Particle Computation: Device Fan-out and Binary Memory

Aaron Becker, Rose Morris-Wright, Erik D. Demaine, Sándor P. Fekete

***Abstract*— Consider a 2D grid world, where all obstacles and robots are unit squares, and for each actuation, robots move maximally until they collide with an obstacle or another robot. We demonstrated *particle computation* in this world, designing obstacle configurations that implement AND and OR logic gates. By using dual-rail logic, we designed NOT, NOR, NAND, XOR, XNOR logic gates. However, we could not implement a FAN-OUT gate. This prevented us from creating arbitrary digital circuits. In this work we prove unit-sized robots cannot generate a FAN-OUT gate. We introduce 2×1 robots, which can create fan-out gates that produce multiple copies of the inputs. Using these gates we can create complex digital circuits. As an example we connect our logic elements to produce a 3-bit counter. We also implement a data storage element.**

In general I would pay more attention to structuring each paragraph so that you guide your reader by telling them first what you will accomplish in that paragraph and then making sure you support all your assertions. Int he above paragraph it took me until the end of the paragrph to realize that you were trying to make a universal sequence that you could use in all situations. if you say that first, then the reader has some context for why you are talking about these other sequences and what your are trying to accomplish.

I. Introduction

Currently, micro- and nanorobot systems with many robots are steered and directed by a common control signal [?], [?], [?], [?], [?], [?], [?]. These common control signals include global magnetic or electric fields, chemical gradients, and turning a light source on and off. In this paper, we show how a common control signal, mobile particles, and unit-sized obstacles can implement a computer. We do not present particle logic as an alternative to electronic computing. Frankly, this form of computation is impractical, being both slow, requiring large amounts of space, and being vulnerable to manufacturing defects. Rather, we want to quantify the computing power of mobile robotics at the most simple level in order to gain insight for massively-parallel, automated assembly at the micro and nano length-scales. This paper builds on the techniques for controlling many simple robots with uniform control inputs presented in [?], [?], [?], using the following rules:

1) A planar grid workspace W is filled with some unit-square robots (each occupying one cell of the grid) and some fixed unit-square blocks.
2) All robots are commanded in unison: the valid commands are "Go Up" (u), "Go Right" (r), "Go Down" (d), or "Go Left" (l). The robots all move in the commanded direction until they hit an obstacle or a stationary robot. A representative command sequence is $\langle u, r, d, l, d, r, u, \ldots \rangle$. We assume the area of W is known and issue each command long enough for the robots to reach their maximum extent.

A. Becker is with the Department of Cardiovascular Surgery, Boston Children's Hospital and Harvard Medical School, Boston, MA, 02115 USA {first name.lastname}@childrens.harvard.edu, R. Morris-Wright is with the rmorriswright@gmail.com, E. Demaine is with the Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA 02139, USA, edemaine@mit.edu, S. Fekete is with the Dept. of Computer Science, TU Braunschweig, Mühlenpfordtstr. 23, 38106 Braunschweig, Germany, s.fekete@tu-bs.de.

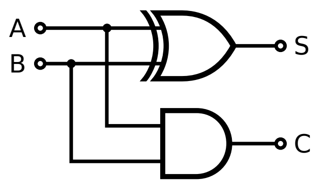


Fig. 1. The half adder shown above requires two copies of $\mathbf{A}$ and $\mathbf{B}$.

After a brief overview of related work, the contributions of this paper are as listed:

1) prove the necessity of dual-rail logic for Boolean logic (Section **??**)
2) prove the insufficiency of unit-size particles for gate fan-out (Section **??**)
3) design FAN-OUT GATES (Section **??**)
4) design memory latches (Section **??**)
5) present architecture for device integration, design a common clock sequence, and present a binary counter (Section **??**)

A. Fan-out

The *fan-out* of a logic gate output is the number of gate inputs it can feed or connect to. With particle logic, as demonstrated in [?], each logic gate output could fan-out to only one gate. This is sufficient for *sum of products* and *product of sums* operations in CPLDs (complex programmable logic devices), but insufficient for more flexible architectures. Consider the half-adder shown in Fig. **??**. The inputs $\mathbf{A}$ and $\mathbf{B}$ are needed to compute both the SUM and the CARRY bits, so the fanout of $\mathbf{A}$ and $\mathbf{B}$ is two.

AND and OR can be implemented with particles, as shown in [?]. However, particle logic is *conservative*—particles are neither created nor destroyed–and we were unable to implement a NOT gate. To implement NOT gates and other logic we used *dual-rail logic*, where two lines for each input are supplied to explicitly represent the variable and its complemen [?].

II. Related Work

Our efforts have similarities with *mechanical computers*, computers constructed from mechanical, not electrical components. For a fascinating nontechnical review, see [**?**]. These devices have a rich history, from the *Pascaline*, an adding machine invented in 1642 by a nineteen-year old Blaise Pascal; Herman Hollerith's punch card tabulator in 1890; to the mechanical devices of IBM culminating in the 1940s. These devices used precision gears, pulleys, or electric motors to carry out calculations. Though our GRIDWORLD implementations seem an anachronism, note that we require none of these precision elements—merely unit-size obstacles, and 2×1 and 1×1 sliding particles.

A. Collision-based computing

Collision-based computing refers to implementations of logical circuits or other computing devices in homogeneous media with traveling mobile localizations. lousy preceding sentence For a survey of this area, see the excellent collection [**?**]. One example is Conway's game of life [**?**] these simple rules have been examined in depth and used to build a Turing-complete computer [**?**]. Game of life scenarios are fascinating, but lack a physical implementation. They require *cells* that live or die based on the number of neighbors. In this paper we present a collision-based system for computation and provide a physical implementation. Todo: this paragraph is incomplete

B. Sliding-block puzzles

Sliding block puzzles use rectangular tiles that are constrained to move in a 2D workspace. The objective is to move one or more tiles to desired locations. They have a long history. Hearn [**?**] and Demaine [**?**] showed tiles can be arranged to create logic gates, and used this technique to prove P-SPACE complexity for a variety of sliding block puzzles. Hearn expressed the idea of building computers from the sliding blocks—many of the logic gates could be connected together, and the user could propagate a signal from one gate to the next by sliding intermediate tiles. This requires the user to know precisely which sequence of gates to enable/disable. In contrast to such a hands-on approach, with our architecture we can build circuits, store parameters in memory, and then actuate the entire system in parallel using a global control signal.

III. Theory

First we provide terminology to define how robots interact with each other.

a) Define workspace: A *workspace* is a 2-dimensional grid. Each unit square in the workspace is either *free*, which a robot may occupy or *obstacle* which a robot may not occupy. Each square in the grid can be referenced by its Cartesian coordinates (x, y).

b) Define command sequence: A *command sequence* M consists of a ordered sequence of moves $M(k)$ where each $M(k) \in \{u, d, r, l\}$

When each $M(k)$ is executed all robots move in the specified direction until they are stopped by an obstacle or another robot. Suppose for move $M(k)$, the robot a ends at location (i, j). With the addition of more robots, it is possible for one robot to hit another. Why (x_i, y_j), and not (x_i, y_i). – what is j? Could we instead just say location (i, j)?

c) Definition of hit: During move $M(k)$, robot a *hits* robot b if a is prevented from reaching location (i, j) because robot b occupies this location. Robot b's location at the end of $M(k)$ will be (i, j) and robot a's location at the end of $M(k)$ can be calculated as follows:

$$\begin{cases} (i-1, j) & \text{if } M(k) = r \\ (i+1, j) & \text{if } M(k) = l \\ (i, j-1) & \text{if } M(k) = u \\ (i, j+1) & \text{if } M(k) = d \end{cases}$$

Theorem 1: If given an workspace W and a command sequence M that moves a robot initially at s to a goal location g, adding additional robots anywhere in W at any stage of the command sequence cannot prevent g from being occupied at the conclusion of sequence M.

Proof: This theorem addresses how adding more robots can effect the final outcome. As robot a travels from s to g it passes through a sequence of locations $\{(x_i, y_j)\}$. We call this sequence of locations a's *path*.

Consider the effect of adding robot b to this existing workspace. If no hit occurs between a and b, then a follows the same path and at the conclusion of M occupies g. The only way to change a's path is to give it a new obstacle to hit, namely another robot.

Suppose a hits b. By the definition of a hit, b prevents a from reaching some location (x, y) because b already occupies this location. After the hit, the command sequence will continue and so robot b will continue on a's original path, following the same instructions and therefore ending up in the same location, g. Thus by adding more robots, it is impossible to prevent some robot from occupying g at the conclusion of M. ■

Corollary 2: A NOT gate without dual-rail inputs cannot be constructed

Proof: By contradiction. A particle logic NOT gate without dual-rail inputs has one input at s, one output at g, an arbitrary number of asserted inputs which are all initially occupied, and an arbitrary number of waste outputs. Given a command sequence M:

- if s is initially unoccupied, at the conclusion of M g must be occupied
- if s is initially occupied, at the conclusion of M g must be unoccupied

By Theorem **??**, if s initially unoccupied results in g being occupied at the conclusion of M, the addition of a robot at s cannot prevent g from being filled, resulting in a contradiction. ■

Inputs			*Outputs*			
A	$\overline{A}$	1	A	A	$\overline{A}$	$\overline{A}$
0	1	1	0	0	1	1
1	0	1	1	1	0	0

TABLE I

FAN-OUT OPERATION. THIS CANNOT BE IMPLEMENTED WITH 1×1 PARTICLES AND OBSTACLES. OUR TECHNIQUE USES 2×1 PARTICLES.

A	B	$A \vee B$	AB	$\overline{A \vee B}$	$\overline{AB}$	$A \oplus B$	$\overline{A \oplus B}$
0	0	0	0	1	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	1	1	0	0	0	1

TABLE II

POSSIBLE BOOLEAN OPERATIONS IN DUAL-RAIL PARTICLE LOGIC.

Theorem 3: If given an workspace W and a command sequence M that moves two robots initially at s_1 and s_2 to respective goal locations g_1 and g_2, then removing one robot results in either g_1 or g_2 being occupied at the conclusion of M.

Proof: If robots s_1 and s_2 never hit when both exist, then the remaining robot continues to its goal location unchanged.

If robots do hit when both exist

```
show that this cannot lead to a goal
location g_3, when only one robot is
used, it goes to the goal location
occupied by the last robot to be hit
when there are two.
```

■

Corollary 4: A FAN-OUT gate cannot be constructed using only 1×1 robots.

I think we have all the pieces here, but we need to say something about it being conservative. That is a key piece of our argument here.

Proof: By contradiction. Consider a FAN-OUT gate W, dual-rail inputs s_a, $s_{\bar{a}}$, dual rail outputs $\{g_{a1}, g_{a2}, g_{\bar{a}1}, g_{\bar{a}2}\}$, and one or more supply locations initially occupied. A FAN-OUT gate implements the truth table shown in Table **??**. Given an arbitrary command sequence M:

- if s_a is initially occupied and $s_{\bar{a}}$ vacant, at the conclusion of M g_{a1} and g_{a2} are occupied and the locations $g_{\bar{a}1}$ and $g_{\bar{a}2}$ are vacant.
- if s_a is initially vacant and $s_{\bar{a}}$ occupied, at the conclusion of M g_{a1} and g_{a2} are vacant and the locations $g_{\bar{a}1}$ and $g_{\bar{a}2}$ are occupied.

However, if input s_a is initially vacant, by Thm. **??**, either g_{a1} or g_{a2} must be occupied at the conclusion of M. By Thm. **??**, adding an additional robot at location $s_{\bar{a}}$ cannot prevent one of g_{a1} and g_{a2} being filled, thus arriving at a contradiction. ■

IV. DEVICE AND GATE DESIGN

A. Choosing a clock signal

The *clock sequence* is the ordered set of moves that are simultaneously applied to every particle in our workspace. We call this the clock sequence because, as in digital computers, this sequence is universally applied and keeps all logic synchronized.

A clock sequence determines the basic functionality of each gate. To simplify implementation in the spirit of Reduced Instruction Set Computing (RISC), which uses a simplified set of instructions that run at the same rate, we want to use the same clock cycle for each gate and for *all* wiring. Our early work used a standard sequence $\langle d, l, d, r \rangle$. This sequence can be used to make AND, OR, XOR, and any of their inverses. This sequence can also be used for *wiring* to connect arbitrary inputs and outputs, as long as the outputs are below the inputs. Unfortunately, $\langle d, l, d, r \rangle$ cannot move any particles upwards. To connect outputs as inputs to higher level logic requires an additional reset sequence that contains an $\langle u \rangle$ command. Including all four directions is a necessary condition for a valid clock sequence. We choose the simplest such sequence, $\langle d, l, u, r \rangle$, and by designing examples prove that this sequence is sufficient for logic gates, FAN-OUT gates, and wiring.

This clock sequence has the attractive property of being a clockwise rotation through the possible input sequences. One could imagine our particle logic circuit mounted on a wheel rotating perpendicular to the ground. If the particles were moved by the pull of gravity, each counter-clockwise revolution would advance the circuit by one clock cycle.

B. A FAN-OUT gate

A FAN-OUT gate with two outputs implements the truth table in Table **??**. This cannot be implemented with 1×1 particles and obstacles, by corollary **??**. Our technique uses 2×1 particles. A single input, two-output FAN-OUT gate is shown in Fig **??**. This gate requires a dual-rail input, a supply particle, and a 2 × 1 slider. The *clockwise* control sequence $\langle d, l, u, r \rangle$ duplicates the dual-rail input.

The FAN-OUT gate can drive multiple outputs. In Fig. **??** a single input drives four-outputs. This gate requires a dual-rail input, three supply particles, and a 2 × 1 slider. The *clockwise* control sequence $\langle d, l, u, r \rangle$ quadruples the dual-rail input. In general, an n output FAN-OUT gate with control sequence $\langle d, l, u, r \rangle$ requires a dual-rail input, $n - 1$ supply particles, and one 2 × 1 slider. It requires $4(n+1) \times 2(n+1)$ area.

C. Data Storage

A general-purpose computer must be able to store data. A 2 × 1 particle enables us to construct a read/writable data storage for one bit. A single-bit data storage latch is shown in Fig. **??** and implements the truth table in Table **??**. By combining an n-out FAN-OUT gate shown in Fig **??** with n

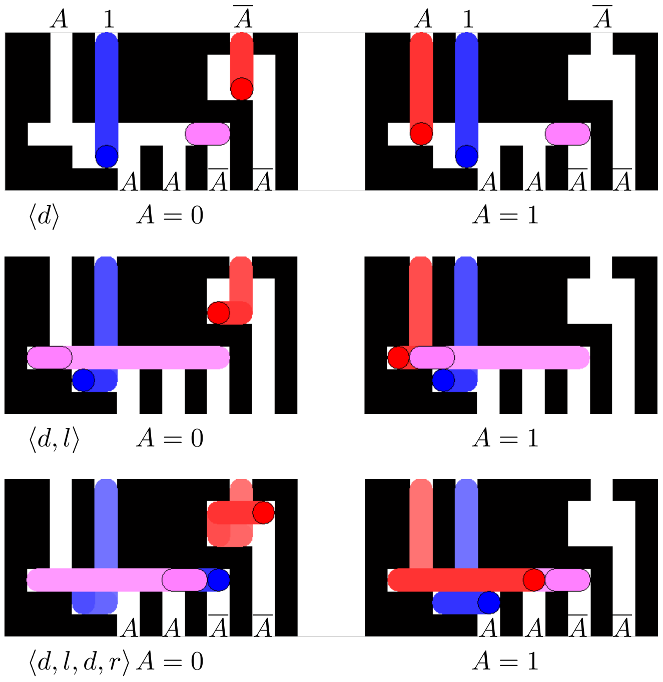

A 1 $\overline{A}$
A 1 $\overline{A}$
A A $\overline{A}$ $\overline{A}$
$\langle d \rangle$ $A=0$
$A=1$
$\langle d, l \rangle$ $A=0$
$A=1$
$\langle d, l, d, r \rangle$ $A=0$
$A=1$

Fig. 2. A single input, two-output FAN-OUT gate. This gate requires a dual-rail input, a supply particle, and a 2×1 slider. The *clockwise* control sequence $\langle d, l, u, r \rangle$ duplicates the dual-rail input.

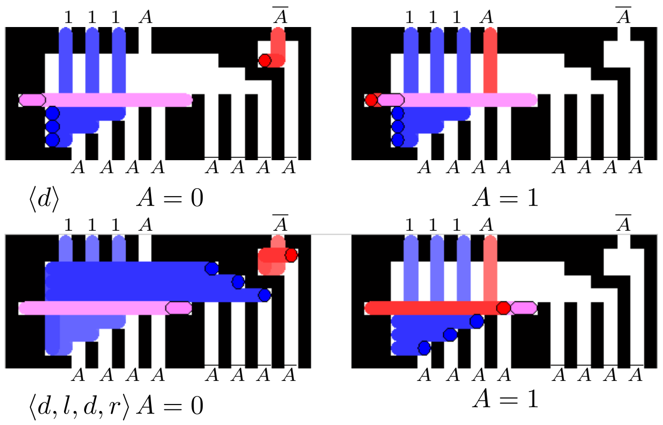

1 1 1 A $\overline{A}$
A A A A $\overline{A}$ $\overline{A}$ $\overline{A}$ $\overline{A}$
$\langle d \rangle$ $A=0$
$A=1$
$\langle d, l, d, r \rangle$ $A=0$
$A=1$

Fig. 3. The FAN-OUT gate can drive multiple outputs. Here a single input drives four-outputs. This gate requires a dual-rail input, three supply particles, and a 2×1 slider. The *clockwise* control sequence $\langle d, l, u, r \rangle$ quadruples the dual-rail input.

data storage devices, we can implement an n bit memory. To maintain *conservative* properties of the computer, i.e. the same number of robots enter and leave each gate, single-bit data storage latches must be used in pairs to record the state and its inverse.

D. A binary counter

Using the FAN-OUT gate from Section **??** we can generate arbitrary Boolean logic. The half-adder from Fig. **??** requires a single FAN-OUT gate.

We illustrate how many gates can be combined by constructing a binary counter, shown in Fig. **??**. Six logic gates are used to implement a 3-bit counter. A block diagram of the device is shown in Fig. **??**, and Fig. shows the results of each computation stage. The counter requires three FAN-OUT gates, two summers, and one carry. Six 1×1 particles

Q	R	S	C	Q	Q_R	W_1	W_2	$\overline{Q}_R$
0	1	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
1	1	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1
1	0	0	1	0	0	1	1	0

TABLE III
A SINGLE-BIT DATA STORAGE LATCH WITH STATE Q.

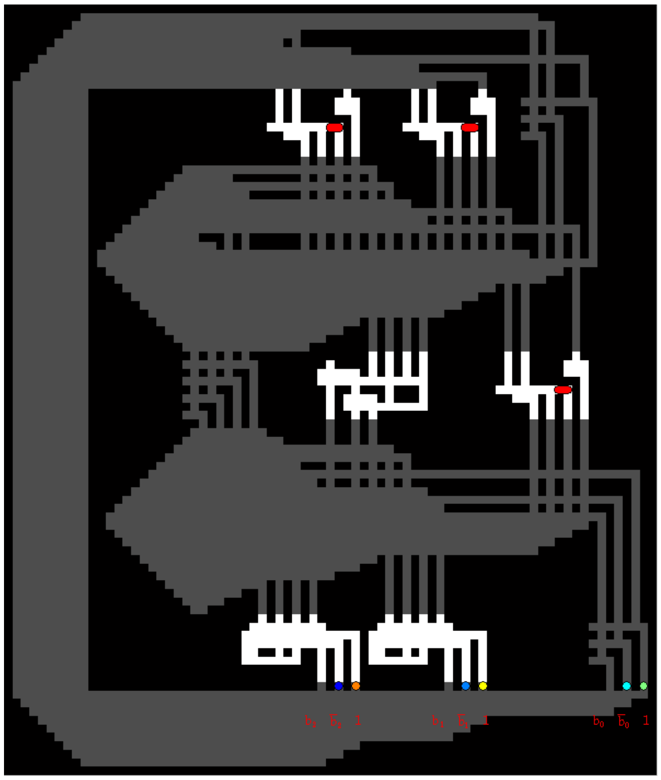


Fig. 5. A three-bit counter implemented with particles. The counter requires three FAN-OUT gates, two summers, and one carry. Six 1×1 particles and three 2×1 particles are used. The counter has three levels of gates actuated by CW sequence $\langle d, l, u, r \rangle$ and requires three interconnection sequences $\langle d, l, u, r \rangle$, for a total of 24 moves.

and three 2×1 particles are used. The counter has three levels of gates $\langle d, l, d, r \rangle$ and requires three interconnection moves $\langle d, l, d, r \rangle$, for a total of 24 moves. Figure **??** shows the ending configuration for each iteration of the counter.

E. Scaling issues

Particle computation requires multiple clock cycles, workspace area for gates and interconnections, and many particles. In this section we analyze how these scale with the size of the counter, using Fig. **??** as a reference.

d) gates: an n-bit counter requires n FAN-OUT gates, $n-1$ summers (XOR) gates, and $n-2$ carry (AND) gates.

e) particles: we require n 1×1 particles, one for each bit and n 2×1 particles, one for each FAN-OUT gate.

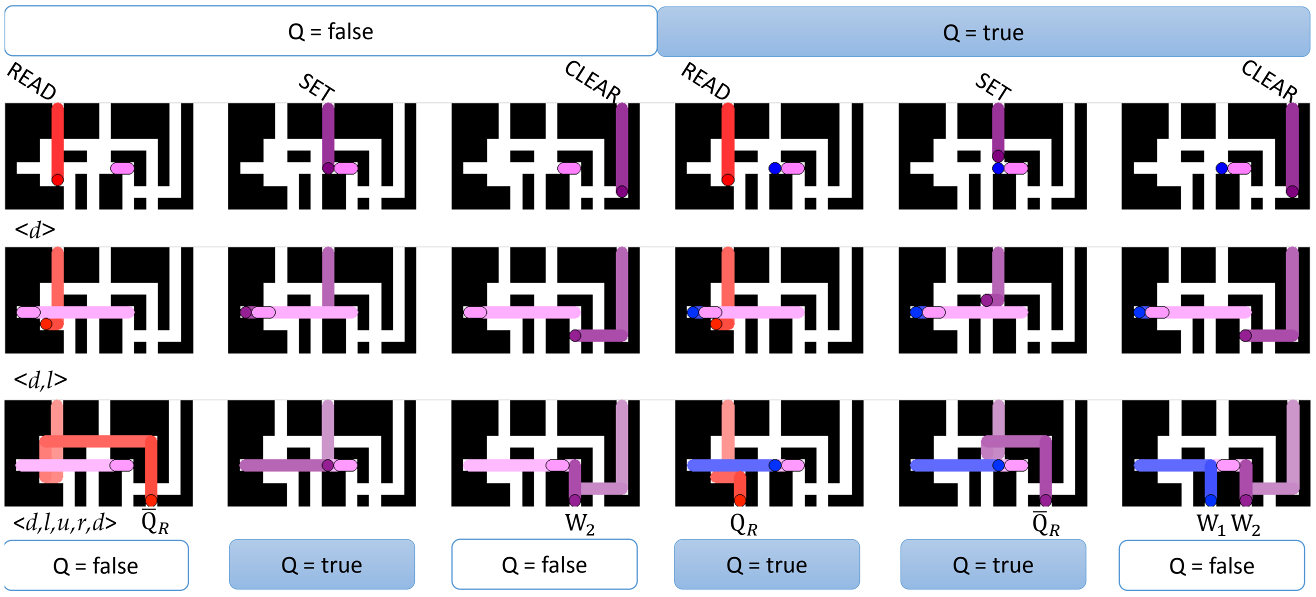


Fig. 4. A flip-flop memory. This device has three inputs, *Read*, *Set*, *Clear*, a state variable (shown in blue), and a 2×1 slider. Depending on which input is active, the control sequence $\langle d, l, d, r \rangle$ will read, set, or clear the memory.

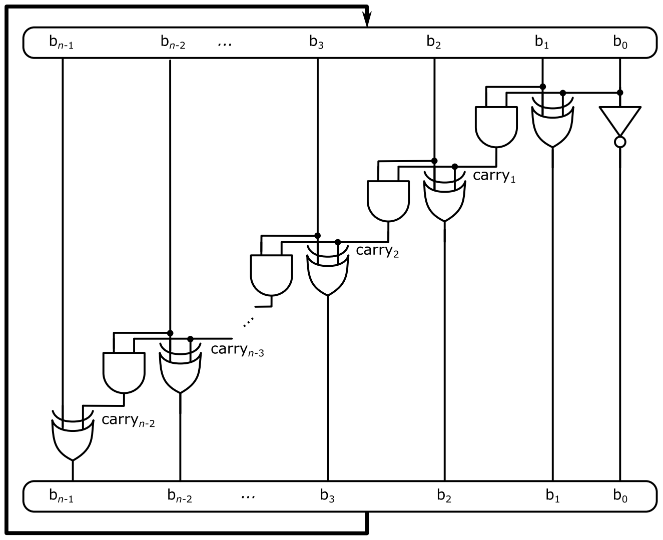


Fig. 6. Gate-level diagram for an n-bit counter. The counter requires $n-1$ XOR gates, $n-2$ AND gates, and 1 NOT gate.

f) propagation delay: the counter requires n stages of logic, and n corresponding wiring stages. Each stage requires a complete clock cycle $\langle d, l, u, r \rangle$ for a total of $8n$ moves.

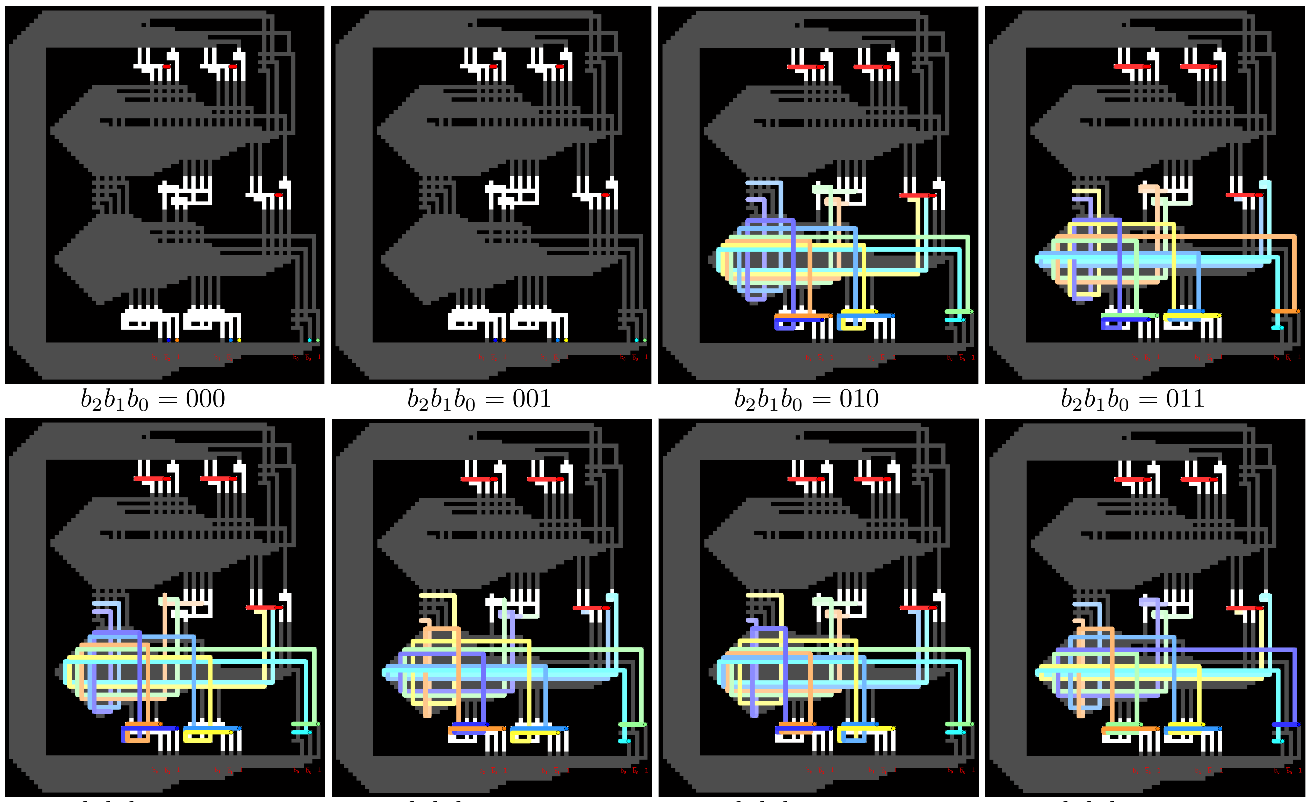
These are comparable to a ripple-carry adder: the delay for n bits is and requires $x =$ gates. Numerous other schemes exist to speed up the computation. The advantage of gates is that they are easily reused and connected. If speed was critical, instead of using discrete gates, we could engineer the workspace to directly compute logic.

F. Optimal Wiring schemes

With our current CW clock cycle, we cannot have outputs at the same column as inputs – outputs must be either one to the right, or three to the left. Choosing one of these results in shifts horizontally at each stage and thus spreads out the logic. A better wiring scheme would cycle through three layers that go right one, followed by one layer that goes left three. We also want the wiring to be tight left-to-right. If our height is also limited, *wire buses* would be a compact solution.

V. Conclusion

In this paper we...



$b_2b_1b_0 = 000$ $b_2b_1b_0 = 001$ $b_2b_1b_0 = 010$ $b_2b_1b_0 = 011$

$b_2b_1b_0 = 100$ $b_2b_1b_0 = 101$ $b_2b_1b_0 = 110$ $b_2b_1b_0 = 111$

Fig. 7. Ending configuration for each stage of the computation.