addition algorithm here is a 3D version of our $O(\sqrt{n})$ Worst

Case Addition TAC (Section D.2). In order to ease the analysis we will assume that each logical step of the algorithm happens in a step-by-step fashion even though parts of the algorithm are running concurrently.

Runtime For Copy Block. The system copies the input number up and south using a copy block. The input for a copy block is an $O(\sqrt{n})$ by $O(\sqrt{n})$ rectangle. For each row running north to south a copy is generated independently, so we analyze the process in a longitudinal section and then expand it to the entire block. Figure 37 shows an example of a copy box copying one row up and to the left. The copy box starts the copy from the leftmost input. This tile is marked as "r". If a tile is marked as "r", the tile on its right will mark its north tile as "r", otherwise, the tile will just copy itself to the north. After a tile receives an "r", the tile on its north will then send the label both up and to the left. After that, the tiles sending labels to the left will also copy the label from the bottom to the top, thus sending the labels up. Each tile under the "r" tiles is dependent on the tiles on its left and bottom, so the runtime for all tiles under the attached diagonal is $O(\sqrt{n})$. The tiles after the "r" tiles are dependent on the tiles on the right and bottom, so the total runtime of this rotate and copy box is $O(\sqrt{n})$.

Runtime For Shift Block. The system shifts the input numbers in order to attach or remove bits using a Shift Block. An example of shifting one row of input to the left is shown in Figure 38(a). In this example, bits are shifted to the left. So, from the left, the assembly starts to grow on each of the even tiles of the input at the same time, copies each of the even inputs to the second layer, then grows to the left and combines the input from each of the odd tiles in parallel. Then, the tiles on top of the odd input tiles contain both the information of the input tiles below, as well as the input tile to its right. This tile can then repeatedly be copied up and left, until it reaches the desired position. The runtime of shifting the block will be the distance it moved plus 2. In our algorithm, the length of shifting the input for the next addition is no longer than the length of the input rectangle, so the runtime is at most $O(\sqrt{n})$.

Runtime For Addition Block. Figure 36 shows an example of two-number multiplication, which is performed by stacking several Addition Blocks and Shift Blocks. Recall that in the $O(\log n)$ Average Case, $\sqrt{n}$ Worst Case Addition (Section 5), we achieved a runtime of $O(\sqrt{n})$ for a two number addition in 2D. We use the same principle to build the Addition Block in 3D. Instead of growing to the center of every two rows in 2D, the Addition Block gets its result on top of the input layer. For simplicity, we abandon the $O(\log n)$ average case addition mechanism here. Therefore, the Addition Block will take $O(\sqrt{n})$ time to finish.

In the first logical step, the assembly repeatedly stacks up the rectangular Copy blocks and Shift blocks to form a vertical column, which is used to deploy the desired numbers to all of the $O(\sqrt[3]{n})$ layers. In this vertical column, each layer contains a Copy block, and between each layer there is a Shift block, so the runtime for this vertical column is $O(\sqrt{n}) \times O(\sqrt[3]{n}) + O(\sqrt{n}) \times O(\sqrt[3]{n}) = O(n^{\frac{5}{6}})$.
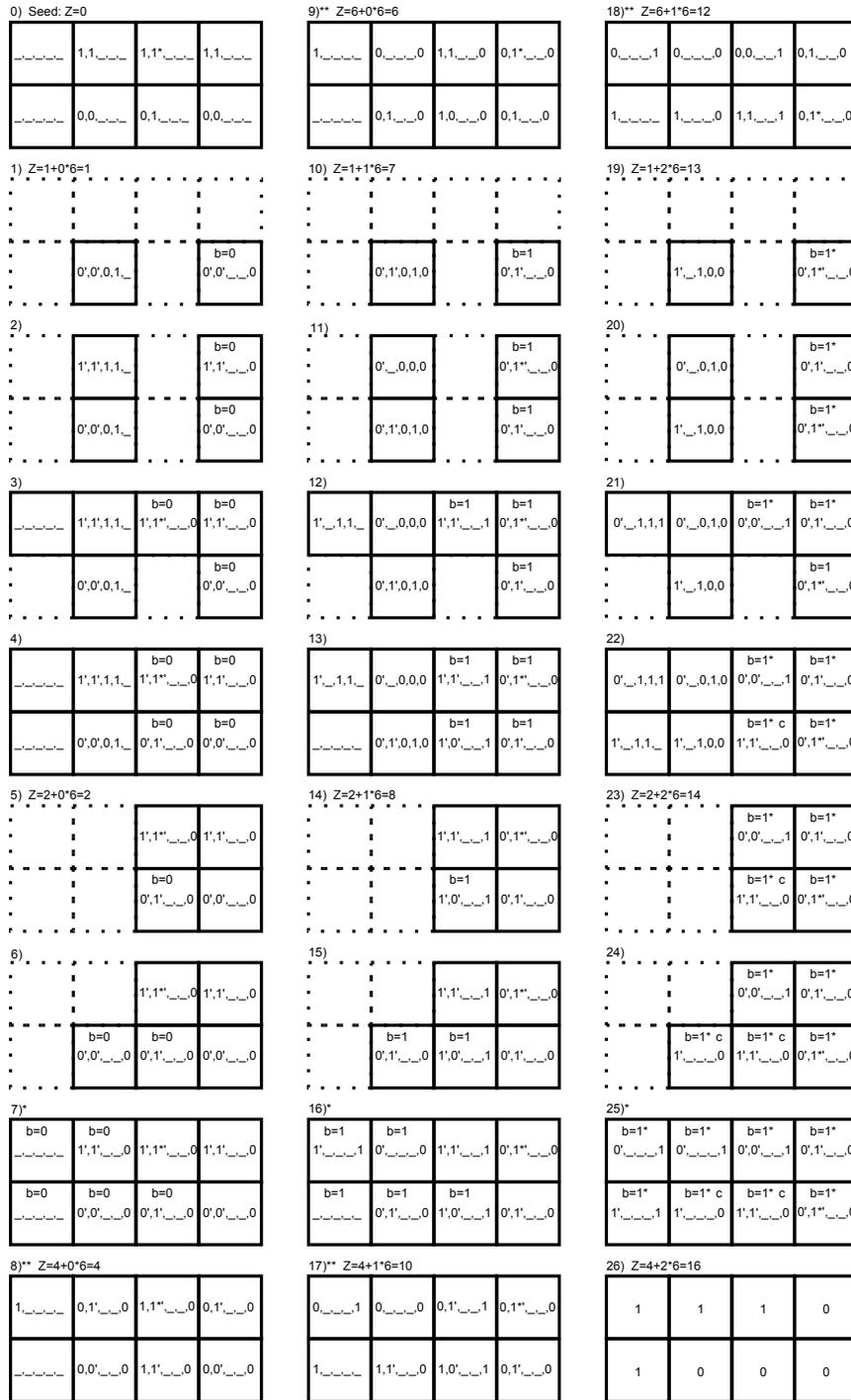
In the second step, which occurs in each layer of the assembly, a north-south row starts to grow from the copy block in the vertical column, which is also the northwest corner of each layer, in order to deploy the desired numbers to all of the $O(\sqrt[3]{n})$ columns by horizontally stacking the copy blocks and shift blocks. This process takes $O(\sqrt[3]{n})$ copy blocks and $O(\sqrt[3]{n})$ shift blocks, so the runtime for this north-south row is $O(n^{\frac{5}{6}})$.

In the third logical step, starting from each copy box in the north-south running row in each layer, columns grow eastward in order to deploy numbers while simultaneously adding them. There will be $O(\sqrt[3]{n})$ additions and $O(\sqrt[3]{n})$ shift blocks, so the runtime for each east-west columns is $O(n^{\frac{5}{6}})$

Then, the last block of the first column in each layer will rotate its information to the south and propagate it to the end of the next column, merging the next result and adding them together, then repeating these moving and adding steps until the last result of the layer is obtained. Finally, the result in the bottom layer rotates up and the results from each layer are added together. The runtime of these moving and adding steps is $O(n^{\frac{5}{6}})$. Therefore, the total runtime of the multiplication process is $O(n^{\frac{5}{6}})$.

F.3 Correctness.

Note that the multiplication of two n-bit numbers A and B is the summation of a string of numbers C_1 ... C_n, where $C_k = A * [B_k] * [2^k]$. The order in which the numbers are summed is as follows: numbers are summed starting on the first layer from from north to south, then, with the same order in each layer, from the bottom layer to the top layer. Within each layer, the west-to-east growing columns sum numbers

Figure 36: This sequence (Parts 1-26) demonstrates the product of A and B, where $A = 100110$ and $B = 1011 * 10$. Z denotes the number of layers shown in the figure. The effective bits of B are the bits after and including the bit with the $*$ symbol. So the actual multiplication is $100110 \times 110 = 11100100$. Parts 1-7 show the first addition block. Part 8 shows that A is been shifted one bit ahead after two layers, and Part 9 shows B has been shifted to another direction after another two layers.
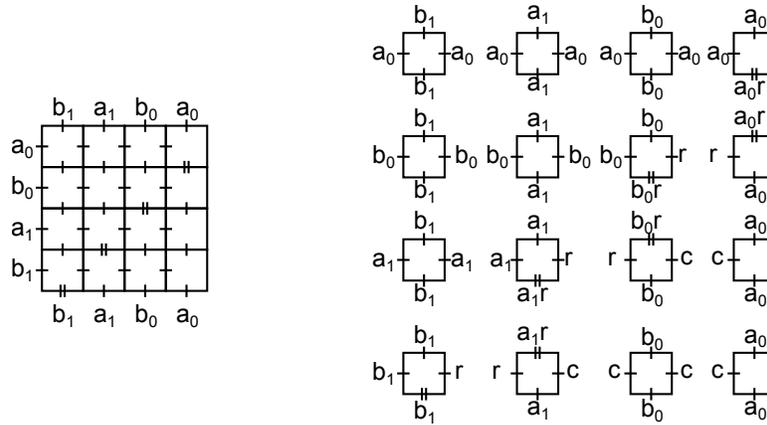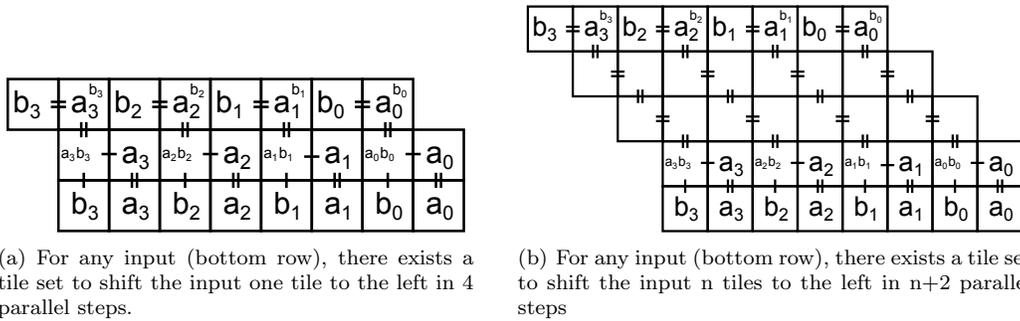
Figure 37: There exists a tile set to copy any input (bottom row) in two directions.

(a) For any input (bottom row), there exists a tile set to shift the input one tile to the left in 4 parallel steps.

(b) For any input (bottom row), there exists a tile set to shift the input n tiles to the left in n+2 parallel steps

Figure 38: The above two figures show how an input can be bit-shifted.

beginning with the northmost column and ending with the southmost column. The head of each column contains $A * [2^i]$ and b_i, where i is the number of columns times $\sqrt[3]{n}$, $1 <= i < n$. Then, starting from the head in each column, shifting and adding are performed. There will be $\sqrt[3]{n}$ shifts performed for both A and B. So, when the columns reach the last block, the number at the end of each column will be the $\sqrt[3]{n}$th number after the number in the head of each column, which means, at this point, all the numbers in the list C have been deployed and there are no duplicates. Then, the summation of all blocks is displayed on the output block and is the product of the seed input.