Electronic Colloquium on Computational Complexity, Report No. 186 (2023)

A VLSI Circuit Model Accounting For Wire Delay

Ce Jin* Ryan Williams† Nathaniel Young‡
MIT MIT Unaffiliated

Abstract

Given the need for ever higher performance, and the failure of CPUs to keep providing single-threaded performance gains, engineers are increasingly turning to highly-parallel custom VLSI chips to implement expensive computations. In VLSI design, the gates and wires of a logical circuit are placed on a 2-dimensional chip with a small number of layers. Traditional VLSI models use *gate delay* to measure the time complexity of the chip, ignoring the lengths of wires. However, as technology has advanced, *wire delay* is no longer negligible; it has become an important measure in the design of VLSI chips [Markov, *Nature* (2014)].

Motivated by this situation, we define and study a model for VLSI chips, called *wire-delay VLSI*, which takes wire delay into account, going beyond an earlier model of Chazelle and Monier [*JACM* 1985].

- We prove nearly tight upper bounds and lower bounds (up to logarithmic factors) on the time delay of this chip model for several basic problems. For example, AND, OR and PARITY require $\Theta(n^{1/3})$ delay, while ADDITION and MULTIPLICATION require $\tilde{\Theta}(n^{1/2})$ delay, and TRIANGLE DETECTION on (dense) n-node graphs requires $\tilde{\Theta}(n)$ delay. Interestingly, when we allow input bits to be read twice, the delay for ADDITION can be improved to $\Theta(n^{1/3})$.
- We also show that proving significantly higher lower bounds in our wire-delay VLSI model would imply breakthrough results in circuit lower bounds. Motivated by this barrier, we also study conditional lower bounds on the delay of chips based on the Orthogonal Vectors Hypothesis from fine-grained complexity.

*Partially supported by NSF grants CCF-2129139 and CCF-2127597. This work was also partially supported by the Ethereum Foundation.

†Supported by NSF CCF-2127597 and a Frank Quick Faculty Research Innovation Fellowship. This work was also partially supported by the Ethereum Foundation.

‡This work was partially completed while the author was at U.C. Berkeley.

ISSN 1433-8092

1 Introduction

Very Large Scale Integration (VLSI) technology arose in the 1970s and has become the basis of almost all computing hardware. Given the ubiquitousness of VLSI, it is of interest to study the theoretical limits of VLSI circuits. One of the most important questions is to understand the best-possible execution time that a circuit needs to compute functions of interest. To study this question mathematically, various theoretical models of VLSI were proposed over the years to reflect the physical reality of VLSI circuits: the gates and wires of a logical circuit are typically placed in a nearly-planar way on a chip, each gate takes up some unit of area on the chip, and each gate takes some unit of time to compute its output from its inputs [Tho80, Ull84, Len90]. When measuring the execution time of a circuit, traditional theoretical models of VLSI focused on tradeoffs between the area required for a chip and the execution time (see [BK81] for another notable example). However, their time measures only took *gate delays* into account, and ignored the possible delay that might occur from placing long wires in the chip. This allowed for VLSI designs with time bounds as low as logarithmic in the input size (for example, [MP83] shows that $O(\log n)$ time is possible for n-bit integer multiplication, given sufficiently large chip area). However, as technology has advanced over the decades—shrinking chips to incredibly tiny sizes—the *wire delay* of circuits has become a non-negligible factor in practice [SP10], and such time bounds are no longer realistic. As Markov mentions in his 2014 survey [Mar14],

> "*gate delays were dominant until 2000, but wires get slower relative to gates at each new technology node. [...] Yet, most electrical engineers and computer scientists continue to focus on gates.*"

We develop our model for this new reality by introducing wire delay that is linear in the length of the wire. In real chips, long communications are done through a linear number of constant-length buffered wire segments to avoid introducing a quadratic diffusion delay, and the speed of the very fastest communications along wires is affected by transmission-line delay, which is linear in distance [RCN04, sections 4.4.5 and 9.3].

1.1 Our Results

In this paper, we define a circuit model that is closer to reality, concretely accounting for wire delays as well as clock synchronization issues. The full definition of our model (called *wire-delay VLSI model*) is given in Section 2, but a quick high-level comparison between our model and previous ones is:

- The standard theoretical VLSI circuit model (e.g., [Tho80, Ull84, Len90]) does not account for wire delays at all, and allows input ports to appear anywhere in the chip.
- Chazelle and Monier [CM85] defined a model that does count wire delays (resulting in slower time bounds), but requires all input ports to be placed on the outer border of the chip.
- Our circuit model allows input ports anywhere in the chip, but also accounts for wire delays. This leads to an interesting middle-ground that is more closely aligned with modern VLSI design. (See Remark 2.1 for a discussion on how realistic our model is.)

We initiate a thorough analysis of the power of our wire-delay VLSI model, determining the (asymptotic) minimum delay for computing several fundamental computational problems in our circuit model. We focus on mildly-3D circuits, with a fixed number of layers in the third dimension. (Three-dimensional circuits could theoretically yield smaller wire lengths, but there are significant technological challenges involved in realizing "full 3d" circuits.)

We first state a general lower bound for our wire-delay VLSI model. Informally, a function $f\colon \{0,1\}^n \to \{0,1\}$ is called *non-degenerate* if it depends on all n input bits. Formally, for every $i \in [n]$, there is an $x \in \{0,1\}^n$ such that $f(x) \neq f(x^{(i)})$ where $x^{(i)}$ agrees with x except on the i-th bit.

Theorem 1.1. *For every non-degenerate $f\colon \{0,1\}^n \to \{0,1\}$, computing f on a wire-delay VLSI chip requires $\Omega(n^{1/3})$ time.*

Next, we consider a few simple functions, and show the $n^{1/3}$ lower bound is tight for these functions.

Theorem 1.2. *The AND, OR, and XOR functions can be implemented on a wire-delay VLSI chip in $O(n^{1/3})$ time.*

Already in the VLSI model of [CM85], every non-degenerate function requires $\Omega(\sqrt{n})$ time, due to their restrictions on input placement. Our $O(n^{1/3})$ bounds in Theorem 1.2 give simple examples of functions which can be computed strictly faster when input ports may be placed anywhere in the VLSI chip.

Next we consider the harder (binary) multiplication function, which was previously studied in the VLSI literature using communication complexity. In our new VLSI model, we can prove a lower bound using similar arguments.

Theorem 1.3. *Computing the product of two n-bit integers on a wire-delay VLSI chip requires $\Omega(\sqrt{n})$ time.*

An implementation of the Fast Fourier Transform by Preparata [Pre83] implies a nearly matching $\tilde{O}(\sqrt{n})$ upper bound in our model.[1]

Theorem 1.4. *Computing the product of two n-bit integers can be implemented on a wire-delay VLSI chip in $\tilde{O}(\sqrt{n})$ time.*

Our most interesting results involve the (binary) addition function. In general, addition is much easier than multiplication, and thus is more difficult to prove lower bounds for. In particular, the communication complexity argument in Theorem 1.3 for multiplication does not apply to addition. Nevertheless, we are still able to show that addition has the same lower bound as multiplication. This theorem is proved using a novel argument that trades off memory and serialization.

Theorem 1.5. *Computing the sum of two n-bit integers on a wire-delay VLSI chip requires time $\Omega(\sqrt{n})$.*

We also establish a nearly-matching upper bound.

Theorem 1.6. *Computing the sum of two n-bit integers can be implemented on a wire-delay VLSI chip in $\tilde{O}(\sqrt{n})$ time.*

The proof of the Addition lower bound (Theorem 1.5) crucially requires the property that our VLSI chip is *semellective*; that is, each bit of the input is supplied to exactly one input port to the chip.[2] Interestingly, if we relax this assumption, then Addition can be solved significantly faster.

Theorem 1.7. *Computing the sum of two n-bit integers can be implemented on a wire-delay VLSI chip (where each input bit can be read twice) in time $O(n^{1/3})$, and the problem requires time $\Omega(n^{1/3})$.*

So far, we have seen $\Omega(n^{1/3})$ lower bounds for single-output functions, and $\Omega(n^{1/2})$ lower bounds for functions with n outputs. The following question is natural:

> *What is the strongest time lower bound one can hope to prove in our VLSI model, for explicit functions?*

Here we observe an $\Omega(n^{1/2})$ lower bound for a *single-output function*, proved using standard techniques from worst-case-partition communication complexity by Papadimitriou and Sipser [PS84].

Let $\textsc{Triangle}\colon \{0,1\}^{\binom{n}{2}} \to \{0,1\}$ be the triangle-detection problem on n-node graphs given as adjacency matrices; that is, $\textsc{Triangle}(G) = 1$ if and only if the given graph G contains a triangle.

Theorem 1.8. $\textsc{Triangle}$ *on $m = \binom{n}{2}$ bits requires $\Omega(\sqrt{m})$ time on a wire-delay VLSI chip.*

We also show a nearly-matching upper bound for $\textsc{Triangle}$.

[1] As is standard in theoretical CS, we use $\tilde{O}(f(n))$ to denote $O(f(n) \cdot \operatorname{poly}\log f(n))$.

[2] In contrast, our $\Omega(n^{1/3})$ lower bound in Theorem 1.1 holds even when circuits are **not** semellective: even assuming that multiple copies of the same input bits may be inserted anywhere in the circuit, we still obtain delay lower bounds.

Theorem 1.9. Triangle *on an n-node graph can be solved by a wire-delay VLSI chip of area $\tilde{O}(n^2)$ and delay $\tilde{O}(n)$.*

Therefore, triangle detection in dense graphs also requires $\sqrt{\text{[input length]}}$ time. Can a higher-than-$n^{1/2}$ lower bound be proved for an explicit function on n-bit inputs? We prove that any unconditional lower bound higher than $n^{1/2}(\log n)^c$ (where $c > 0$ is a fixed constant) would actually lead to a breakthrough lower bound in circuit complexity. This is established by the following simulation result, which allows us to simulate any (normal complexity-theoretic) circuit in our wire-delay VLSI chip model.

Theorem 1.10. *There is a $d > 0$ such that the following holds: Let C be a size-s constant fan-in circuit with depth h. Then C can be implemented on a wire-delay VLSI chip in time $O(h \cdot \sqrt{s} \cdot \log^d(s))$.*

A longstanding open question in circuit complexity is to prove a superlinear-size lower bound for an explicit function against $O(\log n)$-depth circuits. By Theorem 1.10, any explicit lower bound higher than $n^{1/2}(\log n)^{d+1}$ would resolve this major open question.

Since unconditional lower bounds much higher than $n^{1/2}$ appear to be currently out of reach, we now consider conditional lower bounds based on a hypothesis from fine-grained complexity (see the survey of [Vas18]). The following Orthogonal Vectors problem is an important problem in fine-grained complexity that is conjectured to be hard:

Orthogonal Vectors (OV)
Given: Sets $A, B \subseteq \{0,1\}^d$, $|A| = |B| = n$
Decide: Is there a $u \in A$ and $v \in B$ such that for all i, $u[i] \cdot v[i] = 0$?

The obvious algorithm for OV tries all possible pairs in $A \times B$ and runs in $O(n^2 d)$ time. For small d, there are faster algorithms: there is a folklore $O(n + 2^d \cdot \mathrm{poly}(d))$ time algorithm, and when $d = c \log n$ for a constant $c \geq 1$, an $n^{2-1/O(\log c)}$ time algorithm [AWY15, CW21]. The OV conjecture states that for sufficiently high dimensions, the OV problem cannot be solved significantly faster than n^2 time. Here we need a non-uniform version of the OV conjecture.

Conjecture 1.11 (Non-Uniform OV Conjecture)**.** *For every $\varepsilon > 0$ there is a $c > 1$ such that OV with $d = c \log n$ cannot be solved in $O(n^{2-\varepsilon})$ time with $O(n^{2-\varepsilon})$ advice bits.*

The OV conjecture is one of the primary hypotheses in fine-grained complexity; it is implied by the Strong Exponential Time Hypothesis (which roughly says that CNF-SAT requires $2^{n-o(n)}$ time) [Wil05]. The same reduction shows that if the Non-Uniform OV Conjecture is false, then CNF-SAT on n-variable $O(n)$-clause formulas has $(2-\varepsilon)^n$-size circuits for some $\varepsilon > 0$, which is believed unlikely. Many other hardness results in fine-grained complexity are reductions from OV (e.g. [BI18, RV13]).

We first give an upper bound for OV in our wire-delay VLSI model which cube-roots the running time of the brute-force algorithm. (This should not be too surprising, given our $n^{1/3}$-time chip design for the OR function in Theorem 1.2.)

Theorem 1.12. *The Orthogonal Vectors problem on n vectors in d dimensions can be solved by a wire-delay VLSI chip of area $\tilde{O}(n^{4/3}d^2)$ and delay $\tilde{O}(n^{2/3}d)$.*

Then, in order to derive a matching conditional lower bound, we need the following result that simulates our wire-delay VLSI model by a serial algorithm with a cubic slowdown.

Theorem 1.13. *Given a VLSI chip for a single-output function of n inputs in t delay, there is a (typical, serial) algorithm for that function running in time $O(t^3)$ and using $O(t^2 + n \log t)$ advice bits.*

Then, we have the following conditional lower bound for OV in our wire-delay VLSI model.

Theorem 1.14. *Assuming the Non-Uniform OV Conjecture, every wire-delay VLSI chip for* Orthogonal Vectors *with n vectors in $\log^2 n$ dimensions requires at least $n^{2/3-o(1)}$ time.*

1.2 Related work

Besides the fundamental work in VLSI theory already cited, there have been several theoretical models of hardware proposed over the decades that are similar (but not identical) to the model we propose. We begin our discussion with the most related model.

Chazelle and Monier's Model. Another VLSI model that accounts for wire delay was proposed and studied by Chazelle and Monier in the early 80s [CM81, CM83, CM85]. For brevity, we will call theirs the CM model. Besides subtle differences in the modeling of wire placements, the main difference between their model and ours is that the CM model assumes the given circuit is drawn on a convex region of the plane, with all input/output nodes on the boundary of this region, while our model allows input/output ports to be placed anywhere in the circuit. Note that if they also did not allow ports to be reused by different inputs, then any function that depends on all n of its inputs would trivially require $\Omega(n)$ time to be computed, simply because the perimeter of the region has to be at least n, so that even propagating all bits from the boundary to the output (wherever it is) would take $\Omega(n)$ time. They still obtain $O(\sqrt{n})$ delay bounds in some cases, because they allow the "reuse" of input ports. We give a few more details on the CM model and what they prove in Appendix A.

Our wire-delay VLSI model can be seen to subsume the CM model: ours is only more powerful. Indeed, our model is already provably faster than the CM model for simple functions such as the AND, OR, and PARITY functions. In particular, the CM model requires time delay $\Theta(\sqrt{n})$ for such functions (see Appendix A for details), whereas our model obtains delay $\Theta(n^{1/3})$ (Theorem 1.2).

Other Related Work. Kravtsov [Kra67] defined a "cellular" circuit model (also studied recently [LZ20]), which is a more restricted version of almost-planar circuits (with crossing number at most 1) in which gates are placed on a 2-dimensional grid, wires only connect to adjacent gates on the grid, and inputs/outputs must appear on the perimeter. This model is weaker than the CM model: after some point in time, all gates and wires in Kravtsov's circuits are fixed to values (similarly to the usual Boolean DAG circuit model), whereas in models such as CM and our own, the values of gates may change over time as inputs propagate through, until the final output.

Fischer [Fis88] showed that, in general, implementing a $T(n)$-step sequential computation (for some natural sequential model) on an unbounded d-dimensional grid requires at least $\Omega(\sqrt[d+1]{T(n)})$ parallel time (accounting for wire delay); this translates to an $\Omega(T(n)^{1/3})$ lower bound for the planar case. In this paper we show higher $\Omega(\sqrt{T(n)})$ lower bounds for several fundamental computational problems in which it is known that $T(n) \leq O(n)$.

Some references covering VLSI theory include [Tho80, Ull84, Len90, Sav98]. These references focus mainly on the trade-off between the *area* (a.k.a. size) of a VLSI circuit and its *time* measured by *gate delay*. These models are also closely related to the planar circuit model, for which the planar graph separator theorem was useful in proving lower bounds [Sav81, LT79, LT80].

VLSI theory has not been developed much since the early 1990s; it seems, given the realistic assumptions of the time, most of the interesting unconditional bounds were proven early. Despite this, there has been some relevant work on VLSI models in the last decade (e.g. [Bla17, Gro15]). Even more recently, Gianinazzi et al. [GBNB$^+$23] studied a 'spatial computer' model which takes communication distance into account in bounds on computation energy. Their model has very different assumptions to VLSI models, including ours, but they provide a 'wire-depth' measure which is similar to our wire delay measure.

We utilize *two-dimensional arrays of circuits* as an intermediate model of computation, which makes it convenient to describe chips in our wire-delay VLSI model. In particular, all of our chip designs are described by circuit arrays, and in Theorem 2.4 we show how circuit arrays can be simulated in our wire-delay VLSI model with low overhead. The practice of building regular arrays has long been common in real VLSI designs ([GBNB$^+$23] cites several modern examples); Kung and Leiserson [KL79] pioneered the use of *systolic* arrays in the theory of VLSI. Other early references include [Lei79, Kun82, BKL83]. Many of the circuit array chips we describe can easily be seen as "systolic" in nature.

2 Formal definition of the wire-delay VLSI model

In this section, we formally define our VLSI model that captures the time delays due to wires in real-world circuits.

Let us stress upfront that no mathematical model can fully capture every aspect of real-world circuits. Modern semiconductor processes have thousands of special constraints and rules which must be followed in chip layout; these are usually all closely-guarded trade secrets. Rather, our model is only intended to formally capture a few major abstract components (our lower bound proofs will solely use these assumptions):

1. Gates and wires are laid out in a mildly 3-dimensional space (with a few 2-dimensional layers).
2. Gates and wires have a minimum size, and cannot overlap within a layer.
3. As a consequence, the number of wires across any boundary in the chip is at most linear in the size of the boundary.
4. Two bits of information cannot occupy the same space at the same time.
5. Information traveling any distance across the chip requires time at least linear in the distance traveled.

Assumptions 1-4 are similar to the assumptions of VLSI models from the 1980s; see e.g., the eight assumptions in the paper by Brent and Kung [BK81]. The most important difference is in Assumption 5, which is missing from the old models but is crucial in our new model, and this explains the name "*wire-delay* VLSI" of our model.

First, we define the various components of our wire-delay VLSI chip model.

- The chip is on an $N \times M \times \ell$ grid, where ℓ is a small fixed constant. We say the chip has ℓ layers and *area* NM.

  A grid point is specified by its integer coordinate (x, y, i) $(1 \le x \le N, 1 \le y \le M, 1 \le i \le \ell)$. (sometimes we also say point (x, y) on the i-th layer). Two grid points $(x, y, i), (x', y', i')$ are *adjacent* if $|x - x'| + |y - y'| + |i - i'| = 1$.

- Some grid points are called *gates*. Gates have the following types: *logic*, *flip-flop*, *input*, or *output*.

- Wires (also called nets) in the chip form tree structures between the gates. A *net* W is an arborescence (an out-tree directed towards the leaves), whose nodes are grid points, and each edge (*a wire segment*) connects two adjacent grid points. The root of the arborescence is a gate (and W is called the *output wire* of this gate), and the leaves of the arborescence are gates (and W is called an *input wire* of these gates). None of the other nodes in the arborescence (called *internal* nodes) are gates. Different nets in the chip must utilize disjoint sets of internal nodes.

Second, we describe how these components behave.

- Every net W transmits one bit of information along the wires of W, at a fixed speed of one unit distance per unit time. In particular, let s be the root gate of a net W, and let t be a leaf gate of W, which is at distance L from s on the tree. After the output of gate s changes to $b \in \{0, 1\}$, it takes exactly L units of time for the input bit (corresponding to W) of gate t to change to b.

- A *logic gate* has k input wires $(1 \le k \le 2)$ and one output wire, and computes a function $f: \{0, 1\}^k \to \{0, 1\}$. Whenever the input bits to the gate changes, the output bit of the gate immediately changes to the return value of f.

- A *flip-flop* has one input wire, and one output wire. We assume there is a globally synchronized clock: at each clock tick, every flip-flop has its output bit changed to the current value of its input bit.

- An *input gate* has one output wire which is used to feed input data to the circuit. An *output gate* has one input wire which is used to get the answer computed by the circuit. Optionally (if the chip designer wants), for any input gate g, we may have an additional input gate g_v on which a 'valid bit' is provided, which will have value 1 if an input is being provided in g_v in the current cycle, and 0 otherwise; this allows the chip to easily ignore the (useless) value from the input gate when no real input is being provided.

Finally, we describe how a wire-delay chip computes a Boolean function $f\colon \{0,1\}^n \to \{0,1\}^m$.

- For each $i \in [n]$, the chip designer assigns the i-th input bit to an input gate g_i and a cycle number t_i.[3] Similarly, for each $j \in [m]$, the chip designer assigns the j-th output bit to an output gate g'_j and some cycle number t'_j. These numbers are fixed as part of the design of the chip; they are not allowed to depend on the input or any part of the chip operation (in the literature, this is called *where- and when-obliviousness*). We require that two inputs (respectively, outputs) cannot appear in an input gate (respectively, output gate) at the same time; formally, for $i, i' \in [n]$ and $i \neq i'$, $(g_i, t_i) \neq (g_{i'}, t_{i'})$, and for $i, i' \in [m]$ and $i \neq i'$, $(g'_i, t'_i) \neq (g'_{i'}, t'_{i'})$.

  On an input $x \in \{0,1\}^n$, the chip computation proceeds by cycles, each taking T_C units of time. At the beginning of cycle k, we feed x_i to input gate g_i for all $i \in [n]$ with $t_i = k$, then immediately generate a synchronized clock signal for all the flip-flops in the chip. At the end of cycle k, we read $f(x)_j$ from the output gate g'_j for all $j \in [m]$ with $t'_j = k$.

- The *total delay* (i.e., time complexity) of the chip is the total amount of time units before reading the last output bit.

- The initial state of the chip (namely, the values stored in the flip-flops) can be arbitrary state (independent from the input string) specified by the chip designer.

In most cases, we assume that the VLSI chips are **semellective**, i.e., each input bit x_i is supplied to only one input port in the chip, at one cycle (and not to multiple ports, which may be arbitrarily far apart from each other, or at multiple cycles, which may be arbitrarily far apart in time). This assumption is needed for our lower bounds on multiplication (Theorems 1.3), addition (Theorem 1.5), and triangle detection (Theorem 1.8), but (interestingly) the assumption is not required for our lower bounds on general non-degenerate functions such as AND, OR, and PARITY (Theorem 1.1). One exception is in Theorem 1.7 (proved in Section 5), where we consider multilective (i.e., non-semellective) VLSI chips for binary addition that read each input bit twice.

Remark 2.1 (On the Realism of Our Model)**.** A few of the assumptions of our model will benefit from some extra explanation as to why they are realistic.

- For convenience, we explicitly model synchronous chips: that is, chips which are controlled by a clock signal, and hold data at flip-flops between clock cycles. This is for convenience in our upper bounds (and is never actually taken as an assumption for lower bounds). Nearly all digital chips are synchronous, but building a synchronous design requires some care to be devoted to how the clock signal is going to be distributed across the chip (called "clock tree synthesis" in VLSI design flows). Since the clock tree does not affect the delay of the chip directly, nor does it dominate the area, we feel comfortable ignoring this in our model, and simply assume that all flip-flops are provided the clock signal correctly.

- The semellectivity assumption is used to ensure that all communication and memory cost is modeled. If the computation requires an input to be read at two different places or times, it should pay the cost to communicate or remember that input. Real chips often request data from off-chip at runtime, but they are only able to do this because they are communicating with a separate memory chip. Under semellectivity, our model counts all the chip area and delay, including that needed for memory and interactions with memory.

[3]Throughout the paper we denote $[n] = \{1, 2, \dots, n\}$.

- Where- and when-obliviousness is justified similarly to semellectivity: marshalling input and output in a data-dependent way would require computation, communication, and memory resources, which we need the chip to pay for so we can model the cost accurately. Savage described it well in his book: "To do otherwise is to assume that an external agent not included in the model is performing computations on behalf of the user" [Sav98].

- The assumption that input can appear anywhere on the chip at any time is easily justified for lower bounds: modern chips and chip packaging technology allow it. In contrast, the CM model assumed that inputs could only appear on the perimeter of the chip (and heavily relied on this assumption) because this was the only way to build and package chips at the time. However, for upper bounds, a question arises: *how does the input arrive to the chip?* It seems like providing input to all parts of the chip might require significant off-chip routing resources, which are not included in the model. However, there are some scenarios in which these resources are not necessary, which we might miss if we made more restrictive assumptions about input:

  1. **Input generated near where it is consumed:** If the input to the computation is generated by the chip itself, or by another chip (a sensor, perhaps) stacked directly on top, then no extra routing resources are needed as long as the input does not have to travel far. One might envision this situation for 3d imaging applications: a sensor (and a computation chip stacked directly beneath it) as large as a cross-section of the volume being imaged can be exposed to the whole volume one slice at a time, and so a volume of size $n \times n \times n$, thus n^3 inputs, can be processed in time n on a chip of area $n \times n$, assuming the computation chip accepts each input where the sensor produces it.
  2. **Stacked memory chips:** although, as we mention, it is not currently possible to stack many computation chips on top of each other, it is possible to stack many memory chips on top of each other. "High-Bandwidth Memory" (HBM) produced this way [JCL+17] is in common use. It is thus physically possible to have many memory chips stacked directly on top of our computation chip, with input from the memory being streamed into the chip all across its surface, according to the way the input is distributed in the memory stack. This requires memory (although not easily-addressable or quickly-writable memory) and vertical routing in 3d, but crucially the computation is still in 2d.

  Since there are so many ways of integrating chips into larger systems, it is very difficult to model a full system of how input gets into the chip, capturing all tradeoffs. We do not attempt to model the full system, and instead assume it is very powerful. This confines some of our upper bounds to only be realistic in situations like the above, but it also means our lower bounds are more general.

2.1 The circuit array model: an intermediate model

We now define a simple "circuit array" model, which we use as an intermediate model of computation to conveniently describe our chip design upper bounds. We will show that any computation described in the circuit array model can be efficiently implemented in our wire-delay VLSI model.

Definition 2.2 (Circuit arrays)**.** A *circuit array* consists of an $N \times N$ grid of cells.[4] Each cell (i, j) consists of a circuit with 3 types of inputs: inputs external to the chip, inputs from other cells in the previous cycle, and local values from the previous cycle, and 3 types of outputs: outputs external to the chip, outputs to other cells for the next cycle, and local values for the next cycle.

All cells will run in parallel, being provided inputs at the beginning of the cycle and producing outputs at the end. Communication on the chip happens between adjacent cells in the grid, in cardinal directions only.

[4]Throughout this paper, we use the convention that on a two-dimensional plane the vector $(0, 1)$ points to the north (drawn upwards in the figures) and the vector $(1, 0)$ points to the east.

We now show how to efficiently implement any circuit array in our wire-delay VLSI model. For convenience, we will work with them to prove upper bounds throughout the paper. We begin with an implementation of a single cell in a circuit array.

Theorem 2.3. *Suppose a single cell of an array can be implemented by a circuit with size s, at most s inputs and outputs, and fan-in bounded by 2. Then the circuit can be implemented on a wire-delay VLSI chip with 5 layers, area $O(s^2)$, and delay $O(s)$. Additionally, if a unique perimeter position (cardinal direction and integer in $[s]$) is given for each input and output, then circuit inputs and outputs can be placed on the perimeter of the chip according to the given positions.*

Proof. We will describe a 'template' for a chip, in which we 'reserve' nodes and edges in the grid for use by nets carrying certain values, even if they are never actually used by any net. Then we will show how to construct the necessary nets using these reserved nodes.

We will use a square grid of size $6s + 2$, and we will index the grid coordinates starting at -1 and ending at $6s$. This will be convenient for reasons which will become apparent later.

The main idea is to topologically sort the circuit, and lay out the nodes on the diagonal in topological order. Gate i will use the square from $(6i, 6i)$ to $(6i+5, 6i+5)$. The output variable and both input variables of the gate will reserve 'full-chip crosses' using layers 0, 1 and 2, centered in this area. The first input will reserve the single node $(6i, 6i)$ on layer 0, the entire horizontal line $(\cdot, 6i)$ on layer 1, and the entire vertical line $(6i, \cdot)$ on layer 2, plus the between-layer edges connecting them at $(6i, 6i)$. Similarly, the second input will reserve a full cross centered on $(6i + 2, 6i + 2)$, and the output will reserve a full cross centered on $(6i + 4, 6i + 4)$. We will implement the gate itself on layer 0 (see figure 1a).

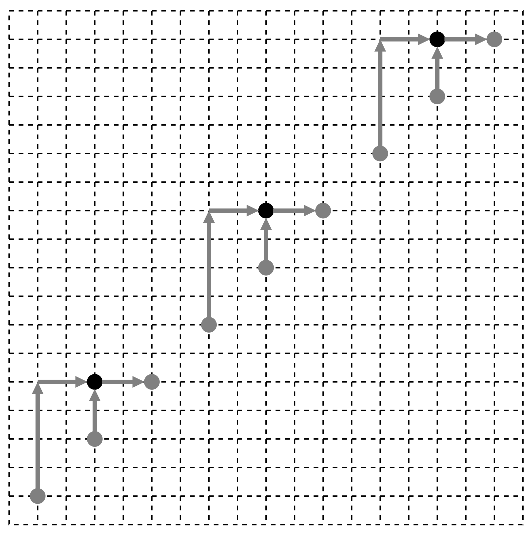
The inputs and outputs to the circuit are placed on the perimeter of the chip, at locations of the form $(-1, 4k)$ or $(6s, 4k)$ on layer 3 for west and east inputs and outputs, or locations of the form $(4k, -1)$ or $(4k, 6s)$ on layer 4 for south and north inputs and outputs (here, k is the integer in $[s]$ used to specify the position).

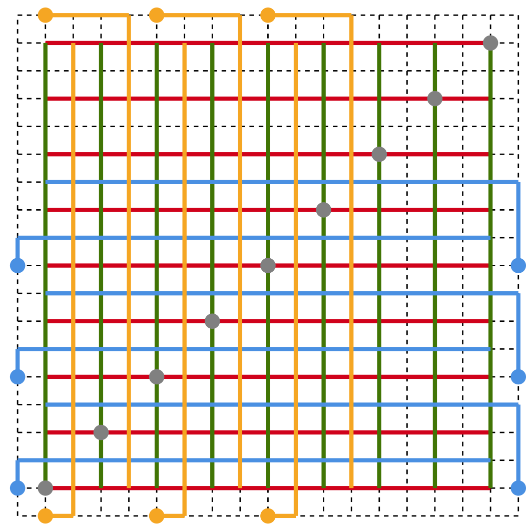
For each west I/O, at $(-1, 4k)$, we will reserve the full horizontal line $(\cdot, 4k + 1)$ from -1 to $6s - 1$, and will connect it to $(-1, 4k)$ using the single edge between them. For each east I/O, we will reserve the full horizontal line $(\cdot, 4k + 3)$ from 0 to $6s$, and will connect it to $(6s, 4k)$ using the length-3 line between them. We will handle north and south I/Os similarly: north I/Os will reserve and connect to the entire vertical line $(4k + 3, \cdot)$; south I/Os, the entire vertical line $(4k + 1, \cdot)$. See figure 1b for a full template illustration.

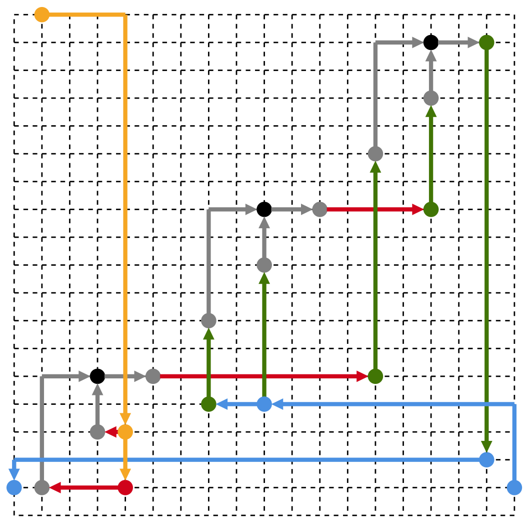
This system of reserved crosses and lines has one very important property: for any two variables in the circuit, there is some position where they overlap; that is, for any variables a and b, there is some $x, y, \ell_1 \geq 1, \ell_2 \geq 1$ where node (x, y) on layer ℓ_1 is reserved to a, and node (x, y) on layer ℓ_2 is reserved to b. Furthermore, it will always be legal to connect ℓ_1 and ℓ_2 using intermediate layers and between-layer edges at (x, y), since they will be the only two out of the upper four layers which are occupied at (x, y): layer 1 will be occupied only if y is even, layer 2 will be occupied only if x is even, layer 3 will be occupied only if y is odd, and layer 4 will be occupied only if x is odd.

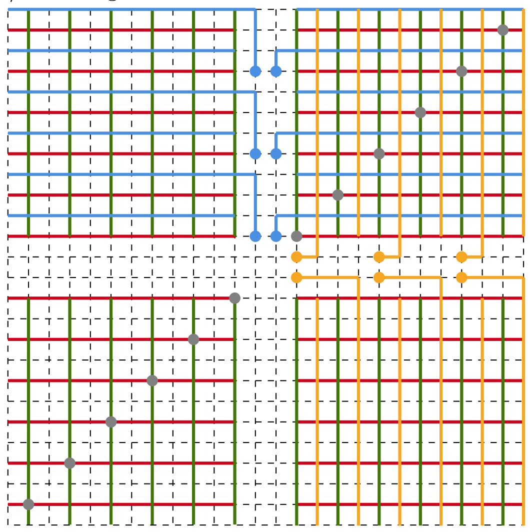
To implement the circuit using this template, we will use these overlap points to connect two variables whenever they are the same values in the circuit. For instance, if the output of gate i is fed into gate j as its second input, we will reserve a connection between layers 1 and 2 at $(6j + 2, 6i + 4)$, allowing a net carrying this value to be drawn from the source at $(6i+4, 6i+4, 0)$ through layers to $(6i+4, 6i+4, 1)$, horizontally to $(6j+2, 6i+4, 1)$, through layers to $(6j+2, 6i+4, 2)$, then vertically to $(6j+2, 6j+2, 2)$, then through layers to $(6j + 2, 6i + 4, 0)$, where the value will be consumed by gate j. Similar nets can be drawn for inputs and outputs; for example, if north input k needs to be fed into gate i as its first input, a net can be drawn from $(4k, 6s, 4)$ to $(4k + 3, 6s, 4)$ to $(4k + 3, 6i, 4)$ to $(4k + 3, 6i, 1)$ to $(6i, 6i, 1)$ to $(6i, 6i, 0)$. An example circuit laid out in this manner is shown in figure 1c.

Clearly this chip takes $O(s^2)$ area, as it is laid out within a bounding box of dimensions $(6s+2) \times (6s+2)$. In order to see that it uses $O(s)$ delay, note first that getting an input from its perimeter location to a gate where it is consumed (or getting an output from the gate where it is produced to its perimeter location) involves a single net of length at most $12s + O(1)$: at most $6s$ steps to get from the perimeter location, along the reserved wire on layer 3 or 4, to the intersection with the cross belonging to the gate, then at most $6s$


(a) Three 2-input, 1-output gates implemented on the diagonal on layer 0 as in theorem 2.3. Black nodes indicate where the gate itself is placed.


(b) Reserved wires on layers 1 through 4, including all possible I/Os, for template in theorem 2.3. Locations of I/Os and gate variables are indicated.


(c) Layout for example circuit as produced by the construction in theorem 2.3. Inputs at north 0 (the 0-th point on the north boundary) and east 0; output at west 0. Gates and nodes with between-layer connections are indicated.


(d) A corner between 4 of the chip templates from Theorem 2.3, showing how inputs and outputs are "matched" in Theorem 2.4

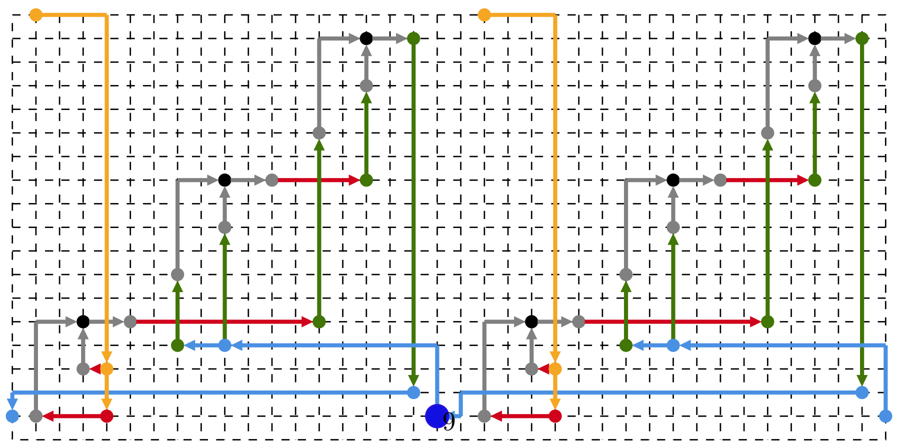

(e) Two example circuits placed next to each other and connected output-to-input as in Theorem 2.4. The large dark blue node indicates where a flip-flop is placed.

Figure 1: Illustrations for Theorem 2.3 and Theorem 2.4

steps to reach the gate, with at most constant extra steps at the beginning and end and to move between layers. Note also that for any path in the circuit beginning and ending at a gate, the sum of lengths of nets implementing it is also at most $12s + O(1)$, because gates are laid out on the diagonal in topological order, and so all values always move west and north, meaning the whole path has only $6s$ steps it can take in each of those two directions. This means the total distance traveled along any path in the computation is at most $36s + O(1)$, which in turn implies at most $36s + O(1)$ delay to complete the computation, from the time the inputs arrive to the time the last output reaches its location on the perimeter. □

Now that we have a way to lay out a circuit describing a single cycle of computation in a square area (a single cell), we can implement a simple grid model of cells. The fact that the I/Os could be assigned locations on the perimeter will be important, as we will be able to abut two chip areas and pass information between them by ensuring their corresponding inputs and outputs line up.

Theorem 2.4. *An $N \times N$ circuit array which takes $\mathcal{C}$ cycles can be implemented in area $A = O(N^2 s^2)$ and total delay $T = O(\mathcal{C}s)$, where s is the size of the largest circuit in the grid.*

Proof. We will use $C_{i,j}$ to refer to the circuit in cell (i, j). We can create an $N \times N$ grid of cells of size $(6s+2)\times(6s+2)$, and use cell (i, j) (placed at $((6s+2)i, (6s+2)j)$) to implement $C_{i,j}$ according to Theorem 2.3. We will place the inputs and outputs along the edges of the cell, ensuring that they "match up" with the corresponding inputs and outputs in the neighboring cell. We will have all cell inputs use flip-flops, and the input to the flip-flop will be the corresponding output in the neighboring cell (see Fig. 1e). The area and delay in the theorem statement can be seen easily: each of the N^2 cells takes area $O(s^2)$, and the delay for a single cycle is $T_C = O(s)$, by Theorem 2.3.

□

3 Basic Lower Bounds in the Wire-Delay Model

In this section we observe a few basic lower bounds for our VLSI model. These, as well as a few other ideas in this paper, are taken and modified from earlier unpublished work by one of the authors as part of his MS thesis at UC Berkeley [You22].

We say a single-output function $f : \{0,1\}^n \to \{0,1\}$ *depends on* the i-th input bit if there exists an $x \in \{0,1\}^n$ such that $f(x) \neq f(x \oplus e_i)$, where $x \oplus e_i \in \{0,1\}^n$ means toggling/flipping the i-th bit of x.

In the following, we assume the number of layers ℓ on the chip is a fixed constant.

Theorem 3.1 ($A \leq O(T^2)$)**.** *If a function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a wire-delay VLSI chip in time T, then f can be computed by a wire-delay VLSI chip in time T and area at most $2T \times 2T$.*

Proof. For notational convenience we prove this in the case where the number of layers is $\ell = 1$; the argument easily generalizes to any $\ell \leq O(1)$.

Let $o = (x_o, y_o)$ denote the location of the output on the chip. For any point $p = (x_p, y_p)$ on the chip, by our assumption (Item 5) that information traveling any distance across the chip requires time at least linear in the distance traveled, p is useful (i.e., p can affect the output value within the chip's run time T) only if its ℓ_1-distance from o in the grid ($\|o - p\|_1 = |x_o - x_p| + |y_o - y_p|$) is at most T. Hence, if we simply ignore all points of the chip outside the $2T \times 2T$ area around the output o, we still get a chip that computes the same function as the original function. □

Theorem 3.2 ($AT \geq \Omega(N)$)**.** *A wire-delay VLSI chip that reads N input bits, with area A and time T, requires $AT \geq \Omega(N)$.*

Proof. This follows immediately from our assumption Item 2 that the gates do not overlap in the same layer; since all N input values require their own constant amount of area and time, and no two overlap, we can sum their area-time requirements together, for $AT \geq N \cdot \Omega(1) = \Omega(N)$. □

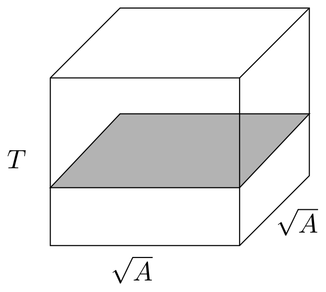


Figure 2: An area-time volume for a square chip, with a single 'time slice' (the chip at a single timestep) indicated.

A more general version of this fact (where N instead refers to the serial time of the computation) also appears in section 12.4 of Savage's book [Sav98]; a similar statement to that appears in our simulation result Theorem 1.13. Figure 2 shows a useful picture to keep in mind: the area of a chip, extended through a third dimension representing time to produce an "area-time volume." With this intuition, Theorem 3.2 states that the area-time volume must be $\Omega(N)$, because each input value requires a constant area-time volume of its own.

Theorem 1.1. *For every non-degenerate $f\colon \{0,1\}^n \to \{0,1\}$, computing f on a wire-delay VLSI chip requires $\Omega(n^{1/3})$ time.*

Proof. Suppose f is computed by a wire-delay VLSI chip in time T. By Theorem 3.1 we assume the chip has area $A \le O(T^2)$. Since f is non-degenerate, all the n input bits must be read by the chip, and hence by Theorem 3.2 we have $AT \ge \Omega(n)$. Combining the two inequalities immediately gives $T \ge \Omega(n^{1/3})$. $\square$

4 Cube-Root Speed-ups for AND, OR, and XOR

In this section, we prove $O(n^{1/3})$ time upper bounds for simple functions such as AND, OR, and XOR in our wire-delay VLSI model claimed in Theorem 1.2, matching the lower bound from Theorem 1.1.

Theorem 1.2. *The AND, OR, and XOR functions can be implemented on a wire-delay VLSI chip in $O(n^{1/3})$ time.*

Proof. In the following we focus on computing XOR. The chip for computing AND and OR follows from essentially the same construction.

The rough idea for proving Theorem 1.2 is as follows: we will place $\Theta(n^{2/3})$ many XOR gates and input gates on a chip of area $\Theta(n^{1/3}) \times \Theta(n^{1/3})$. Each XOR gate computes the XOR function of the input value injected into its cell and the values computed by other gates coming from the left and below. Then this gate sends the computed XOR value to the right and up, towards the upper right corner of the grid which contains the output gate. By repeating $\Theta(n^{1/3})$ cycles, we can inject all $\Theta(n^{1/3}) \times (\Theta(n^{1/3}))^2 = n$ input bits into the chip, and the total XOR value of the input bits are aggregated sent to the output gate at the upper right corner, in time delay proportional to the chip width and length $\Theta(n^{1/3})$.

Now we formally design our chip for computing XOR in the circuit array model (Theorem 2.4), where each cell in the array is implemented by a circuit of $O(1)$ size. In a $k \times k$ circuit array (for some parameter k to be determined later), we designate (k,k) as the output cell. Every cell (i,j) (except (k,k)) is assigned an *out-neighbor* cell, defined as follows (see Fig. 3):

- If $i \neq k$, then the out-neighbor of (i,j) is $(i+1,j)$.
- If $i = k$ and $j \neq k$, then the out-neighbor of (i,j) is $(i,j+1)$.

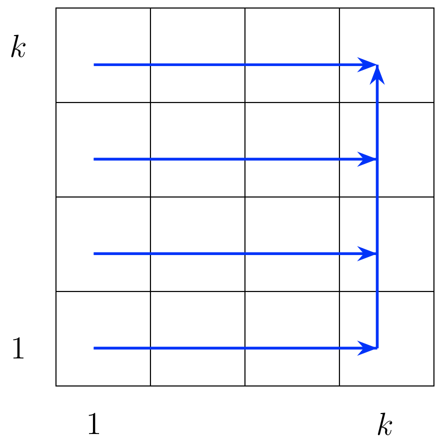


Figure 3: The $k \times k$ grid for the AND/OR/XOR chip. Arrows indicate the direction to send the computed values.

If cell y is an out-neighbor of a cell x, we say x is an *in-neighbor* of y. Note that this out-neighbor relation forms an in-tree rooted at (k, k).

Each of the $k \times k$ cells also receives one input bit (external to the chip) in each cycle. Now we specify the behavior of each cell (i, j) in each cycle.

1. At the beginning of a cycle, cell (i, j) computes the XOR value of all bits sent from the in-neighbors of (i, j) and the external input bit sent to this cell; denote this XOR value by x. Let v denote the value stored by cell (i, j) (computed in the previous cycle).

2. At the end of this cycle, the behavior depends on whether $(i, j) = (k, k)$:

   - If $(i, j) \neq (k, k)$: cell (i, j) sends v to its out-neighbor cell. Then the value stored by cell (i, j) is modified to x.
   - If $(i, j) = (k, k)$: the value stored by this cell is modified to $v \oplus x$ (where $\oplus$ means XOR).

Since in each cycle we can inject $k \times k = k^2$ input bits to the chip, we can use $\lceil n/k^2 \rceil$ cycles to inject all n input bits to the chip. (After that, the cells simply receive zeros instead of actual external input bits.) Then, we run the chip for another $2k$ cycles, and let cell (k, k) output its stored value. We now show that this value is the correct XOR value of all the n input bits.

Let $v_{i,j}^{(t)}$ denote the value stored by cell (i, j) at the end of the t-th cycle. Let $x_{i,j}^{(t)}$ denote the x-value computed by cell (i, j) during the t-th cycle (defined above), and let $y_{i,j}^{(t)}$ denote the external input bit injected to cell (i, j) during the t-th cycle. Let $V^{(t)} = \bigoplus_{i,j} v_{i,j}^{(t)}$ denote the total XOR value of the stored values of all $k \times k$ cells at the end of the t-th cycle. Initially, $v_{i,j}^{(0)} = 0$. Now observe that $V^{(t)} \oplus V^{(t-1)}$ equals the total

XOR value of the external input bits injected to the chip during cycle t. This is because

$$\begin{aligned}
V^{(t)} &= \Big(\bigoplus_{(i,j)\neq(k,k)} v_{i,j}^{(t)}\Big) \oplus v_{k,k}^{(t)} \\
&= \Big(\bigoplus_{(i,j)\neq(k,k)} x_{i,j}^{(t)}\Big) \oplus (v_{k,k}^{(t-1)} \oplus x_{k,k}^{(t)}) && \text{(by step 2)} \\
&= \Big(\bigoplus_{(i,j)} x_{i,j}^{(t)}\Big) \oplus v_{k,k}^{(t-1)} \\
&= \Big(\bigoplus_{(i,j)} \Big(y_{i,j}^{(t)} \oplus \bigoplus_{(i',j')\text{:in-neighbor of }(i,j)} v_{i',j'}^{(t-1)}\Big)\Big) \oplus v_{k,k}^{(t-1)} && \text{(by step 1)} \\
&= \Big(\bigoplus_{(i,j)} y_{i,j}^{(t)}\Big) \oplus \Big(\bigoplus_{(i,j)} v_{i,j}^{(t-1)}\Big),
\end{aligned}$$

and hence $V^{(t)} \oplus V^{(t-1)} = \bigoplus_{(i,j)} y_{i,j}^{(t)}$. Consequently, for all $t \geq \lceil n/k^2 \rceil$, $V^{(t)}$ equals the XOR of all the n input bits.

Now we show that after another $2k$ cycles, all cells except (k,k) will have stored values equal to 0; this would imply (k,k) stores the value $V^{(t)}$ which is the correct answer. This can be done by a simple induction: after one cycle, the stored values in all cells $(1,j)$ become zero (because they have no in-neighbors, and all external input bits are zero). After another cycle, the stored values in all cells $(2,j)$ also become zero, since the only in-neighbors they have are $(1,j)$, which only send zeros from the previous cycle. Since the longest chain in the tree $((1,1) \to (2,1) \to \cdots \to (k,1) \to (k,2) \to \cdots \to (k,k))$ has length $2k-1$, we know after $2k$ cycles all cells except (k,k) store zero values.

Hence, we have shown that the chip produces the correct answer at (k,k) after $\lceil n/k^2 \rceil + 2k$ cycles in total. Setting $k = \Theta(n^{1/3})$, the total delay is only $O(n^{1/3})$. ☐

For $\ell \gg 1$ layers, one can obtain an improved delay bound by directly extending the construction to multiple layers. One can show that the delay is $O((n/\ell)^{1/3})$. However, we note that the circuit model becomes less realistic with multiple layers: how might one "inject" new inputs into ports in the middle of the circuit, when there are several layers of such ports?

5 Results on Addition

5.1 Lower Bound for Addition

In the addition problem, we are given two n-bit input binary numbers A, B, and want to output their sum C in binary. We use $A[i]$ to denote the i-th bit of A, where $A[0]$ is the least significant bit of A.

In the following we prove the lower bound for semellective chips computing addition, restated below.

Theorem 1.5. *Computing the sum of two n-bit integers on a wire-delay VLSI chip requires time $\Omega(\sqrt{n})$.*

First we have the following simple observation similar to Theorem 3.1.

Lemma 5.1. *If a chip which computes addition has delay T, then all its input bits and output bits are confined to an area of $4T \times 4T$ on the chip.*

Proof. Since the most significant output bit $C[n]$ depends on all the inputs $A[0], \ldots A[n-1]$ and $B[0], \ldots, B[n-1]$, these inputs are confined to an area at most $2T \times 2T$ (i.e., radius T) around the point on the chip at which $C[n]$ is produced. Then, since every output bit $C[i]$ depends on at least one input bit (for example, $A[i]$), we know no output can be more than T distance beyond that $2T \times 2T$ area. Hence all inputs and outputs fit in a total area of $4T \times 4T$. See Fig. 4 for an illustration of this argument. ☐

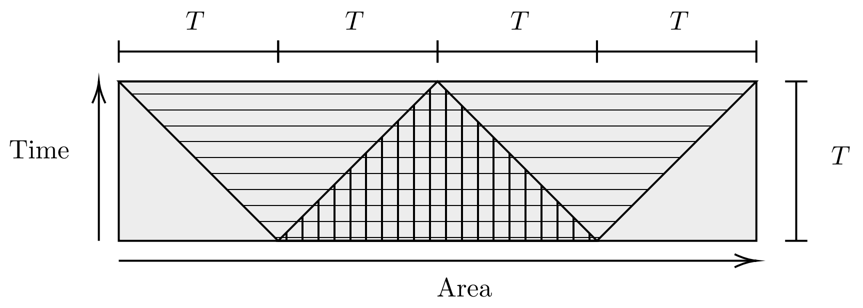


Figure 4: Side view of $4T \times 4T \times T$ volume for Theorem 1.5. Vertical lines indicate volume where inputs may be found, horizontal lines indicate where outputs may be found, shaded region is rectangle for convenience.

Lemma 5.1 means all our inputs and outputs of our delay-T addition chip should fit in a total area-time volume of $4T \times 4T \times T$. We will use this fact to show the contrapositive of Theorem 1.5 (i.e., that delay T is only sufficient to solve addition problems of size $O(T^2)$) in the following lemma.

Lemma 5.2. *Consider any delay-T_0 addition chip $\mathcal{C}$ with ℓ layers (with inputs $A[0..n-1], B[0..n-1]$ and outputs $C[0..n]$), which fit in a total area-time volume of $4T_0 \times 4T_0 \times T_0$ by Lemma 5.1.*

There is a universal constant c such that the following holds for all $1 \leq T \leq T_0$: for any chip subvolume of area $4T \times 4T$ with ℓ layers and time interval T in chip $\mathcal{C}$, there are at most $c\ell T^2$ indices $i \in \{0, 1, \ldots, n-1\}$ for which $A[i]$ is read, and $C[i]$ is produced, on that $4T \times 4T$ section of the chip within that T time interval.

In particular, by taking $T := T_0$ and considering the entire n-bit addition chip $\mathcal{C}$ of delay T_0 in Lemma 5.2, we conclude $n \leq c\ell T_0^2$, which proves Theorem 1.5.

We emphasize that Lemma 5.2 is stated more generally for any chip *subvolume* and a subset of the problem input and output; the area and delay in question need not be the entire area and delay of the chip. This will be important for the proof, as it uses induction on the size of the subvolume.

Proof. Let I denote the set of indices i under consideration, namely the indices $0 \leq i < n$ for which $A[i]$ is read and $C[i]$ is produced on that $4T \times 4T$ section of the chip within that T time interval. Let $m := |I|$ be the quantity we want to bound.

We use induction on T. As a base case, we use the following trivial bound: the number of inputs that can be read by the $4T \times 4T$ section of the chip within the T time interval is at most $4T \times 4T \times T \times \ell = 16\ell T^3 \leq c\ell T^2$ as desired for all $T \leq c/16$. Hence, in the following we assume $T > c/16$.

At a high level, the basic idea of our proof is to show a "memory or serialization" trade-off:

> Either the chip must "remember" inputs (or a similar amount of data) for a long time, thus increasing the area and therefore wire delay, or the chip must wait a long time for some outputs to be finished before reading more inputs, thus reducing parallelism and increasing delay.

Formally, let $1 \leq \tau < T/3$ be some time increment parameter to be determined later, and define $S \subseteq I$ to be the set of indices i such that at most τ time passes between the time $A[i]$ is read and the time $C[i]$ is produced. We consider two cases depending on whether $|S| \leq m/2$:

- **"Memory" case:** $|S| \leq m/2$.

  For each timestep t in $[0, T-1]$, the set of 'straddling indices' for timestep t, denoted by $I_t \subseteq I$, is defined as the set of indices $i \in I$ such that $A[i]$ is read at or before time t, but $C[i]$ is produced after time t. We claim that $|I_t| \leq 6T \times 6T \times \ell$ must hold for all t. The proof of this claim follows from a communication complexity argument: in the addition instance, we set $B[i] = 0$ for all i and $A[i] = 0$ for all $i \notin I$, leaving $A[i]$ for all $i \in I$ unfixed. Then, note that a snapshot of the state of the chip at timestep t (within a radius of $4T + 2t \leq 6T$)contains all the information needed to reproduce the

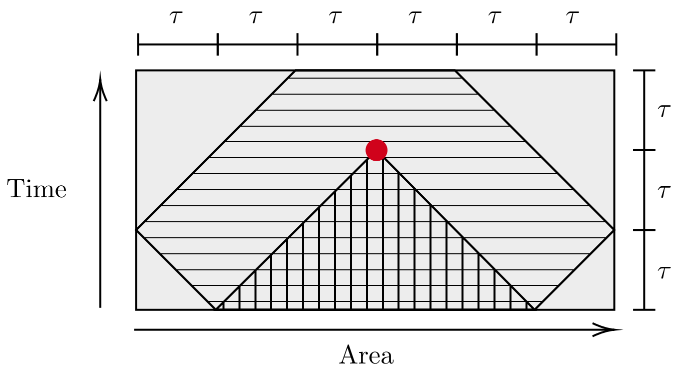


Figure 5: Side view of $6\tau \times 6\tau \times 3\tau$ volume for the serialization case of Lemma 5.2. The red point is where $C[j]$ is produced; vertical lines indicate the volume where inputs $A[i]$ may be read, and horizontal lines indicate the volume where outputs $C[i]$ may be produced, for $i \in S'$.

output $C[i]$ (which equals the input $A[i]$) for all straddling indices $i \in I_t$. So the number of bits stored in the snapshot, $6T \times 6T \times \ell$, must be at least $|I_t|$.

By assumption of $|S| \leq m/2$, we have at least $|I| - |S| > m/2$ indices which are straddling for at least τ distinct timesteps, and thus $\sum_{0 \leq t \leq T-1} |I_t| > \tau m/2$. By averaging, there is a timestep t with $|I_t| \geq \frac{m\tau}{2T}$. Then, the claim from the previous paragraph implies $\frac{m\tau}{2T} \leq 36\ell T^2$, or $m \leq 72\ell T^3/\tau$.

- **"Serialization" case:** $|S| \geq m/2$.

  In this case we will use the inductive hypothesis. Let j be the largest (highest-order) index in S, and define $S' \subseteq S$ to be the set of indices $i \in S$ such that $A[i]$ is read at most 2τ timesteps before $C[j]$ is produced.

  Note that for such $i \in S'$, $C[j]$ depends on $A[i]$ (due to $i \leq j$), so $A[i]$ must be read within radius 2τ of where $C[j]$ is produced. Furthermore, the respective output $C[i]$ is produced at most τ time after $A[i]$ is read by definition of S, and is therefore produced at most τ distance away from where $A[i]$ is read. Hence, the inputs $A[i]$ and outputs $C[i]$ for all $i \in S'$ must fit in a $6\tau \times 6\tau \times 3\tau$ area-time volume (see Fig. 5).

  By the inductive hypothesis with parameter 3τ (this is why we needed $\tau < T/3$), this means only $9c\ell\tau^2$ such input-output pairs exist, i.e., $|S'| \leq 9c\ell\tau^2$, and $|S \setminus S'| \geq |S| - 9c\ell\tau^2$. Note that for all $k \in S \setminus S'$, $A[k]$ is read at least 2τ timesteps before $C[j]$ is produced (by definition of S'), and $C[k]$ is produced at most τ after $A[k]$ is read (by definition of S), so $C[k]$ is produced at least τ before $C[j]$ is produced.

  Repeating the argument with $S \leftarrow S \setminus S'$, we can find a $C[k']$ for some $k' \in S \setminus S'$ which is produced at least τ time before $C[k]$ is produced (and thus at least 2τ time before $C[j]$ is produced), and so on. We iterate this argument for at least $\lfloor \frac{|S|}{9c\ell\tau^2} \rfloor \geq \lfloor \frac{m}{18c\ell\tau^2} \rfloor$ iterations until S becomes empty, and in this way we conclude that $C[j]$ must be produced at time at least $T \geq \tau \lfloor \frac{m}{18c\ell\tau^2} \rfloor \geq \tau \left(\frac{m}{18c\ell\tau^2} - 1 \right) = \frac{m}{18c\ell\tau} - \tau$ since the very beginning, that is, $m \leq 18c\ell\tau(T + \tau) < 36c\ell\tau T$.

Summarizing the two cases, we must have the upper bound

$$m \leq 72\ell T^3/\tau + 36c\ell\tau T = c\ell T^2 \cdot 36 \left(\frac{2T}{c\tau} + \frac{\tau}{T} \right).$$

Recall that $T > c/16$. We set parameter $\tau := \lceil T/\sqrt{c} \rceil \geq 1$, so the upper bound becomes

$$m \leq c\ell T^2 \cdot 36 \left(\frac{2}{\sqrt{c}} + \frac{T/\sqrt{c} + 1}{T} \right) \leq c\ell T^2 \cdot 36 \left(\frac{3}{\sqrt{c}} + \frac{16}{c} \right) < c\ell T^2$$

as desired, by setting the constant c to be large enough. Note that τ satisfies the requirement of $\tau < T/3$ for large enough c.

$\square$

Note that our lower bound matches the following upper bound (up to logarithmic factors).

Theorem 1.6. *Computing the sum of two n-bit integers can be implemented on a wire-delay VLSI chip in $\tilde{O}(\sqrt{n})$ time.*

Proof. The carry-lookahead adder [Ros60] is a Boolean circuit of depth $O(\log n)$ and size $O(n)$ that computes the addition of two n-bit integers. By our simulation result in Theorem 1.10 (to be proved in Section 8), this implies a wire-delay VLSI chip for adding two n-bit integers in $\tilde{O}(\sqrt{n})$ time. $\square$

5.2 Two-Pass Addition With Less Delay

Now we show a VLSI chip reading each input bit twice can solve Addition significantly faster than the above lower bound, proving the upper bound part of Theorem 1.7 (the $\Omega(n^{1/3})$ lower bound claimed in Theorem 1.7 easily follows from the proof of Theorem 1.1).

Theorem 5.3. *There is a wire-delay VLSI chip of area $O(n^{2/3})$ and delay $O(n^{1/3})$ which computes the addition of two n-bit binary integers A and B, reading input bits twice.*

Proof. We assume that n is a perfect cube. For convenience, we will sometimes view our input numbers $A[0\dots n-1], B[0\dots n-1]$ and output number $C[0\dots n]$ in 3 dimensions: define $A[i,j,k] = A[n^{2/3}i+n^{1/3}j+k]$, and similar for B and C.

We will define a circuit array with a $n^{1/3} \times n^{1/3}$ grid of cells, and use Theorem 2.4. We will view our array as operating in 4 distinct "phases." Phases 1 through 3 will be responsible for computing a set of carry values as setup for phase 4, which will produce the actual result. These phases are part of the analysis and the input/output schedule only; all circuitry on the chip will be running in every cycle, but the values produced will only be correct once enough time has elapsed, and we will design the input/output schedule accordingly.

In the first phase, each cell (i,j) for $i \in [0, n^{1/3}-1]$, $j \in [0, n^{1/3}-1]$ independently accumulates "carry generate" (G) and "carry propagate" (P) values for its own subset of $n^{1/3}$ indices in the addition. In particular, at timestep k, cell (i,j) takes in $A[i,j-1,k]$ and $B[i,j-1,k]$ as inputs to the chip, along with G_{ij} and P_{ij}, which are remembered from the previous timestep (or initialized to 0 and 1 respectively at the first timestep). Then it computes:

$$G_{ij} \leftarrow (A[i,j-1,k] \wedge B[i,j-1,k]) \vee \big(G_{ij} \wedge (A[i,j-1,k] \vee B[i,j-1,k])\big)$$

and

$$P_{ij} \leftarrow P_{ij} \wedge (A[i,j-1,k] \vee B[i,j-1,k]).$$

At the end of phase 1 (after $n^{1/3}$ cycles), the output G_{ij} and P_{ij} together give us information about the carry behavior of the addition in the index range assigned to cell (i,j). P_{ij} is "carry propagate": if the carry-in bit at index $n^{2/3}i + n^{1/3}(j-1)$ is 1, will the carry-out bit at index $n^{2/3}i + n^{1/3}(j+1) - 1$ also be 1? G_{ij} is "carry generate": will the carry-out bit be 1 even if the carry-in is 0? This notion of "generate" and "propagate" is the same one used in practical fast adders such as "carry-lookahead" adders [Ros60]. Note that $j-1$ is sometimes negative, since $j = 0$ for the west column of the grid. This means that cell $(0,0)$ has no range of input values assigned to it; it remains idle in phase 1, and produces $G_{0,0} = P_{0,0} = 0$.

Phase 2 involves 'collecting' these generate and propagate values along rows. Each cell will take two inputs from the west: $G'_{i,j-1}$ and $P'_{i,j-1}$. The cell will produce two matching outputs to the east: at the output,

$$G'_{ij} \leftarrow G_{ij} \vee (G'_{i,j-1} \wedge P_{ij}) \text{ and } P'_{ij} \leftarrow P'_{i,j-1} \wedge P_{ij}.$$

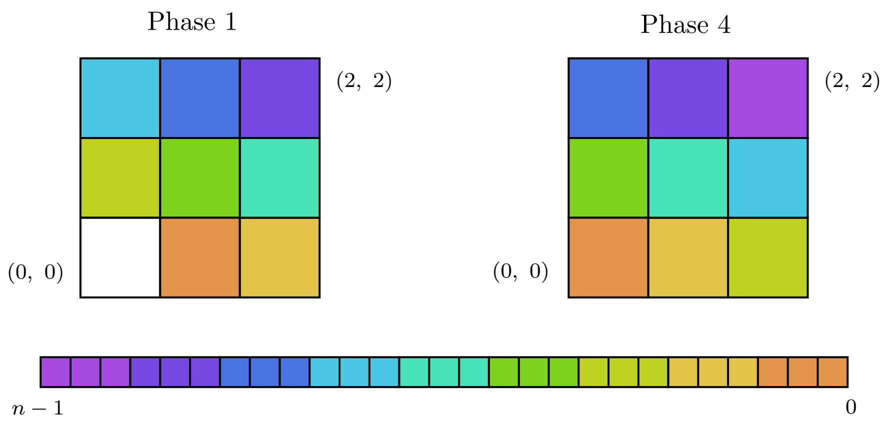


Figure 6: Mapping of input/output indices to cells for phases 1 and 4 ($n = 27$)

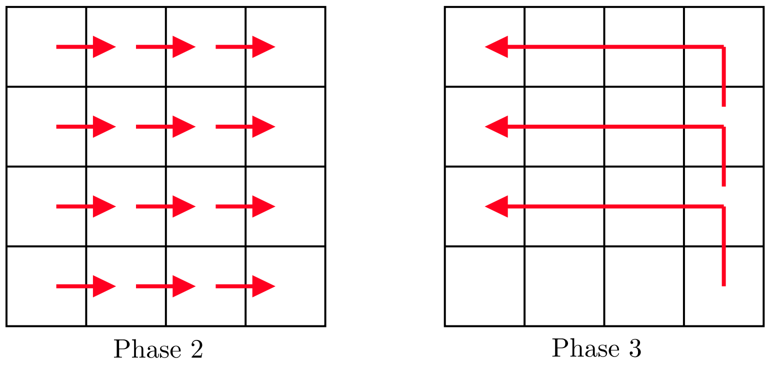


Figure 7: Communication patterns of G' and P' values for phase 2, and c values for phase 3 ($n = 64$)

The westernmost cells $(i, 0)$ will simply produce P_{i0} as output P'_{ij}, and G_{i0} as output G'_{ij}. Each cell will also remember the output G'_{ij} and P'_{ij}. After $n^{1/3}$ timesteps of this phase, generate and propagate values have been communicated across the entire length of each row. G'_{ij} now encodes whether a carry will be generated by the index range $n^{2/3}i$ to $n^{2/3}i + n^{1/3}(j + 1) - 1$, and P'_{ij} encodes whether a carry would be propagated through that same index range. We consider phase 2 to have ended once enough time has elapsed that these values are correct in every cell, which is $n^{1/3}$ cycles after the end of phase 1.

Phase 3 involves collecting the new generate values across rows. Each cell (i, j) needs to know the carry-in bit at index $n^{2/3}i + n^{1/3}j$. Note that every cell in the southernmost row $i = 0$ already knows this bit: G'_{0j} is whether a carry will be generated in the range 0 to $n^{1/3}j - 1$, which since the lower end of the range is 0 is exactly whether there will be a carry out from $n^{1/3}j - 1$ (and thus into $n^{1/3}j$). So for these cells, we can remember the true carry-in value as $c_{n^{1/3}j} = G'_{0j}$. Each cell (i, j) not in the southernmost row or the easternmost column will take one input from the east, $c_{n^{2/3}i-1}$, and replicate it as an output to the west. Since this value encodes whether a carry appears at $n^{2/3}i$, which is the lower end of the range covered by G'_{ij} and P'_{ij}, we can combine them to produce

$$c_{n^{2/3}i+n^{1/3}j-1} = G'_{ij} \vee \left(c_{n^{2/3}i-1} \wedge P'_{ij}\right)$$

which is the necessary carry-in value for this cell. The $c_{n^{2/3}i-1}$ values come from the easternmost column – each cell $(i, n^{1/3} - 1)$ in the easternmost column will take $c_{n^{2/3}i-1}$ as input from the south, and replicate it as output to the west, while producing its necessary carry-in value

$$c_{n^{2/3}(i+1)-1} = G'_{i(n^{1/3}-1)} \vee c_{n^{2/3}i-1} \wedge P'_{i(n^{1/3}-1)}$$

to be remembered for the next phase and as an output to the north. This phase will take $2n^{1/3}$ timesteps ($n^{1/3}$ timesteps to propagate values up the easternmost column, and another $n^{1/3}$ to finish propagating west along the northernmost row), after which every cell (i, j) will have computed its carry-in value.

In phase 4, we will bring our inputs in again and produce outputs as we do so, using the computed carry values. At timestep k in this phase, cell (i, j) takes in $A[i, j, k]$ and $B[i, j, k]$ as inputs to the chip, along with c_{ij}, remembered from the previous timestep (or initialized to the $c_{n^{2/3}i+n^{1/3}j-1}$ value computed in phase 3). Then it does a full-adder operation: it computes $C[i, j, k] = c_{ij} \oplus A[i, j, k] \oplus B[i, j, k]$ and $c_{ij} \leftarrow \mathrm{MAJ}(c_{ij}, A[i, j, k], B[i, j, k])$, where $\oplus$ is exclusive-or and MAJ is majority. The last cell, $(n^{1/3} - 1, n^{1/3} - 1)$, should also output its final carry-out value as $C[n]$. □

6 Lower Bound for Multiplication

We now turn to proving an $\Omega(\sqrt{n})$ delay lower bound for multiplying two n-bit binary integers. The lower bound follows from communication complexity arguments, which were also used in old VLSI literature.

Denote the input bits of the circuit by $a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$, and the output bits by $c_0, c_1, \ldots, c_{2n-1}$ (where a_0, b_0, c_0 are the least significant bits). First we make the following observation.

Recall that we say an output bit o *depends on* an input bit i if there exists an input $x = (a, b)$ where flipping the input bit i would change the output bit o.

Proposition 6.1. *For $0 \le i \le n-1$, c_i depends on the input bits $a_0, a_1, \ldots, a_i, b_0, b_1, \ldots, b_i$.*

For $n \le i \le 2n-2$, c_i depends on all the input bits.

Proof. The first statement immediately follows from considering the input instance where $a_j = b_{i-j} = 1$, while all other input bits are zeros.

It remains to verify the second statement for any output bit c_i (where $n \le i \le 2n-2$) and input bit a_j (the case for b_j is similar). If $j \ge i - (n-1)$, then we can use the same argument as in the first proof. Hence we assume $0 \le j \le i - n$.

Consider the input instance $a = 2^n - 1, b = 2^{n-1} + 1$, i.e., let $a_0 = a_1 = a_2 = \cdots = a_{n-1} = 1$, and let $b_0 = b_{n-1} = 1, b_1 = b_2 = \cdots = b_{n-2} = 0$. We have $a \cdot b = 2^{2n-1} + 2^{n-1} - 1$, so in the binary expansion of this product we have $c_{n-1} = c_n = \cdots = c_{2n-2} = 0$. In particular, $c_i = 0$.

Now we flip the input bit a_j, and have $a' = 2^n - 2^j - 1$. Then,

$$\begin{aligned} a' \cdot b &= 2^{2n-1} - 2^{n-1+j} - 2^{n-1} + 2^n - 2^j - 1 \\ &= (2^{n-j} - 1) \cdot 2^{n-1+j} + (2^{n-1} - 2^j - 1) \\ &\in \big[(2^{n-j} - 1) \cdot 2^{n-1+j}, 2^{n-j} \cdot 2^{n-1+j}\big). \end{aligned}$$

So in the binary expansion of $c' = a' \cdot b$ we have $c'_{n-1+j} = c'_{n-1+j+1} = \cdots = c'_{2n-2} = 1$. In particular, $c'_i = 1$. This shows that the output bit c_i depends on the input bit a_j. □

It is clear that for such a pair of i, o, the output bit must be printed *after* reading the input bit, and the output port of o must be within T distance from the input port of i.

Theorem 1.3. *Computing the product of two n-bit integers on a wire-delay VLSI chip requires $\Omega(\sqrt{n})$ time.*

Proof. Recall that we assume the chip is semellective: each input bit only appears in one of the input ports at one moment in time during the computation.

In the following, we will focus on the set $O = \{c_{n-1}, c_n, \ldots, c_{2n-2}\}$ of output bits (from the "higher-order half" of output bits). We have $|O| = n$, and from Proposition 6.1 we know that every output bit in O is a non-degenerate function: it depends on all input bits. Let the delay of the chip be T. By a similar argument to Theorem 3.1 and Lemma 5.1, we may assume the area of the chip is $O(T) \times O(T)$.

Now we consider at which time steps the chip reads the input bits and prints the output bits. There must be a particular time step t^* such that, *all* the input bits are loaded by the end of time step t^*,

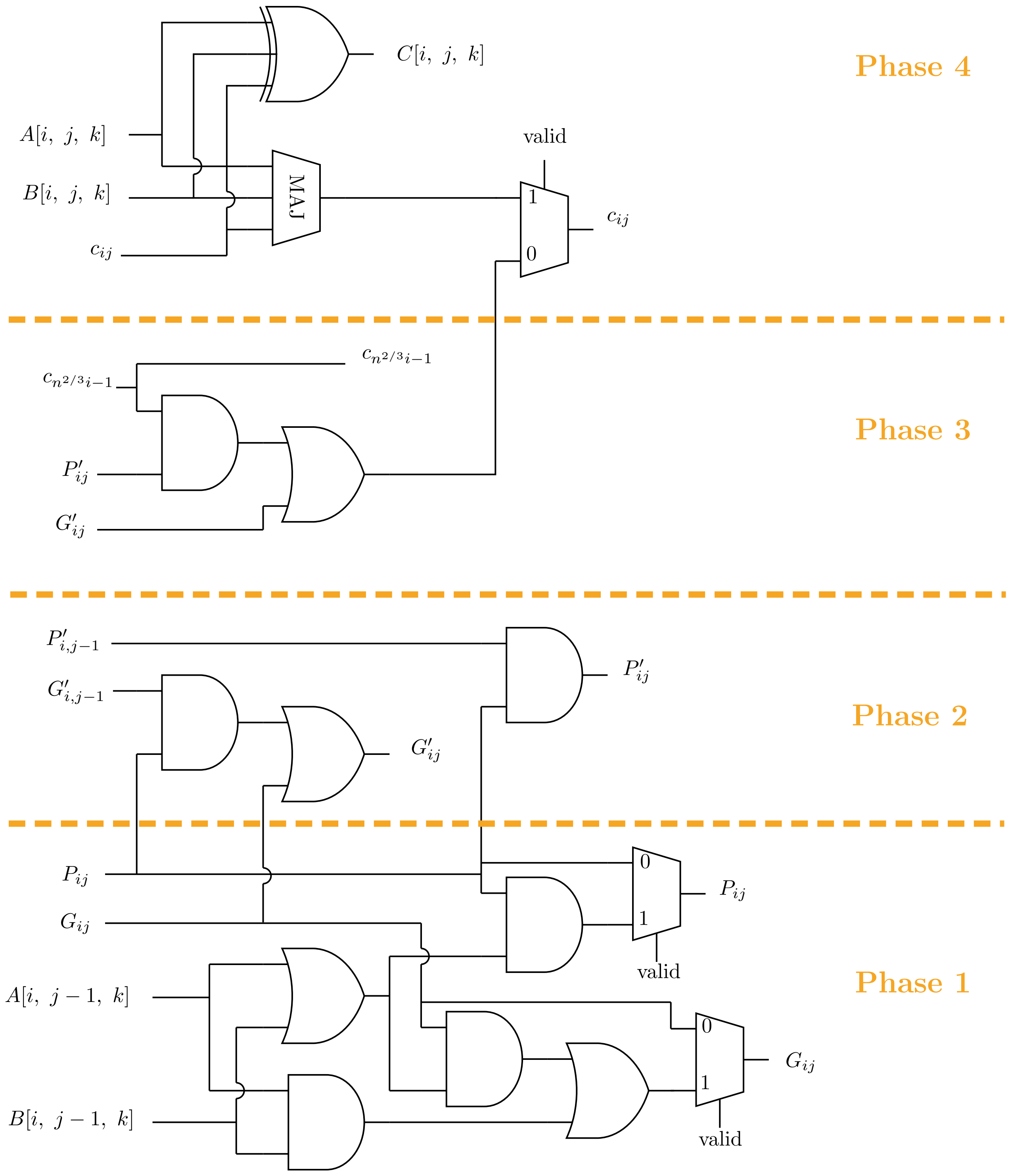


Figure 8: The full circuit for a cell (not on the edge of the array) in Theorem 5.3. Flip-flops to remember values for the circuit are implicit whenever an input and output are identical (except for $c_{n^{2/3}i-1}$, which is an input on the east and output on the west). "Valid" inputs refer to the validity of A (or equivalently, B) inputs in their phase.

and all the output bits in O are printed after time step t^*. Observe that, by setting $a = 2^{n-1}$, we have $(c_{n-1}, c_n, \ldots, c_{2n-2}) = (b_0, b_1, \ldots, b_{n-1})$.

Hence, we can obtain a one-way communication protocol for transmitting an n-bit string, using only $O(T^2)$ bits of communication: Alice sets $a = 2^{n-1}$, lets b encode the n-bit string, and simulates the chip computation up to time step t^*, and sends the current configuration of the chip to Bob. Then Bob continues simulating and obtains the output bits $c_{n-1}, \ldots, c_{2n-2}$ which are identical to the string Alice wants to send. By a basic information theoretic argument, we must have $n \le O(T^2)$, which finishes the proof. □

Note that this lower bound matches the following upper bound (up to logarithmic factors).

Theorem 1.4. *Computing the product of two n-bit integers can be implemented on a wire-delay VLSI chip in $\tilde{O}(\sqrt{n})$ time.*

Proof. The well-known Schönhage-Strassen algorithm [SS71] for integer multiplication implies that there is a Boolean circuit for multiplying two n-bit integers of size $\tilde{O}(n)$ and depth $(\log n)^{O(1)}$. By our simulation result Theorem 1.10 (to be proved in Section 8), this implies a wire-delay VLSI chip for multiplying two n-bit integers in $\tilde{O}(\sqrt{n})$ time. Note that a more direct construction also follows from Preparata's VLSI design for binary multiplication [Pre83]. □

7 Triangle

Let TRIANGLE: $\{0,1\}^{\binom{n}{2}} \to \{0,1\}$ be the triangle-detection problem on n-node graphs given as adjacency matrices; that is, TRIANGLE$(G) = 1$ if and only if the given graph G contains a triangle.

Theorem 1.8. TRIANGLE *on $m = \binom{n}{2}$ bits requires $\Omega(\sqrt{m})$ time on a wire-delay VLSI chip.*

Proof. Let $m = \binom{n}{2}$. Papadimitriou and Sipser [PS84] showed a worst-case-partition communication complexity lower bound for TRIANGLE: for any bipartition of the m input bits into two equal-size sets A, B distributed to Alice and Bob respectively, Alice and Bob need to communicate at least $\Omega(n^2)$ bits to decide the TRIANGLE problem deterministically.

Suppose we have a wire-delay VLSI chip solving TRIANGLE in T time. First note that TRIANGLE is a non-degenerate function (the output bit depends on all the input bits), so the chip can only output the answer only after all the input bits are read. By a similar argument as in Theorem 3.1, we can assume that the chip has area at most $2T \times 2T$.

Now consider the smallest time step t such that at least $m/2$ input bits are read by the chip in time interval $[1, t]$. Let $A_0 \subseteq [m]$ be the set of input bits read in time interval $[1, t-1]$ (note that $|A_0| < m/2$), and pick $m/2 - |A_0|$ of the input bits read at time step t and deonte them by A_1. Note that $|A_1| \le 2T \times 2T$ because the number of input ports is bounded by the chip area. Then $A = A_0 \cup A_1$ and $B = [m] \setminus A$ is a bi-partition of the input bits into two equal-size parts. We have the following (one-way) communication protocol for TRIANGLE when Alice holds input bits in A and Bob holds B: Alice simulates the VLSI chip up to time step $t-1$ (she knows all the input bits A_0 required to do this simulation), and sends the state of the chip together with all the input bits of A_1 to Bob. The message size is $O(2T \times 2T) + |A_1| \le O(T^2)$. Then, Bob has all the required information to simulate the chip starting from time step t to the end, and the output bit of the chip gives the correct answer to the TRIANGLE problem.

By the worst-case-partition communication complexity lower bound of $\Omega(m)$, for TRIANGLE, we must have $T^2 \ge \Omega(m)$, that is $T \ge \Omega(\sqrt{m})$. □

Now we show how to decide TRIANGLE in our wire-delay VLSI model, in $\tilde{O}(n)$ time. We will first describe a version of the classic systolic array algorithm for $n \times n \times n$ matrix multiplication, which we will modify slightly for our triangle detection upper bound, as well as our later upper bound for ORTHOGONAL VECTORS (Theorem 1.12). Systolic arrays for matrix multiplication and similar problems are well-known [KL79] and common in practice [JYP+17]; the one described here is similar to the older Cannon's algorithm [Can69].

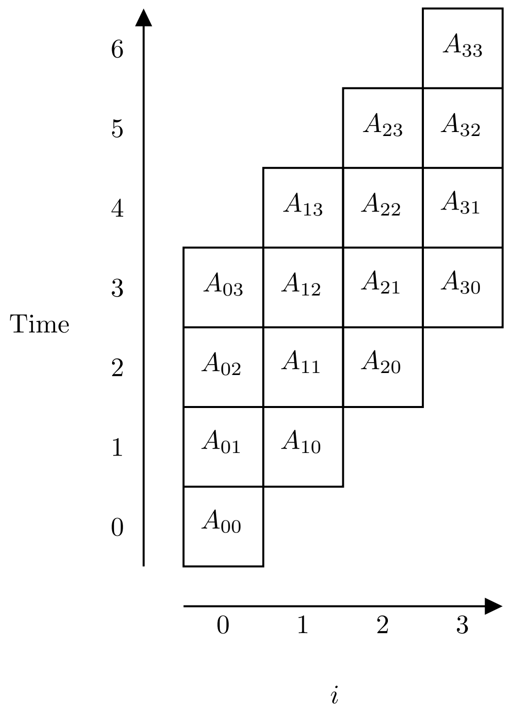


Figure 9: Input schedule for a 4x4 matrix in the systolic array

Construction: Systolic Array for Matrix Multiplication. We will create a $n \times n$ circuit array. Each cell (i, j) is responsible for accumulating the output C_{ij}. It starts running at cycle $i + j$, and runs for n cycles. At cycle $i + j + k$, it receives B_{kj} from the south and A_{ik} from the west, and performs $C_{ij} \leftarrow C_{ij} + A_{ik} * B_{kj}$, then replicates A_{ik} at the east output and B_{kj} at the north output. In this way, the entire matrix multiplication can be done in $3n$ cycles by successive outer products, with the matrices A and B input at the south and west edge of the array.

Theorem 1.9. TRIANGLE *on an n-node graph can be solved by a wire-delay VLSI chip of area $\tilde{O}(n^2)$ and delay $\tilde{O}(n)$.*

Proof. We will view our input as the upper-triangular-part of the adjacency matrix of a graph G. Let A be the upper-triangular matrix which agrees on this upper triangle but has zeros besides, i.e., $A_{ij} = 1$ for all edges (i, j) and $i < j$. We recall there is a triangle (i, k, j) (where $i < k < j$) in G if and only if there is an (i, j) such that $A_{ij} > 0$ and $A^2_{ij} > 0$ [IR78]. We will create a $n \times n$ array to multiply A by itself, then check whether there is any i, j for which $A^2_{ij} > 0$ and $A_{ij} > 0$.

Our array will differ from the simple matrix multiplication array above in three ways: first, we have to handle input carefully, since we need to multiply A by itself (and cannot simply replicate it at the input due to semellectivity); second, we need to keep A around so that we can compare it to A^2 at the end; third, we can cap output values at 1, since we are only interested in *whether* there is a 2-path for each pair of nodes, not how many 2-paths there are.

In our array, each cell (i, j) will take the bit A_{ij} as input at the start of the computation, and remember it throughout (if $i \geq j$, this bit will be initialized as zero instead). In order to get elements of A to the borders of the array (from which they can be sent through normally), every cell will also have inputs from the north and east, and outputs to the south and west, in order to bring values of A to the borders at the correct time. Cell (i, j) will use a cycle counter (of $\log n$ bits) to determine when to send A_{ij} to the south or west output. Since A_{ij} needs to be at cell $(i, 0)$ at cycle $i + j$, and cell (i, j) is j cells (and thus j cycles) away, it will output A_{ij} to the south at cycle i; otherwise, it will replicate the north input at the south.

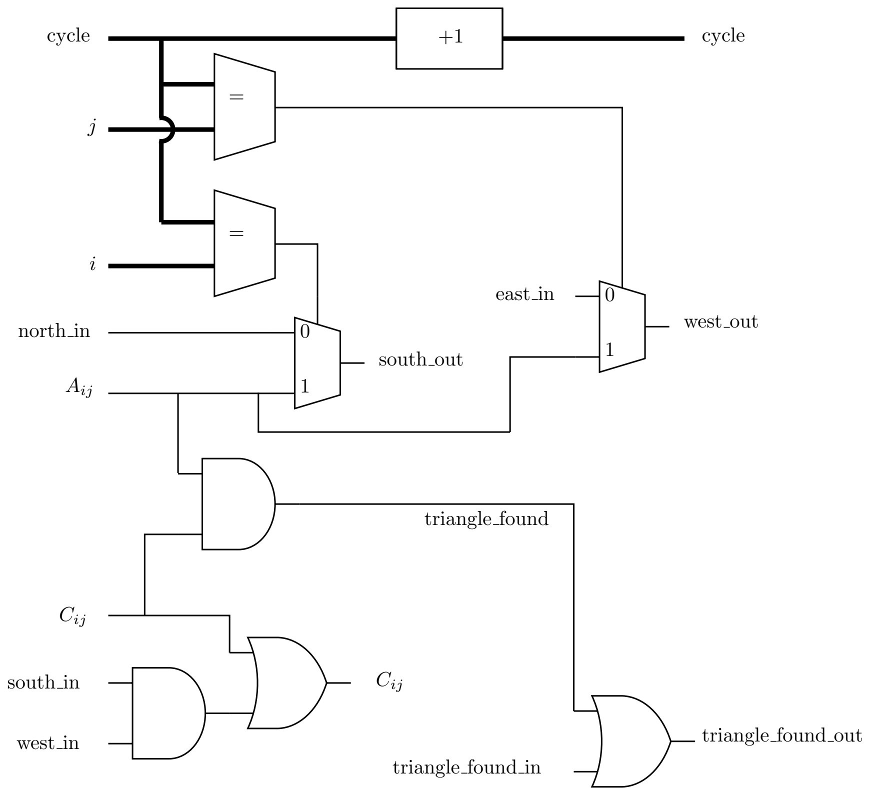


Figure 10: The circuit for a (non-border) cell of the array in Theorem 1.9. Flip-flops are implicit whenever inputs and outputs have the same name (and for A_{ij}, i, and j, which do not change). Thick lines indicate bundles of $\log n$ wires for cycle counts.

Similarly, it will output A_{ij} to the west at cycle j.

The squared adjacency matrix will be computed normally, except instead of cell (i, j) performing $C_{ij} \leftarrow C_{ij} + A_{ik} * A_{kj}$ at cycle $i+j+k$, it will perform $C_{ij} \leftarrow C_{ij} \vee (A_{ik} \wedge A_{kj})$ (i.e., Boolean matrix multiplication). Thus C_{ij} will be a single bit indicating whether there is at least one 2-path between nodes i and j in the input graph.

After all inputs have been processed at cell (i, j), the cell will then compute $C_{ij} \wedge A_{ij}$, which is true if and only if a triangle has been found containing the edge (i, j). These bits need to be OR-ed together to produce the final output of whether any triangle exists; this can be done by accumulating first along columns (each cell will take a combined 'triangle found' bit from the north, OR that bit with its own, and send the result to the south), and then across columns (each cell in the southernmost row will take a combined 'triangle found' bit from the north and east, OR them with its own, and send the result to the west).

To see that this chip solves TRIANGLE in area $\tilde{O}(n^2)$ and delay $\tilde{O}(n)$, note that each cell can be implemented by a circuit of size $O(\log n)$, and the array as a whole takes $O(n)$ cycles to finish: $3n$ cycles for the matrix multiplication, plus another $2n$ for the accumulation of 'triangle found' bits. By Theorem 2.4, this means the array as a whole takes area $O(n^2 \log^2 n)$ and delay $O(n \log n)$. $\square$

8 Simulating Typical Circuits in Our Model

In this section, we will give the proof of Theorem 1.10, showing how to implement any circuit of size n and depth d with fan-in and fan-out bounded by two, using a VLSI chip of area $n \cdot \text{poly}(\log n)$ and delay $d\sqrt{n} \cdot \text{poly}(\log n)$. We recall that the fan-out two restriction can be removed without loss of generality, as

Hoover, Klawe, and Pippenger [HKP84] showed that every size-s depth-d circuit with constant fan-in and unbounded fan-out can be simulated by a circuit with fan-in two, fan-out two, size $O(s)$, and depth $O(d)$.

We will implement our VLSI chip that simulates the circuit in the array model and apply Theorem 2.4: assume WLOG that n is a perfect square, and we will implement our circuit in a $\sqrt{n} \times \sqrt{n}$ grid of cells, where each cell is mapped (arbitrarily) to a unique gate in the circuit. To simulate the circuit, for every directed edge (u, v) in the circuit, we need to let the cell representing gate u send its output value to the cell representing gate v. To perform these communications, we will implement a *packet-routing interconnect*, following the grid pattern. When a cell receives the two binary inputs to its gate, it will produce up to two $O(\log n)$-bit packets carrying the output, along with hard-coded destinations. Inputs to the chip can be placed directly in the registers which would otherwise carry the gate inputs, and outputs from the chip can be emitted as soon as they are produced instead of (or in addition to) generating packets. We will design our interconnect and routing algorithm such that:

- every cell can be implemented by a circuit of size $\mathrm{poly}(\log n)$, and
- every packet reaches its destination within $O(\sqrt{n})$ clock cycles.

For a circuit of depth d, the total number of cycles to complete the circuit will be $O(d\sqrt{n})$, since all paths in the circuit run in parallel and each edge takes at most $O(\sqrt{n})$ cycles. Our result will be provided by using Theorem 2.3 to implement each cell, thus achieving area

$$\sqrt{n} \cdot \mathrm{poly}(\log n) \times \sqrt{n} \cdot \mathrm{poly}(\log n) \leq n \cdot \mathrm{poly}(\log n)$$

and a clock period of $\mathrm{poly}(\log n)$, resulting in a circuit of total delay at most $d\sqrt{n}\mathrm{poly}(\log n)$.

8.1 Routing the packets

We will communicate packets among the cells using a well-known scheme due to Valiant and Brebner [VB81, section 8]. We first send the packet to a random column in the source row, then send it along that column the destination row, and then send it along the destination row to the destination node. Since we know all packet sources and destinations ahead of time from the topology of our circuit and its placement in the grid, we do not have to bother with randomness, and can simply hardcode an optimal choice of "random" columns.

We only need the following facts from our network and routing:

1. When a packet enters a node and does not need to change directions (i.e. it entered from the north and is destined for the south, or entered from the east and is destined for the west, or vice versa), it is given priority and exits the opposing side in the next cycle with no stall and without being placed in a queue.
2. Packets in a queue are sent out (in any order) whenever the necessary output port is open; they do not wait longer than necessary.
3. At most $O(\sqrt{n})$ packets ever pass through any given node.
4. At most $O(\log n)$ packets ever change direction at any given node.

We will construct cells to ensure (1) and (2); (3) and (4) were shown to be true by Valiant and Brebner with high probability over the choice of random columns, and so will certainly be true in our case, where we hardcode an optimal choice. Valiant and Brebner worked in a slightly different setting, where all packets were produced at the same time and at most one packet was produced and received per cell. These facts still hold in our setting where packets are produced at arbitrary times because they are combinatorial statements about the set of paths, not statements about timing directly, and they still hold for our 2 packets produced and received per cell because they are asymptotic.

To find the delay of a packet reaching its destination, note that every packet only waits in 3 queues at most (at its source, at the random column in the source row, and at the random column in the destination row). Each time it waits, it waits at most $O(\sqrt{n})$ cycles for other packets to pass by (due to facts (2) and (3)), and when it is traveling, it takes at most $O(\sqrt{n})$ cycles to reach the next node where it waits (due to fact (1)). This means every packet reaches its destination at most $O(\sqrt{n})$ cycles after it is produced at its source.

8.2 Packet headers

Every node that a packet passes through needs to know where to send it next. In order to accomplish this, every packet will include a header of $O(\log n)$ bits describing its route: $\log n$ bits for the index of the 'random' column, $\log n$ bits for the destination row, $\log n$ bits for the destination column, 1 bit for whether it represents the left or right input of the destination gate, and 2 bits for which phase the packet is in (send to random column, send to destination row, or send to destination column). Packets have a 1-bit payload, and will be accompanied by a 1-bit value indicating validity, so that each cell knows whether a new packet is present at any given input in any given cycle.

8.3 The grid block

Now we describe how to implement each cell in the array.

8.3.1 Structure

A cell consists of several components:

- An input and output in each cardinal direction, each capable of transmitting one packet per cycle (so $O(\log n)$ bit width)
- A queue capable of storing $O(\log n)$ packets, enough to not overflow during operation (so $O(\log^2 n)$ bits of storage total)
- Two slots for packets holding gate input for this cell, and two slots for packets holding gate output (pre-filled with the correct packet headers for routing the outputs)
- Logic for controlling the behavior of the cell as described below

8.3.2 One-cycle behavior

At each cycle, every input is checked to determine whether there is a packet coming in. If there is, the 3 coordinate values and the phase bits are used to determine where the packet should be sent next. For instance, if the phase is 'random column', the packet is coming from the east, and the first coordinate is smaller than the coordinate of the current cell, the packet needs to be sent directly to the west output.

If a packet is going "straight," it should be sent to the correct output immediately. If the current cell is its final destination, it should be placed in the correct gate input register. If the packet is headed in any other direction, it is "turning," and should be placed in an empty spot in the queue.

If any of the 4 outputs are unused after considering the packets which are going "straight," the queue should be checked to see if it contains any packets headed for an unused output. For every unused output for which at least one packet was found in the queue, the first such packet should be removed from the queue and sent to the output.

If both gate input registers contain valid packets, they should be invalidated, and both gate output spots in the queue should be filled with the output of the gate on the two input payloads and marked valid for the next cycle.

This behavior involves only $O(\log n)$ operations of $O(\log n)$ bits each, and so it is easy to see it can be implemented with a circuit of size $O(\log^2 n)$.

9 A Conditional Lower Bound

In this section, we borrow a popular conjecture in fine-grained complexity to show a conditional $n^{2/3-o(1)}$ delay lower bound in our wire-delay VLSI model, for a simple decision problem. Recall the Orthogonal Vectors problem stated in the introduction:

Orthogonal Vectors (OV)
Given: Sets $A, B \subseteq \{0,1\}^d$, $|A| = |B| = n$
Decide: Is there a $u \in A$ and $v \in B$ such that for all i, $u[i] \cdot v[i] = 0$?

Recall the Non-Uniform OV Conjecture (Conjecture 1.11) states that for superlogarithmic d, there is no circuit family deciding Orthogonal Vectors with size $O(n^{2-\varepsilon})$, for every $\varepsilon > 0$.

How quickly can OV be solved in our VLSI model? A square-root speed-up is not hard to achieve: there is a simple formula of depth $O(\log n)$ and size $O(n^2 d)$ for Orthogonal Vectors [KW19]. Applying our simulation theorem Theorem 1.10, this implies a VLSI chip for OV that runs in time $n \cdot \text{poly}(\log n)$ when $d = \text{poly}(\log n)$. In fact, OV can be solved with a cube-root speed-up in our VLSI model.

Theorem 1.12. *The Orthogonal Vectors problem on n vectors in d dimensions can be solved by a wire-delay VLSI chip of area $\tilde{O}(n^{4/3}d^2)$ and delay $\tilde{O}(n^{2/3}d)$.*

Proof. (Sketch) For convenience, we assume $n^{1/3}$ is an integer, and we assume the input vectors are $A[0], \ldots A[n-1] \in \{0,1\}^d$ and $B[0], \ldots B[n-1] \in \{0,1\}^d$. The computation for this chip will be done by a $n^{2/3} \times n^{2/3}$ array, which will operate in a very similar manner to the classic systolic array for matrix multiplication described in Section 7. In order to check whether any of the n^2 vector pairs are orthogonal, each cell in the array will be responsible for checking $n^{2/3}$ vector pairs (one pair per cycle), and will accumulate a bit indicating whether an orthogonal pair has been found. More specifically, each cell (i,j) (for $i,j \in [0, n^{2/3}-1]$) at cycle $i+j+k$ (where $0 \le k < n^{2/3}$) will receive two vectors,

$$A\left[i \cdot n^{1/3} + \left(k \bmod n^{1/3}\right)\right] \text{ from the south, and } B\left[j \cdot n^{1/3} + \left\lfloor \frac{k}{n^{1/3}} \right\rfloor\right] \text{ from the west,} \tag{1}$$

and will compute whether these vectors are orthogonal, and update its ‘orthogonal pair found’ bit accordingly. It will also replicate the A vector from the south at the north output, and the B vector from the west at the east output. See Fig. 11 for an illustration. (Note that cell (i,j) will start receiving vector pairs at cycle $i+j$, and will finish receiving vector pairs at cycle $i+j+n^{2/3}$.) In order to implement this schedule, the cell can include a cycle counter of $O(\log n)$ bits which counts the current cycle number.

From Eq. (1) it is easy to verify that each of the n^2 vector pairs is checked by some cell at some cycle: the pair $(A[x], B[y])$ $(0 \le x, y < n)$ is checked by

$$\text{cell } (i,j) = \left(\left\lfloor \frac{x}{n^{1/3}} \right\rfloor, \left\lfloor \frac{y}{n^{1/3}} \right\rfloor\right) \text{ at cycle } \left(y \bmod n^{1/3}\right) \cdot n^{1/3} + \left(x \bmod n^{1/3}\right) + i + j.$$

Hence we can correctly solve OV by aggregating the checking results of all cells in the array.

After a cell has checked all the $n^{2/3}$ vector pairs assigned to it, its ‘orthogonal pair found’ bit will correctly represent whether any of the pairs were orthogonal. These bits need to be ORed together to produce the final output of whether any orthogonal pair exists; this can be done by accumulating first along columns (each cell will take a combined ‘pair found’ bit from the north, OR it with its own, and send the result to the south), and then across columns (each cell in the southmost row will take a combined ‘pair found’ bit from the north and east, OR them together and with its own, and send the result to the west).

Now it remains to describe how to implement the schedule so that the cells receive the correct vector pair according to Eq. (1).

Since each cell replicates its A and B input vectors to the north and east respectively in the next cycle, same as the classic matrix multiplication array in Section 7, it is sufficient to implement the correct schedule

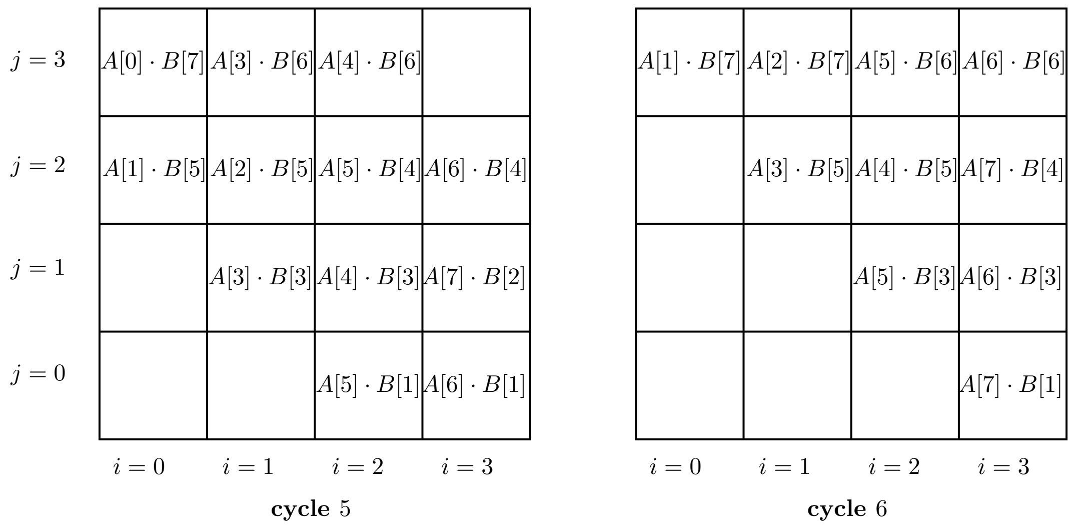


Figure 11: An illustration of the $n^{2/3} \times n^{2/3}$ array where $n = 8$ (and $n^{2/3} = 4$) at cycle 5 and cycle 6 is shown. It is indicated inside each cell which vector pair it is checking at this cycle, as defined in Eq. (1); empty cells are idle. As one can observe from comparing cycle 5 and cycle 6, vectors from A are sent one step to the north, and vectors from B are sent one step to the east.

of A vectors at the southernmost row and B vectors at the westernmost column, and the rest of the schedule will follow.

In order to implement this schedule, we will augment the array at the south and west sides: we will create a $n^{2/3} \times n^{1/3}$ array of cells to the south which will store and provide A vectors to the main array, and a $n^{1/3} \times n^{2/3}$ array to the west which will store and provide B vectors. Cell (i, j) of the south array will store the entire vector $A[i \cdot n^{1/3} + j]$, which it will take all at once as input at the start of the computation. Similarly cell (i, j) of the west array will store the entire vector $B[j \cdot n^{1/3} + i]$.

Each cell in the south (respectively, west) array will include an input from the south and an output to the north (respectively, west and east) in order to carry these vectors to the main array when they are needed, as well as a cycle counter and logic for when to output its own vector to the north (respectively, east) instead. Note that, since all vectors propagate in lockstep, one cell per cycle, we do not have to worry about collisions between vectors on these networks. No east-west (respectively, north-south) communication is necessary, because the assignment of vectors to cells ensures that vectors are at exactly the column (respectively, row) in the main array where they will be needed.

The south array needs to provide vector $A[i \cdot n^{1/3} + j]$ to the southernmost row of the main array at every cycle $i + k$ where $k \in [n^{2/3}]$ and $k \bmod n^{1/3} = j$. This vector is stored at cell (i, j) of the south array, which is $n^{1/3} - j$ cells away from the main array. Thus, cell (i, j) should output its vector to the north whenever it finds that $(c - i + n^{1/3} - j) \bmod n^{1/3} = j$, where c is the value of the cycle counter; otherwise, it should replicate its south input to the north. This equation can be checked by the cell using a circuit of size $O(\operatorname{poly}\log n)$. The west array will behave similarly: cell (i, j) in the west array should output its vector to the east whenever it finds that $\lfloor (c - j + n^{1/3} - i)/n^{1/3} \rfloor = i$. Note that, for convenience, we are dealing here with negative cycle counts down to $-n^{1/3} + 1$. All cycle counters should start at this value instead of zero.

To see that this chip takes area $\tilde{O}(n^{4/3}d^2)$ and delay $\tilde{O}(n^{2/3}d)$, note that each cell in the main array contains a size-$O(d)$ circuit to compute whether its two input vectors are orthogonal, a size-$O(\log n)$ cycle counter circuit, and only a constant number of gates besides, for a single-cell circuit size of $O(d + \log n)$. Similarly, each cell in the west and south arrays can be implemented by a circuit of size $d + \operatorname{poly}\log n$. The computation will finish in $5n^{2/3} + n^{1/3}$ cycles: $n^{1/3}$ cycles for the first vectors to leave the west and south arrays, $2n^{2/3}$ cycles until the last cell starts receiving vector pairs, $n^{2/3}$ cycles for the last cell to finish checking those vector pairs, and $2n^{2/3}$ cycles for the accumulation of 'orthogonal pair found' bits to

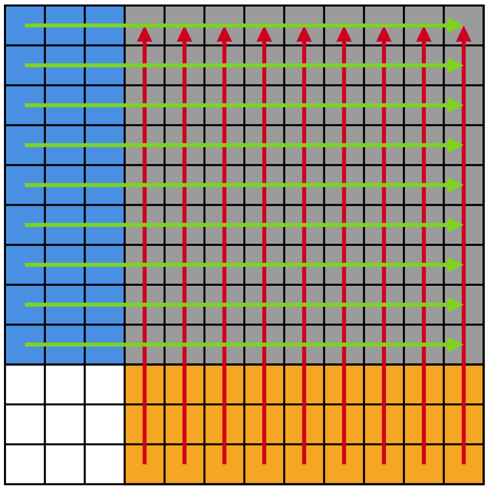


Figure 12: The full array for Theorem 1.12, with $n = 27$. The main computation array is indicated in gray; the west and south arrays are in blue and orange respectively. Blank cells are unused. Red arrows indicate travel of A vectors and green arrows indicate travel of B vectors.

finish. By Theorem 2.4, since we have an $O(n^{2/3}) \times O(n^{2/3})$ array, this is $O(n^{4/3}d^2 \cdot \text{poly}\log n)$ area and $O(n^{2/3}d \cdot \text{poly}\log n)$ delay in total. □

To complete our conditional lower bound argument, we need to show that our wire-delay VLSI model can be efficiently implemented in a typical serial random-access model of computation. The next theorem accomplishes this.

Theorem 1.13. *Given a VLSI chip for a single-output function of n inputs in t delay, there is a (typical, serial) algorithm for that function running in time $O(t^3)$ and using $O(t^2 + n \log t)$ advice bits.*

Proof. By a similar argument as in Theorem 3.1, the computation on the chip that can affect the single output bit within t time steps must be confined in an area of at most $O(t) \times O(t)$, so we can assume that the VLSI chip has area $O(t) \times O(t)$.

Then, we can use a serial algorithm to simulate the computation of the chip, which has area-time volume $O(t^2) \times O(t)$ (it can be visualized as an $O(t) \times O(t) \times O(t)$ dimensional cube). We assume that the description of the chip is given to us in $O(t^2 + n \log t)$ advice bits: the area of the chip is $O(t^2)$ so it takes $O(t^2)$ to describe the chip layout (including the wires and gates) by specifying the behavior of each point (i, j) in the grid (whether it is a gate or is a part of a wire, and the gate function or the direction of the wire) in $O(1)$ bits. Then, for each input bit x_i, $1 \le i \le n$ (and each output bit) of the function, we need to use $O(\log t)$ bits to specify which time step it appears in the chip at which gate. Hence the total description length is $O(t^2 + n \log t)$ bits. Each time step of the chip computation can be simulated in $O(t^2)$ time and space by a serial algorithm with random access to the input. So the total serial time is $O(t^3)$. □

The following conditional lower bound is immediate from Theorem 1.13:

Theorem 1.14. *Assuming the Non-Uniform OV Conjecture, every wire-delay VLSI chip for* Orthogonal Vectors *with n vectors in $\log^2 n$ dimensions requires at least $n^{2/3-o(1)}$ time.*

10 Open Problems

There are many new open problems to consider regarding our wire-delay VLSI model. Here are a few that we especially like.

- Given that OV can be solved with a cube-root speedup in our wire-delay VLSI model (Theorem 1.12), and given the reduction from CNF-SAT to OV [Wil05], is it possible that CNF-SAT on n variables and $O(n)$ clauses can be solved in $2^{n/3} \cdot \text{poly}(n)$ time in our model? This is not immediate, because reductions in the serial world do not necessarily compose well on a VLSI chip. In particular, the reduction from SAT to OV introduces $\Theta(2^{n/2})$ binary vectors of length $O(n)$, the bits of which would need to be "transported" at various places in the chip in order to carry out our OV algorithm in the wire-delay model.

- We have shown (Theorem 1.8) that triangle detection on dense graphs (with $\Theta(n^2)$ edges) requires $\Omega(n)$ delay in our model, and it can be solved with $\tilde{O}(n)$ delay. What about sparse graphs? It is well-known [AYZ97] that triangle detection can be done in $O(m^{3/2})$ time on m-edge graphs on typical serial models of computation. Thus the natural conjecture would be that $O(\sqrt{m})$ delay is necessary and sufficient for triangle detection in our model. If this conjecture is true, it would nicely generalize our results for the dense case.

- What other conditional lower bounds can be found and matched in the wire-delay VLSI model? For instance, assuming the OV hypothesis, both Edit Distance and Longest Common Subsequence require $n^{2-o(1)}$ time [BI18, ABW15, BK15]. By the simulation from Theorem 1.13, this means these two problems do not have wire-delay VLSI chips in $O(n^{2/3-\varepsilon})$ time under the non-uniform OV hypothesis. Are there wire-delay chips solving these problems in $\tilde{O}(n^{2/3})$ time? The well-known $O(n^2)$-time serial algorithm for these problems based on dynamic programming has large depth and does not seem to immediately imply an $\tilde{O}(n^{2/3})$ wire delay.

- Under realistic assumptions like those of wire-delay VLSI, which algorithms (parallel and serial) are more or less likely to be high-performance, compared to more common models? How should algorithm structure be studied to account for wire delay?

- Our $\Omega(n^{1/2})$ lower bounds for various problems rely on the assumption that the chips are *semellective*. Is there an $\Omega(n^{1/2})$ delay lower bound for a natural problem without this assumption, where each input bit may be read *arbitrarily* many times by the chip?

- In our wire-delay model, every nondegenerate function needs at least $\Omega(n^{1/3})$ delay. Moreover, we can simulate any time t wire-delay chip on a serial model that uses time about t^3 (Theorem 1.13). Thus our model could "only" ever speed up a serial computation by a cube root. Is there a conceivable parallel model (consistent with known physics) in which the speed-up over serial computation could be asymptotically reduced even further, even less a cube root? Fisher [Fis88] shows that under natural physical assumptions, the fastest parallel simulation of a t-time algorithm that we could plausibly hope for in a two-dimensional model is $\Omega(t^{1/3})$. (In a three-dimensional computational device, $\Omega(t^{1/4})$ delay is required, under his assumptions.) But we are unconvinced that the final word has been written on this topic. For instance, we have not considered quantum computational phenomena at all in this work; nor did Fisher.

Acknowledgements. N.Y. thanks John Wawrzynek, his M.S. advisor, for excellent support on earlier work on this topic, as well as Grace Dinh, Jonathan Greene, and Adrian Fan for useful conversations. C.J. and R.W. thank the Ethereum Foundation, in particular Justin Drake and Dankrad Feist, for highlighting various real-world computing issues that this work attempts to address. We all thank Avishay Tal for bringing us together to work on this topic.

References

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Williams. Tight hardness results for lcs and other sequence similarity measures. In *Proceedings of IEEE FOCS*, pages 59–78, 10 2015. doi:10.1109/FOCS.2015.14.

[AWY15] Amir Abboud, R. Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Proceedings of ACM-SIAM SODA*, pages 218–230. SIAM, 2015. doi:10.1137/1.9781611973730.17.

[AYZ97] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. doi:10.1007/BF02523189.

[BI18] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

[BK81] Richard P. Brent and H. T. Kung. The area-time complexity of binary multiplication. *J. ACM*, 28(3):521–534, 1981. doi:10.1145/322261.322269.

[BK15] Karl Bringmann and Marvin Kunnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of IEEE FOCS*, pages 79–97, 10 2015. doi:10.1109/FOCS.2015.15.

[BKL83] Richard P. Brent, H. T. Kung, and Franklin T. Luk. Some linear-time algorithms for systolic arrays. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 865–876. North-Holland/IFIP, 1983.

[Bla17] Christopher Graham Blake. *Energy Consumption of Error Control Coding Circuits.* PhD thesis, University of Toronto, 2017.

[Can69] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, USA, 1969. AAI7010025.

[CM81] Bernard Chazelle and Louis Monier. A model of computation for VLSI with related complexity results. In *Proceedings of ACM STOC*, pages 318–325. ACM, 1981. doi:10.1145/800076.802485.

[CM83] Bernard Chazelle and Louis Monier. Unbounded hardware is equivalent to deterministic turing machines. *Theor. Comput. Sci.*, 24:123–130, 1983. doi:10.1016/0304-3975(83)90044-0.

[CM85] Bernard Chazelle and Louis Monier. A model of computation for VLSI with related complexity results. *J. ACM*, 32(3):573–588, 1985. doi:10.1145/3828.3834.

[CW21] Timothy M. Chan and R. Ryan Williams. Deterministic APSP, orthogonal vectors, and more: Quickly derandomizing Razborov-Smolensky. *ACM Trans. Algorithms*, 17(1):2:1–2:14, 2021. doi:10.1145/3402926.

[Fis88] David C. Fisher. Your favorite parallel algorithms might not be as fast as you think. *IEEE Transactions on Computers*, 37(02):211–213, 1988.

[GBNB$^+$23] Lukas Gianinazzi, Tal Ben-Nun, Maciej Besta, Saleh Ashkboos, Yves Baumann, Piotr Luczynski, and Torsten Hoefler. The spatial computer: A model for energy-efficient parallel computation, 2023. arXiv:2205.04934.

[Gro15] Pulkit Grover. Information friction and its implications on minimum energy required for communication. *IEEE Transactions on Information Theory*, 61(2):895–907, 2015. doi:10.1109/TIT.2014.2365777.

[HKP84] H. James Hoover, Maria M. Klawe, and Nicholas Pippenger. Bounding fan-out in logical networks. *J. ACM*, 31(1):13–18, 1984. doi:10.1145/2422.322412.

[IR78] Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput.*, 7(4):413–423, 1978. doi:10.1137/0207033.

[JCL$^+$17] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, 2017. doi:10.1109/IMW.2017.7939084.

[JYP$^+$17] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. URL: http://arxiv.org/abs/1704.04760, arXiv:1704.04760.

[KL79] Hsiang Tsung Kung and Charles E Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for Industrial and Applied Mathematics, 1979.

[Kra67] S. S. Kravtsov. Realization of boolean functions in a class of circuits of functional and switching elements. *Probl. Kibern*, (19):285–292, 1967.

[Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982. doi:10.1109/MC.1982.1653825.

[KW19] Daniel M. Kane and R. Ryan Williams. The orthogonal vectors conjecture for branching programs and formulas. In *Proceedings of ITCS*, volume 124 of *LIPIcs*, pages 48:1–48:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ITCS.2019.48.

[Lei79] Charles E Leiserson. Systolic priority queues. In *Caltech Conference on VLSI*. California Institute of Technology, January 1979.

[Len90] Thomas Lengauer. VLSI theory. In *Algorithms and Complexity*, pages 835–868. Elsevier, 1990.

[LT79] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[LT80] Richard J. Lipton and Robert E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[LZ20] Sergei Andreevich Lozhkin and Vadim Sergeevich Zizov. Refined estimates of the decoder complexity in the model of cellular circuits with functional and switching elements. *Proceedings of Kazan University. Physics & Mathematics Series/Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki*, 162(3), 2020.

[Mar14] Igor L. Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014.

[MP83] Kurt Mehlhorn and Franco P. Preparata. Area-time optimal VLSI integer multiplier with minimum computation time. *Inf. Control.*, 58(1-3):137–156, 1983. doi:10.1016/S0019-9958(83)80061-8.

[Pre83] Franco P. Preparata. A mesh-connected area-time optimal VLSI multiplier of large integers. *IEEE Trans. Computers*, 32(2):194–198, 1983. doi:10.1109/TC.1983.1676203.

[PS84] Christos H. Papadimitriou and Michael Sipser. Communication complexity. *J. Comput. Syst. Sci.*, 28(2):260–269, 1984. doi:10.1016/0022-0000(84)90069-2.

[RCN04] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits- A design perspective.* Prentice Hall, 2ed edition, 2004.

[Ros60] Gerald B. Rosenberger. Simultaneous carry adder, December 27 1960. US Patent 2,966,305.

[RV13] Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of ACM STOC*, pages 515–524. ACM, 2013. doi:10.1145/2488608.2488673.

[Sav81] John E Savage. Planar circuit complexity and the performance of VLSI algorithms. In *VLSI Systems and Computations*, pages 61–68. Springer, 1981.

[Sav98] John E Savage. *Models of computation*, volume 136. Addison-Wesley Reading, MA, 1998.

[SP10] Rupesh S Shelar and Marek Patyra. Impact of local interconnects on timing and power in a high performance microprocessor. In *Proceedings of the 19th international symposium on Physical design*, pages 145–152, 2010.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen [Fast multiplication of large numbers]. *Computing*, 7(3-4):281–292, 1971. doi:10.1007/BF02242355.

[Tho80] Clark David Thompson. *A complexity theory for VLSI.* Carnegie Mellon University, 1980.

[Ull84] Jeffrey D Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[Vas18] Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians*, pages 3447–3487. World Scientific, 2018.

[VB81] Leslie G. Valiant and Gordon J. Brebner. Universal schemes for parallel communication. In *Proceedings of ACM STOC*, pages 263–277. ACM, 1981. doi:10.1145/800076.802479.

[Wil05] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.

[You22] Nathaniel Young. An updated model of computation for VLSI and applications to FPGA implementation. Master's thesis, EECS Department, University of California, Berkeley, May 2022. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-108.html.

A Appendix: Further Details on the CM Model

Chazelle and Monier utilize the following simple lemma in their lower bounds:

Lemma A.1 ([CM85]). *Let P be an arbitrary convex polygon with perimeter N. Then the maximum distance between any vertex of P and an arbitrary point in the plane is $\Omega(N)$.*

An immediate consequence of their lemma is that it takes $\Omega(N)$ time to "propagate a bit" from any point inside a convex region to N points on the boundary of that region. In their lower bounds, they argue that if p is the number of input ports used by the circuit to compute on some input, and the function being computed depends on all n of its inputs, then it must take at least $\Omega(n/p+p)$ time to compute that function (the factor of $\Omega(p)$ coming from Lemma A.1, the factor of $\Omega(n/p)$ coming from the fact that all n bits need to eventually be put into the circuit). We end up requiring $\Omega(\sqrt{n})$ time as the bound is minimized for $p \approx \sqrt{n}$.

Chazelle and Monier also define a notion that we would nowadays call "sensitivity", as follows.

Definition A.2 ([CM85])**.** A function $f : \{0,1\}^n \to \{0,1\}$ is a *fan-in of degree* n if there's an $a \in \{0,1\}^n$ such that for all i, $f(a) \neq f(a^{(i)})$ (recall $a^{(i)}$ is a with the i-th bit flipped). We say that such an a is a *hard input* for f.

Example: AND on n bits is a fan-in of degree n, with hard input $(1,\ldots,1)$.

Theorem A.3 ([CM85])**.** *Every circuit in the CM model computing a fan-in of degree n requires time* $\Omega(\sqrt{n})$.

Corollary A.4. *Addition of two n-bit integers requires* $\Omega(\sqrt{n})$ *time in the CM model.*

Proof. Recall that we can reduce the AND function (which is a fan-in of degree n) to addition of two n-bit integers with essentially no overhead: the AND of x is determined by reading the high-order bit of $x+1$. $\square$

Similarly, computing PARITY on n bits also requires $\Omega(\sqrt{n})$ time in the CM model. One can also compute PARITY in $O(\sqrt{n})$ time in the CM model, by having a $\sqrt{n} \times \sqrt{n}$ "grid" of XOR gates which has the output gate on one corner of the grid, the $p = O(\sqrt{n})$ gates on the perimeter taking in p bits at a time, and the grid computing the XOR of the perimeter and sending that to the corner. This takes $O(n/p) \leq O(\sqrt{n})$ iterations to cover all n bits.