Molecular Mechanical Computing Systems

Ralph C. Merkle • Robert A. Freitas Jr. • Tad Hogg • Thomas E. Moore • Matthew S. Moses • James Ryley

Abstract. It has long been known that, in theory, computing processes can be made arbitrarily energy-efficient. The power and heat problems which plague current computers stem from the use of inefficient computing elements (e.g., electronic transistors), not fundamental principles of physics (which indicate that computations can take essentially zero energy).

While essentially all modern computers are electronic, computers can also be implemented mechanically. Power consumption in electronic computers versus mechanical computers stem from fundamentally different phenomenon. Power consumption in an electronic computer is proportional to electrical resistance, while power consumption in a mechanical computer is proportional to friction.

A new design paradigm for mechanical computers has been created which not only vastly simplifies the design of mechanical computers, but relies solely upon mechanisms with very low friction. An analysis of the potential capabilities of a mechanical computer based on this new design paradigm shows that, in a properly designed molecular-scale mechanical computer, friction consumes far less energy than electrical resistance. As a consequence, a mechanical computer designed as described herein has the potential to provide 10^{12} GFLOPS/Watt, over 10^{11} times more efficient than conventional "green" supercomputers, which currently provide about 7 GFLOPS/Watt.

Cite as: Ralph C. Merkle, Robert A. Freitas Jr., Tad Hogg, Thomas E. Moore, Matthew S. Moses, James Ryley, "Molecular Mechanical Computing Systems," IMM Report No. 46, April 2016; http://www.imm.org/Reports/rep046.pdf

Table of Contents

Mechanical Computing Background

Methods for mechanical computation have been known for centuries. Simple examples include function generators and other devices which are not capable of general purpose (Turing-complete) computing, as well as those capable of general purpose computing. For review of various types of mechanical computing devices, see [1].

The earliest example of a general purpose computer (which is referred to as "Turing-Complete," and requires both combinatorial and sequential logic) is probably Babbage's Analytical Engine (https://en.wikipedia.org/wiki/Analytical_Engine), which was described in 1837. However, with the advent, and subsequent rapid progress, of electronic computing, mechanical computing became largely relegated to a curiosity.

While previous designs for mechanical computing vary greatly, the few designs capable of general purpose computing require a substantial number of basic parts, such as various types of gears, linear motion shafts and bearings, springs (or other energy-storing means), detents, ratchets and pawls, and clutches. These mechanisms dissipate substantial energy due to friction.

The use of many parts may also increase the mass and complexity of the devices, thereby making them more difficult to manufacture, and potentially requiring the devices to operate at lower frequencies than would otherwise be necessary. However, reducing the number, complexity, and mass of parts in a mechanical computer is not a simple task due to the need to provide the sophisticated functions of both combinatorial logic (e.g., AND, NAND, NOR, etc.) and sequential logic (memory).

Sequential logic in particular, being the basis for memory, requires the ability to conditionally couple and decouple parts of a computer from the current inputs. This is because memory cannot be required to be a deterministic result of just the current inputs, otherwise all previous information is erased. Storing information, which is easily accomplished in electronic systems using latches or flip-flops, is not as easily accomplished in a mechanical system which may have to actually connect and disconnect parts of the system from each other (e.g., using a clutch-like mechanism) at appropriate times.

Issues such as these are presumably among the reasons for the relatively complex designs of existing mechanical computational systems. This is also perhaps the reason that electronic computing eventually became the standard, even though designs for mechanical computers pre-dated the advent of the transistor (or vacuum tube).

This raises the question as to whether it is possible to substantially simplify the design of mechanical computing systems, and if so, what benefits would accrue.

Computing With Only Two Parts

Our work shows that existing designs for mechanical computing can be vastly improved upon in terms of the number of parts required to implement a complete computational system. Only two types of parts are required: Links, and rotary joints. Links are simply stiff, beam-like structures. Rotary joints are joints that allow rotational movement in a single plane.

Perhaps counterintuitively, no type of clutch-like mechanism is required. All parts of the system can remain permanently connected and yet still provide all necessary combinatorial and sequential logic.

4-Bar Linkages

To demonstrate how this can be accomplished, we start with a simple, well-known mechanism, the 4-bar linkage (also referred to as a 3-bar linkage – the 4th bar is provided by the anchor block or base and is sometimes ignored for naming purposes). The 4-bar linkage relies only upon links and rotary joints (see Figure 1).

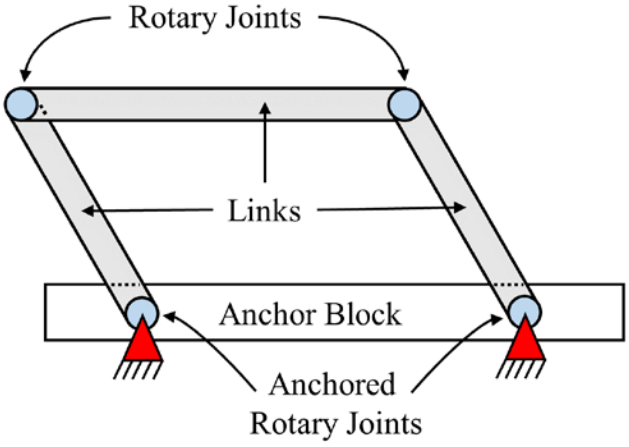

Figure 1: 4-Bar Linkage, Left-leaning

A 4-bar linkage can rotate around its anchored rotary joints (denoted by a circle with a triangle, while unanchored rotary joints are just circles), allowing, for example a "left-leaning" configuration to rotate into a "right-leaning" configuration (see Figure 2).

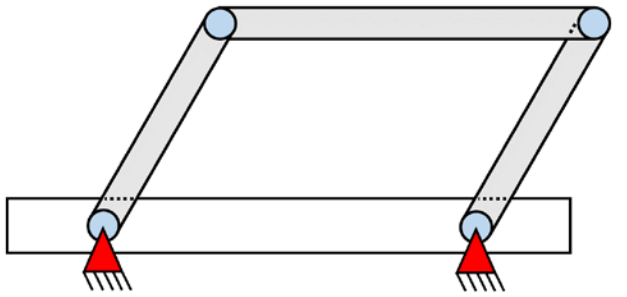

Figure 2: 4-Bar Linkage, Right-leaning

Note that, while a 4-bar linkage could assume many positions, we focus on the use of two distinct positions. This is because two positions can be used to signify 0 and 1, which is

convenient when creating a system for binary computing. The actual angle traversed by the links when moving from, e.g., left-leaning to right-leaning is not critical. As diagrammed, it is approximately 45 degrees, but could be more or less as long as the design supports reliably differentiating between two positions; one representing 0, and the other representing 1. Subsequently, examples will be shown that use other positions.

It is important to note that in a parallelogram-type 4-bar linkage (meaning, the two side links must be the same length) if an additional link is added to the center of the linkage, as long as the length and angle is the same as the side links, the linkage will still rotate around the anchored rotary joints (see Figure 3).

However, if the additional link is not the same length, or is not at the same angle, as the existing side links, the mechanism will not move (see Figure 4). This is because the additional link will be attempting to pivot through a different arc than the side links. The effect of this is that the side links and the center link are trying to move the top link in two different directions at once. This results in the mechanism binding up, or "locking." As will be described, this is actually a desirable property.

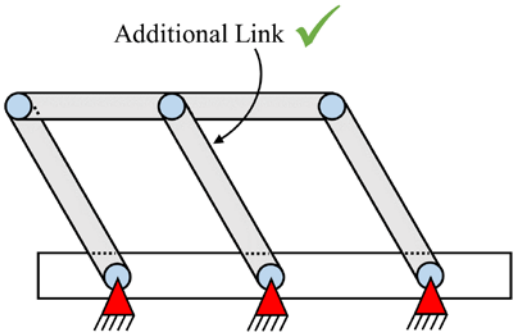

Figure 3: Mobile Linkage

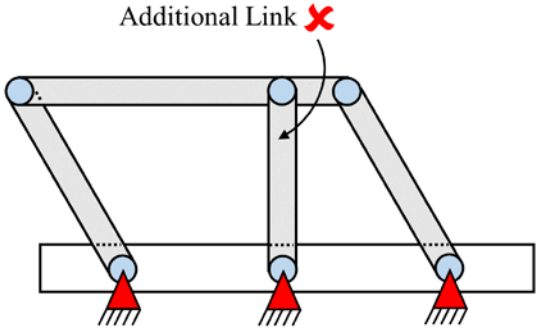

Figure 4: Non-mobile Linkage

The "Lock"

A "Lock" is a mechanism composed of two 4-bar linkages, rotated 180 degrees with respect to each other, and connected in the center via a connecting link (see Figure 5). The connecting link is the same length as the two side links of each 4-bar linkage, and, in the starting position, is parallel to all four side links.

The connecting link is held on the top and bottom side by two extra links, each pair of which form a rigid triangle with the rest of the respective 4-bar linkage. This triangular projection allows the connecting link to be the same length as the side links. If the triangular projection

were not present and so the connecting link had to connect directly to the horizontal links of the 4-bar linkages, it would not be the same length as the side links. As already mentioned, this would cause the 4-bar linkages to bind up, or "lock" (hence the name). To function in the intended manner, all of the vertical links attached to the horizontal links must be the same length and must, at least initially, be parallel to the other vertical links.

As previously noted, linkages need not have their two positions be left-leaning and right-leaning. Rather, for binary computing purposes, they just need to have two distinct positions which can represent 0 and 1. Figure 5 uses the convention that an input of "0" results in the side links being straight up, while an input of "1" would cause them to lean to the right.

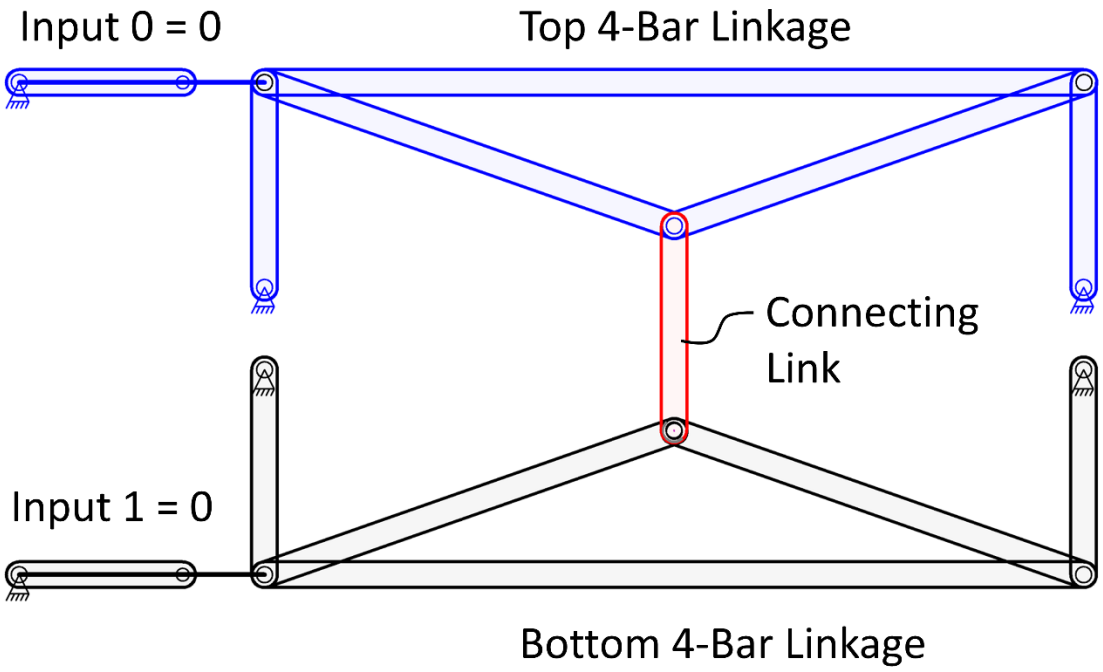

Figure 5: A lock in the (0,0) position

Representative inputs have been added to Figure 5, to show the direction of actuation and where input could be connected. It is assumed that there is one input for each 4-bar linkage, or one input each for the "top" and "bottom" of the lock. Note that the inputs are depicted as linear slides that can push one of the 4-bar linkages from the 0 position to the 1 position. In an actual system, the inputs would be connected to other parts of the system (e.g., data from memory, or clock signals, would normally provide actuation). Linear slides are not required.

The lock position depicted in Figure 5 would be called the (0,0) position, referring to the state of the inputs. See Figure 6 for a depiction of the (1,0) state, where Input 0 has been set to 1 and thereby pushed its respective 4-bar linkage into the right-leaning configuration. Flipping the figure upside down would represent the (0,1) state.

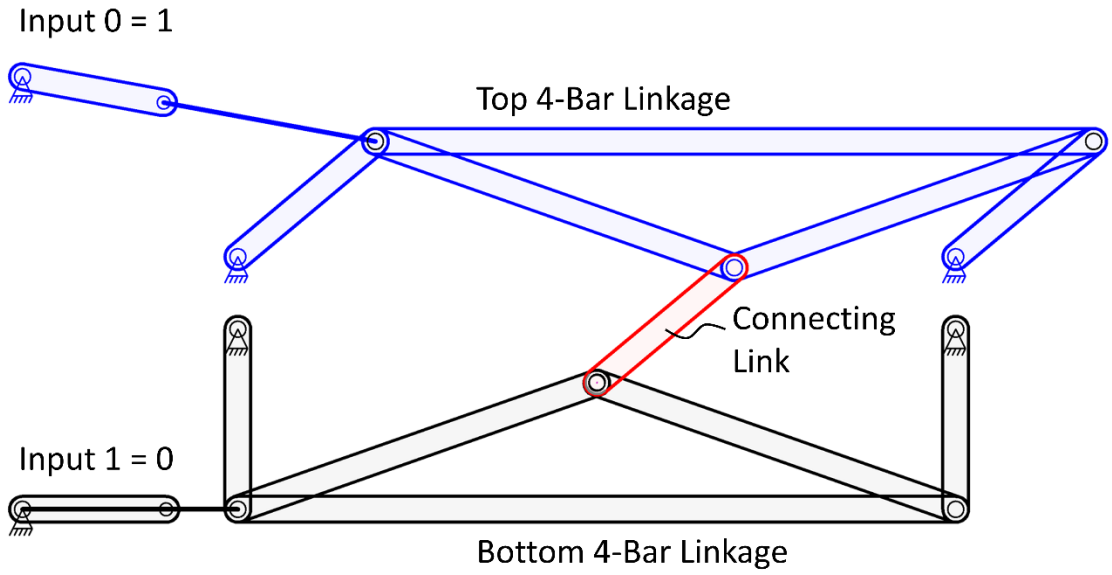

Figure 6: A lock in the (1,0) position

The (1,1) state is not possible, and this is a key aspect of a lock. Once one of the inputs has been set to 1, one side of the lock rotates around both its anchored rotary joints, and its connecting link. Crucially, while the connecting link never changes length, it does change angle. As depicted in Figure 6, once the lock moves into the (1,0) position, the connecting link is still parallel to the side links of the top 4-bar linkage, but is no longer parallel to the side links of the bottom 4-bar linkage. Due to the requirement that, if a 3rd central link is present, this link be both the same length and at the same angle as the side links, once either the top or bottom 4-bar linkage moves, the other cannot; it is "locked."

It is for this reason that the (1,1) position is impossible. Once the lock has moved from (0,0) to (1,0) or (0,1), the only possible movement is back to the (0,0) position. Once back to the (0,0) position, either input could be set to 1, but both inputs can never be set to 1 at the same time.

The lock is one of two key mechanisms on which link- and rotary-joint computing is based. The other is the balance.

The "Balance"

A balance, so named because it is superficially similar to a classic pan balance, connects an input to a link which has three rotary joints: one at each end, and one in the center. The input is connected to the center rotary joint. The two end rotary joints are connected to other structures, most commonly, locks (and more specifically, generally a set of locks where one of the locks is locked, while the other is not).

The result is that, when the balance input is changed, one of the side rotary joints remains stationary, while the other moves. Which side is stationary and which side moves can be determined by data inputs.

How this works in practice is best illustrated by example. Figure 7 depicts a balance (green) connected to two locks (blue and gray). While the diagrammatic representations have been simplified slightly, it will be obvious that each lock corresponds to the lock mechanism depicted in Figure 5.

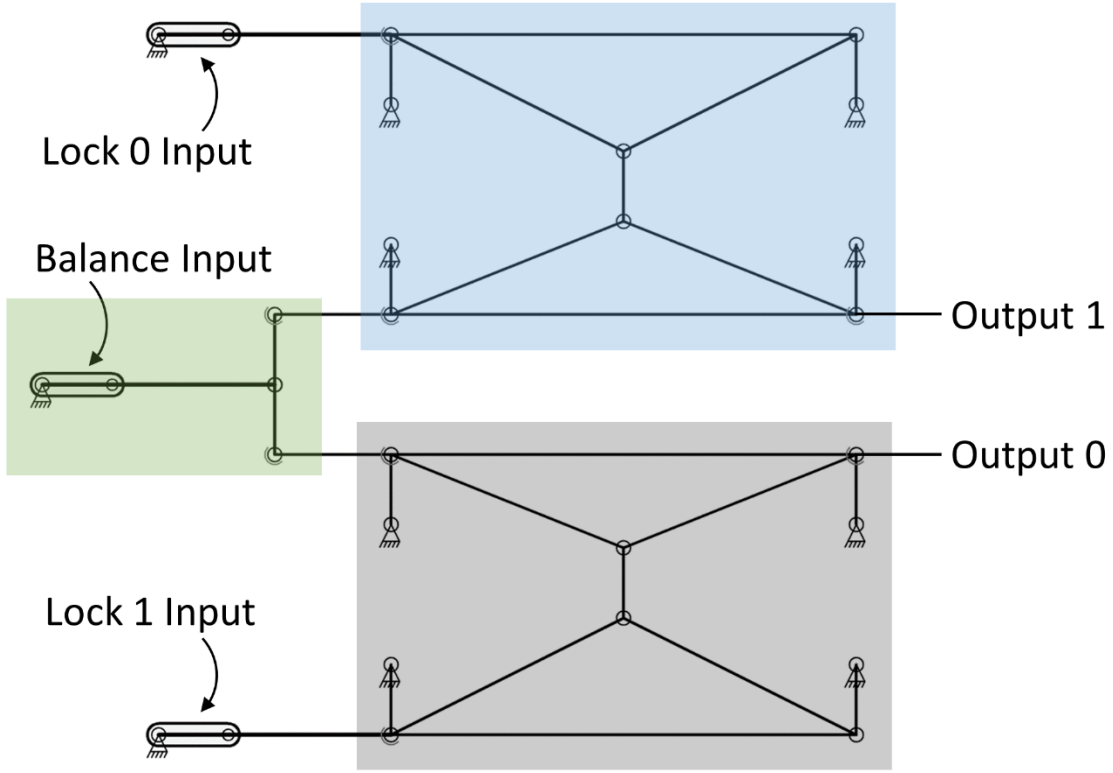

Figure 7: A balance coupled to two locks

The only difference between the locks in Figure 7 and the lock in Figure 5 is that in Figure 7, rather than having two individual inputs, each lock has one input that is specific to a given lock, and the other input is supplied indirectly by the balance, which also has an input.

Note that input to balances will often be supplied by a clock system. It is worth mentioning that, like conventional computers, a multi-phase clocking system is required for the described computing system. Such a clocking system is assumed to be present; describing how to implement such a system is beyond the scope of this document.

There are two main rules governing the mechanism shown in Figure 7, and these rules are enforced by the overall system design; they are not inherent in the mechanism depicted:

- Either the Lock 0 Input, or the Lock 1 Input, but never both, must be set to 1.
- The lock inputs and the balance inputs must be set sequentially, not simultaneously.

Given these rules, the operation of the mechanism is as follows. Each step could be thought of as a different clock phase (which implicitly enforces the second rule above):

1) The mechanism starts with all inputs set to 0.
    - This means that neither lock is locked.
2) Either the Lock 0 Input is set to 1, or the Lock 1 Input would be set to 1.
    - This results in locking one of the locks.
    - Locking one of the locks constrains which side of the balance can move.
3) The balance's input is set to 1.
    - Since only one side of the balance is free to move at this point, the balance input is transmitted down the path which is free to move.
    - This produces an output of 1 at either Output 0 or Output 1, depending on which lock input was set to 1.

This provides one example of performing simple logic that can, for example, be used to route data, shunting inputs down one path or another based on the states of Lock 0 and Lock 1. For this reason, this mechanism may be referred to as a switch gate.

Note that this simple logic and conditional routing has been accomplished using only links and rotary joints, which are solidly connected at all times. No gears, clutches, switches, springs, or any other mechanisms are required (keeping in mind that the input mechanisms shown are representative; an actual system does not require linear slides).

Universal Combinatorial Logic

Although the previous example could be thought of as performing simple logic, it is perhaps more useful for routing data. The mechanism in Figure 7 cannot provide all the logic necessary for a complete computational system. However, it is possible to create all necessary logic using nothing but locks and balances (and a few extra links and rotary joints to route and/or copy data).

Any traditional 2-input logic gate, including AND, NAND, NOR, NOT, OR, XNOR and XOR, can be created directly from the appropriate combination of locks and balances. While most of these gates are illustrated in the Appendix, there is no need to address each in detail to prove that universal logic can be created using links and rotary joints. This is because it is well-known that NAND alone suffices to create all necessary combinatorial logic (i.e., all other logic can be created from combinations of NAND gates[2]). Therefore, as a proof of the fact that links and rotary joints suffice to create the combinatorial logic required for a Turing-complete computing system, Figure 8 shows how a NAND gate can be implemented.

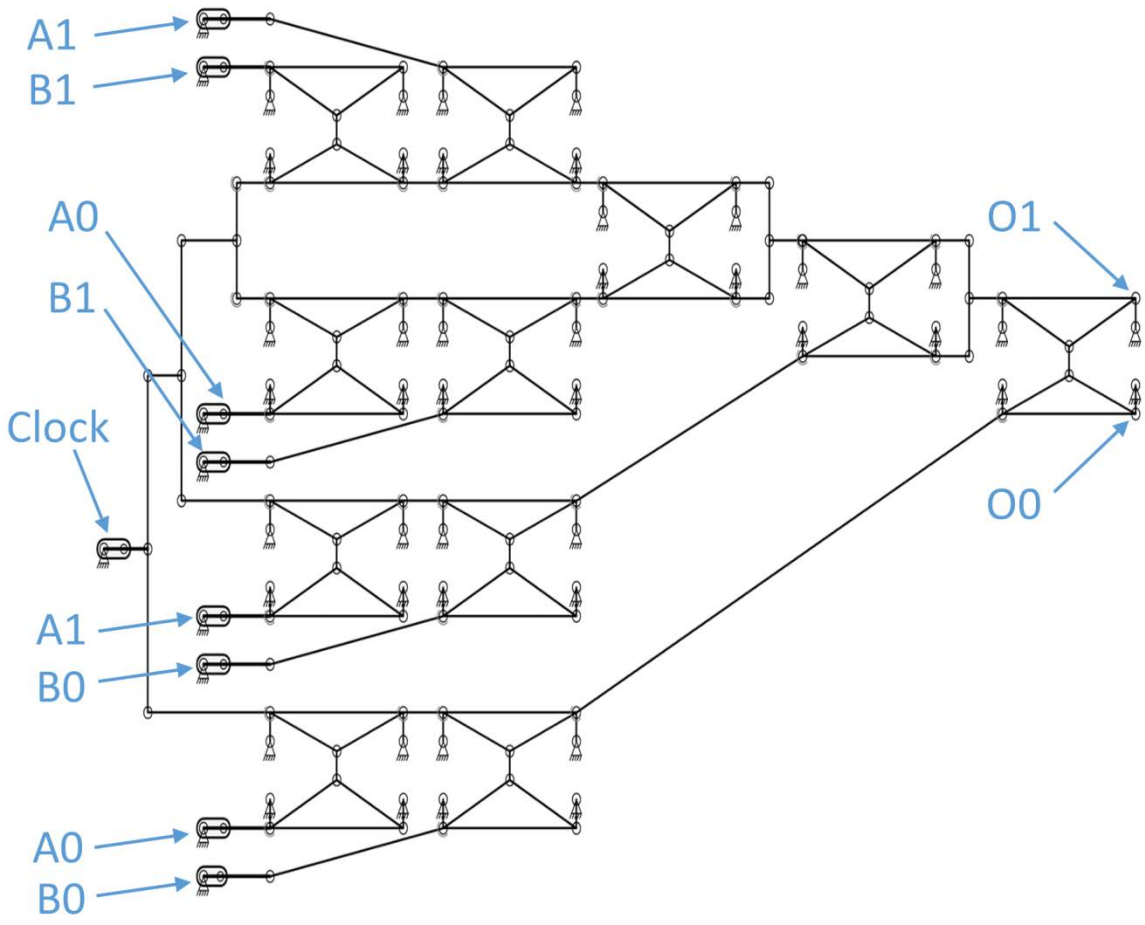

Figure 8: A balance- and lock-based NAND Gate, using a two-link per bit input/output design, and duplicated inputs for diagrammatic clarity.

While substantially more complex than the previous example, the NAND gate functions on the same principle of using locks and balances to implement logic. In this example, a set of inputs are connected to a set of locks. The inputs determine which side of each lock is locked. Another input (a clock signal in this example) is then used to actuate a main balance, the movement of which cascades through a series of additional balances and locks. Each balance moves either its top side or its bottom side, in accordance with the state of the locks to which it is connected. This results in a final output lock either having its top half or its bottom half move forward.

In Figure 8, conceptually there are two binary inputs, A and B, and one binary output, O. The physical model used for the inputs and output is that a single 1-bit input or output is broken into

two separate input or output links (we will refer to this as the “two-input system”). For example, the A input is composed of inputs A0 and A1. If the value of A is to be set to 0, the A0 input moves. If the value of A is to be set to 1, the A1 input moves. A0 and A1 are not permitted to move simultaneously (which is logically obvious since a value cannot be set to 0 and 1 at the same time). The same applies to the input B and output O.

Note that each input is used twice to implement the desired logic, and so each input is represented twice in the diagram. While duplicating inputs in this manner is convenient for diagrammatic purposes, in an actual system it is simple enough to use a link and rotary joint structure that forks an input line, effectively copying the data to multiple locations as needed. There need not be actual duplicate inputs.

Similarly, the geometry of the links that connect the locks and balances is not what would be optimal in an actual system, but rather the use of straight, non-crossing lines has been favored for diagrammatic clarity.

Although complex, the outputs that will result from any allowable set of inputs can be deduced by tracking which side of each lock locks, and which side of the balances move. It can be seen that the truth table replicates that expected from a NAND gate:

Inputs						Output		
A	A1	A0	B	B1	B0	O	O1	O0
zero	0	1	zero	0	1	one	1	0
zero	0	1	one	1	0	one	1	0
one	1	0	zero	0	1	one	1	0
one	1	0	one	1	0	zero	0	1

Note that this is just a single example, used as a proof due to the well-known universal nature of NAND. Using multiple NAND gates to create other logical functions may not always be efficient, and as previously mentioned, we have also shown that any of the standard 2-input logic gates can be implemented directly using locks and balances (see Appendix: Additional Traditional Logic Gate Designs).

Sequential Logic

Having demonstrated that all necessary combinatorial logic can be created using only links and rotary joints (assembled into locks and balances), we turn to sequential logic. The outputs of the NAND gate depicted in Figure 8 are dependent solely upon the current inputs. Such a mechanism provides no way of storing previous inputs or the results of previous logical operations; it provides no means for creating memory. To demonstrate the creation of a simple

memory mechanism, we describe the design of a small shift register, again, only using links and rotary joints.

A shift register can be built by combining locks and balances to create “cells” which are the logical equivalent of electronic flip-flops. Each cell of a shift register is related to its neighbor by virtue of relying upon a preceding or succeeding clock phase, as appropriate. This enables the copying and shifting of data through the shift register, rather than deterministically setting the contents of the entire shift register simultaneously.

A Shift Register Cell

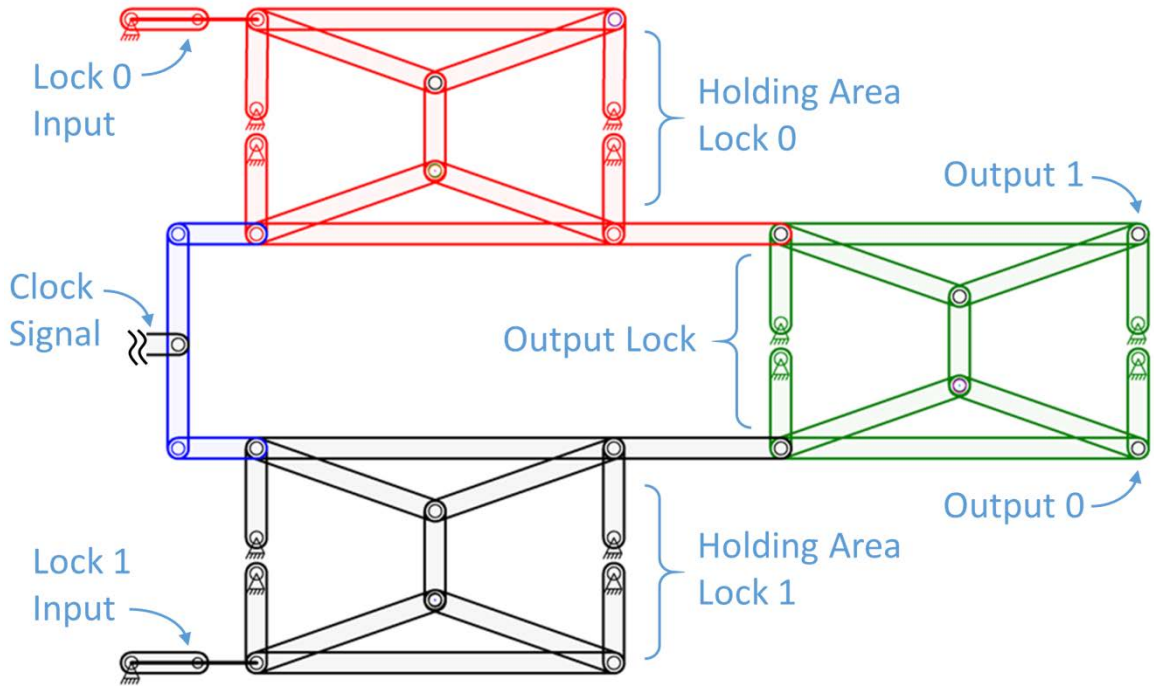

Figure 9: Shift register cell in the (0,0) blank state

Figure 9 depicts a single shift register cell with an input of (0,0), or what may be referred to as the blank state. This state is often to be avoided, and at times, logically impossible. This is because, using the two-input system, an input of zero is represented as (1,0), while an input of 1 is represented as (0,1); there is no (0,0) or (1,1). However, starting from the blank state can be useful to demonstrate how cells function, and may be required when cells are initialized.

Before delving into the actual use of shift register cells, a brief description of the major parts of a cell is in order. First, it is notable that the left-hand portion of a cell is identical to the switch gate of Figure 7, consisting of a balance which is connected to a top lock (“Holding Area Lock 0”) and a bottom lock (“Holding Area Lock 1”). The balance is actuated by a clock signal, again, just

like Figure 7. The only difference is the addition of an extra lock, the "Output Lock," which is connected to the outputs of the left side of the mechanism and in turn provides the final output for the cell. The easiest way to visualize how a cell works is to step through the movements.

Starting from the blank state, an input is set. Since this mechanism uses the two-input system, either the Lock 0 input is set to 1, or the Lock 1 input is set to one, but not both. If we assume that Lock 0 is set to 1, the movement results in the mechanism being in the state depicted in Figure 10.

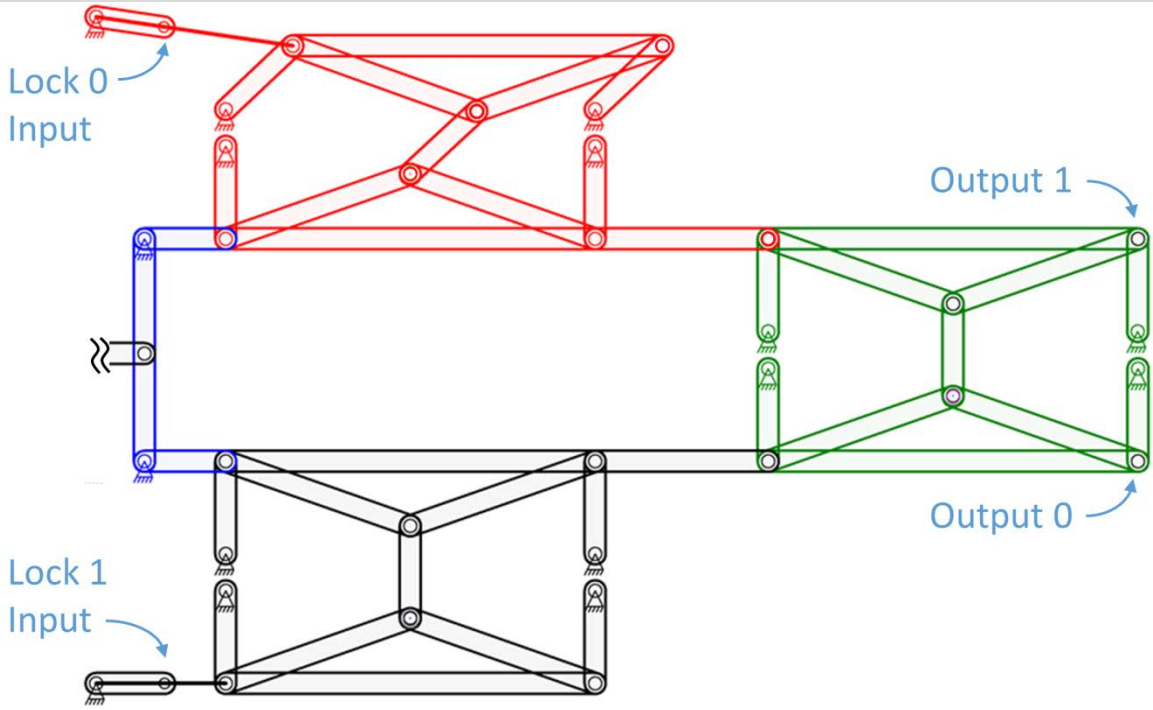

Figure 10: Shift register cell with input (1,0), prior to clock actuation

In Figure 10, Holding Area Lock 0 has moved its upper side. This results in the lower side of that lock locking. However, since shift registers are dependent upon multi-phase clocking, note that the input at Lock 0 has not yet had any effect on the Output Lock. For that to occur, the balance must be actuated, which does not occur until the next clock phase.

During the next clock phase, the balance is actuated (set from 0 to 1, in this case). Since Lock 0 has locked its lower half, which is connected to the upper side of the balance, upon actuation, the balance can only move its lower side. This movement propagates to the Output Lock, resulting in the state depicted in Figure 11.

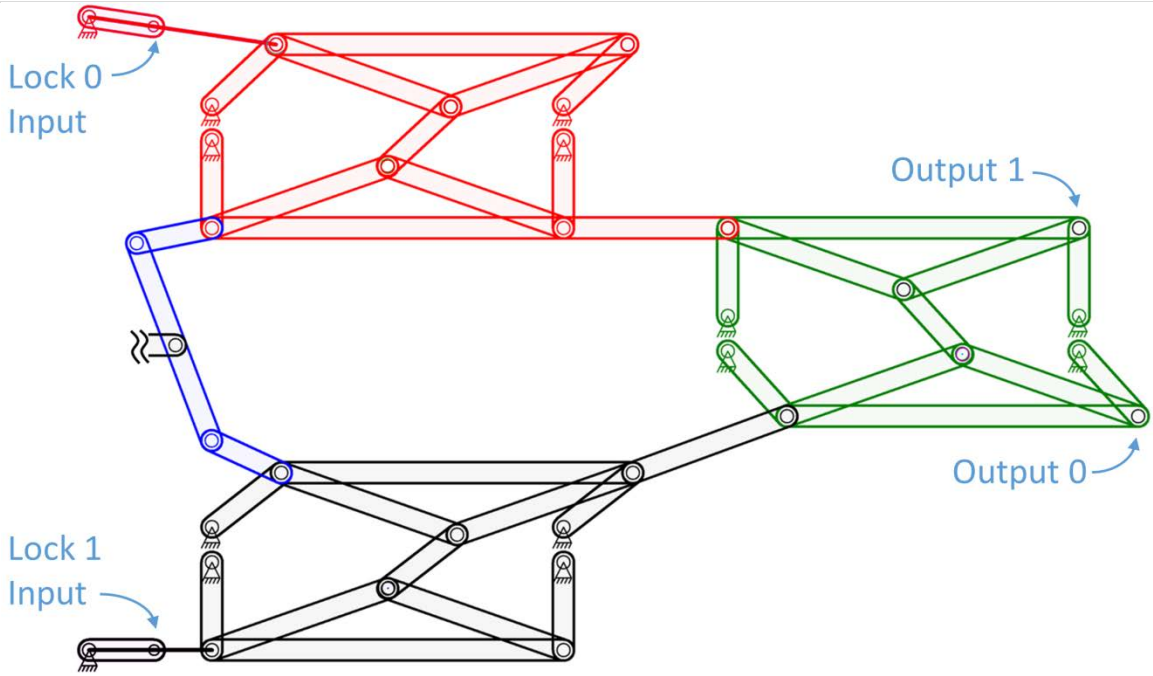

Figure 11: Shift register cell with input (1,0) after clock actuation

The reason the left two locks are referred to as “Holding Area Locks” may now be apparent: They temporarily hold the input data prior to clock actuation. An input to these locks does not instantly result in an output at the output lock. Rather, the clock must actuate first, which results in copying the value, be it 0 or 1, from the holding area locks to the output lock.

Note that we adopt the convention that, with respect to the input locks, the top lock is associated with an input of 0, while on the output lock, the top half is associated with an output of 1. This is because, due to how the mechanism works, when an input of 1 is set at the top input lock, an output of 1 ends up at the bottom half of the output lock, and vice-versa. The mechanism could be easily altered to change this, but as it is currently represented, from a naming perspective it is easiest to have the 0 input result in a 0 output, and the 1 input result in a 1 output.

Thus far, we have only needed two clock phases: On the first phase, the inputs are set, and on the second phase, the balance actuates and the inputs are copied from the holding locks to the output lock. However, in an actual system, additional phases would be used. The subsequent example assumes 4-phase clocking.

Connecting Cells

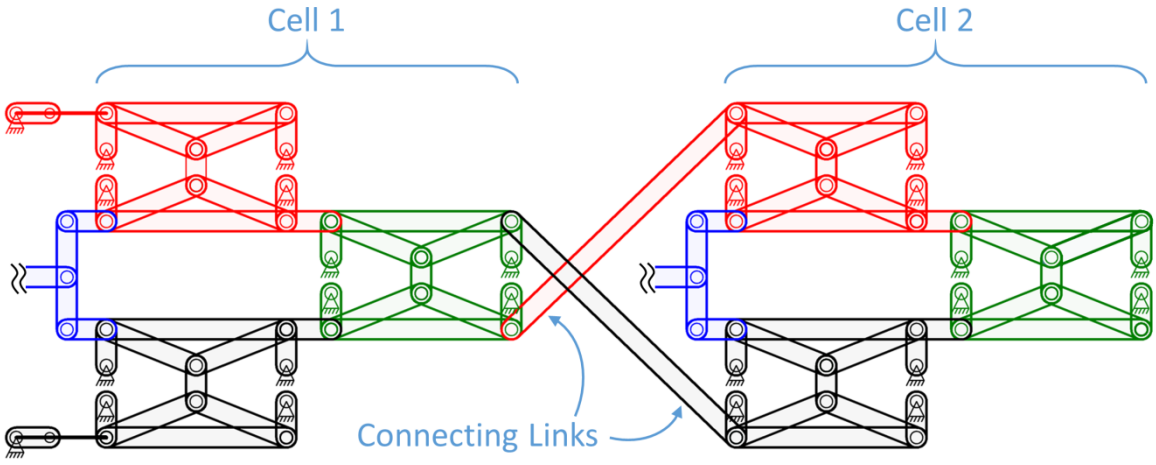

Figure 12: A Two-Cell Shift Register

Figure 12 depicts a two cell shift register to illustrate how two cells would be connected and to explain how data would move from one cell to the next. In this figure, Cell 1 and Cell 2 are each equivalent to the mechanism depicted in Figure 9 - Figure 11. Both cells have a connection to a clock signal (depicted as a partial link to indicate connection to a clock system that is not shown). The connecting links connect the output from Cell 1 to the inputs to Cell 2.

The operation of a single cell has already been described. Now, demonstrating how Cell 1 passes data to Cell 2 will illustrate the function of a minimal shift register. The sequence of events is as follows:

1) As phase 1 begins, the clock input for Cell 1 is presumed to have already been set to 0. During phase 1, the data inputs are set for Cell 1.
    - Either the upper or lower lock of Cell 1 locks, depending on which input was set to 1.
2) During phase 2, the clock signal for Cell 1 is set to 1.
    - This results in the unlocked side of the Cell 1 balance moving, which in turn moves the upper (if the input value was 1) or the lower (if the input was 0) half of the output lock.
    - Cell 1's output lock in turn moves the appropriate connecting link, locking one of Cell 2's holding area locks. This has effectively copied the data from Cell 1's output lock into one of Cell 2's holding area locks.
    - Note that the output lock of Cell 2 still has not moved.
3) During phase 3, the clock signal for Cell 2 is set to 1.
    - This copies the data from Cell 2's holding area locks into Cell 2's output lock.
4) During phase 4, the final phase, the clock signal for Cell 1 is set back to 0.

- As a result, Cell 1's output lock, and hence the connecting links to Cell 2, retract to the 0 position (regardless of the values still stored in Cell 1's holding area cells). This unwrites Cell 1's data using the data from Cell 2's holding area.

This cycle then repeats itself as new data is input into cell 1. In step 2 above, it is noted that the output lock of Cell 2 still has not moved. This behavior allows shift register cells to store previous data. This is a key difference between combinatorial and sequential logic. The state of the NAND gate described previously is completely determined by the current data inputs. However, this is not true of the cells of a shift register. A cell can contain not only a previous input (which ends up being shifted to its output lock during subsequent phases) but also the current input, which is stored in the holding area locks. As a result, the state of the shift register is a function of both the current inputs and the previous inputs. This allows the mechanism to act like a mechanical analog of an electronic flip-flop, thereby forming the basis for memory storage.

To aid in the visualization of how such a mechanism works, an animated model has been prepared using AutoDesk. See Figure 13.

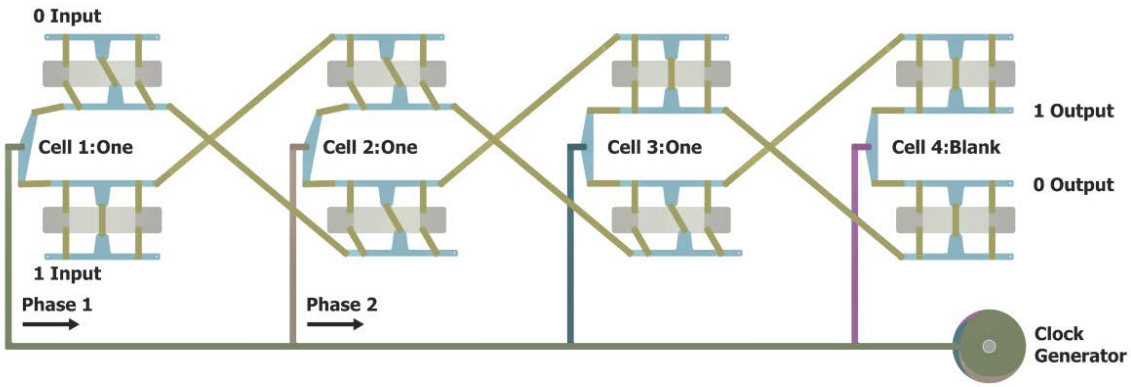

Figure 13: A working 4-cell shift register (click for animation)

Summary: Computing with Only Two Parts

We have demonstrated that, using only links and rotary joints, universal combinatorial logic and sequential logic can be created. Universal combinatorial logic has been demonstrated with the design of a NAND gate, while sequential logic, mimicking electronic flip-flops and sufficient to create memory, has been demonstrated using cells and by combining those cells into shift registers.

It is well-known that universal combinatorial logic and sequential logic together suffice to create a general purpose, or Turing-complete, computational system. Meaning, subject to practical limits of time and memory (as in any computer), such a system can compute anything that can be computed.

One might wonder why anyone should care about these findings. While certainly interesting from a theoretical perspective, what practical problems can be solved by realizing that general purpose computing can be implemented with such simple parts and mechanisms?

Why Mechanical Computing?

Mechanical computers have remained curiosities at least in part because in previous designs for mechanical computers, friction wastes more energy than is wasted by electrical resistance in an electronic computer. There is little value in pursuing mechanical computing if it is clearly inferior to electronic computing. However, the design for mechanical computing described above, by using only rotary joints as the basis for movement, keeps friction very low.

One everyday example of the efficiency of rotary joints is the Vee Jewel bearings used in watches. In general, the smaller the point of contact in such a bearing becomes, the less energy is lost to friction. And, as small as watch movements are, they are enormous compared to molecular implementations of rotary joints. At the molecular scale, properly designed rotary joint-based logic can be much more energy-efficient than comparable electronic mechanisms.

Molecular-Scale Implementation

In the section "Computing With Only Two Parts," we described the general mechanical principles behind link- and rotary-joint based computation. We now address a particular implementation. Specifically, we describe an implementation where the basic parts are molecular-scale.

Atomic Bonds as Rotary Joints

Molecular-scale bearings, rotors, and motors are well-known in the literature [3-8] and one of the smallest implementations of a rotary joint could pivot around single atomic bonds. In Figure 14, an upper housing and a lower housing are shown, connected via acetylenic bonds to a central rotor. The rotor and housing are essentially diamond, although some oxygen atoms have been used (in red) where different valences were appropriate. Computational chemistry simulations indicate that the rotor in a structure like that in Figure 14 rotates with very little friction.

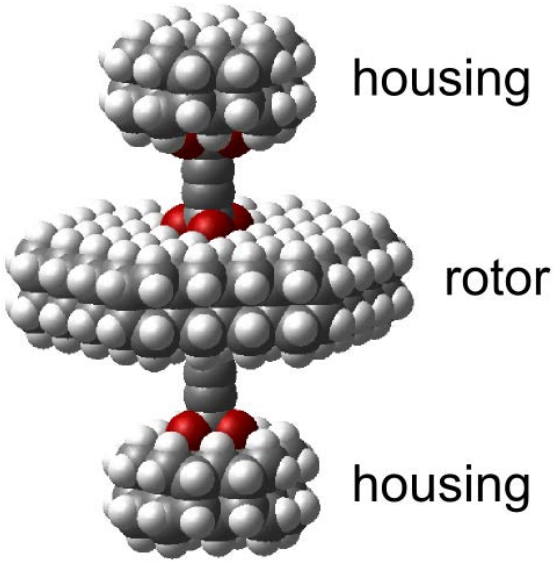

Figure 14: A rotary joint using two atomic bonds

Note that in Figure 14, the number of atoms has been minimized to make the structure amenable to detailed simulation. In an actual system, the housings would be larger (potentially scaffolds that surround the entire system, providing anchor points for any rotary joints requiring them), and the rotor might be a more massive flywheel, or a link (e.g., a rod-like piece of diamond, or a carbon nanotube).

Friction in Single-Bond Rotary Joints

In theory, rotation around atomic bonds themselves has no friction. But, due to the thermal motion of individual atoms, some rotational motion will be converted into, e.g., lateral or tilting motions, resulting in a small amount of friction or drag.

A system composed of just links and rotary bonds provides 1.28×10^{12} GFLOPS/Watt (considering only frictional losses and ignoring the Landauer Limit, which is addressed later; see Appendices for computations). The most efficient conventional supercomputers as of August 2015 provide 6-7 GLFOPS/Watt.[9] In other words, the proposed system is, ideally (caveats to this figure are discussed later), 1.8×10^{11} times more efficient.

This should perhaps not be too surprising when we consider that (1) there is no known theoretical limit to the energy efficiency of reversible computation and (2) conventional computers are not fabricated via molecular manufacturing. To the second point, our designs for

a mechanical computer might more properly be compared with other proposals enabled by molecular manufacturing (e.g., rod logic[15]).

Now that the frictional dissipation of the proposed link- and rotary-joint-based computers has been computed, we return to the question of why such computers are of more than theoretical interest. The reason is efficiency: Electrical systems are subject to power dissipation from electrical resistance. Mechanical systems are subject to power loss from friction. By avoiding both electrical resistance (by simply not being electrical), and the major sources of mechanical friction (e.g., surfaces sliding over one another, by using only rotary joints), our proposed mechanical computational systems can be designed to dissipate far less energy than any existing method of electrical computing.

Note that the efficiency of our proposed mechanical computers increases as the operating temperature is lowered. This is for several reasons. For example, reduced thermal motion of the atoms which form the rotary joints reduces friction. And, reduced thermal motion decreases the positional uncertainty of the links, allowing shorter distances between the 0 and 1 positions without errors occurring. At very low temperatures, advantageous quantum effects may also aid in decreasing energy dissipation. Computing exactly how much more efficient a mechanical system can be at lower temperatures is certainly a topic of interest, but for now, it is enough to realize that even at moderate temperatures, the performance increase of link- and rotary-joint-based computers as compared to electrical computers can be striking.

Links

Little has been said about links, as opposed to rotary joints. This is because links have no moving parts. The only requirement of links is that they be stiff enough for the mechanisms to work as intended. And, given the extremely high strength of materials like diamond and carbon nanotubes, unparalleled stiffness can be obtained even at the molecular scale – far in excess of the everyday materials with which we are familiar.

An Atomic-Scale Lock

Figure 15 and Figure 16 provide an atom-by-atom example of how diamond can be used to create a lock from links and rotary joints. Note that the familiar “triangle” structure which holds a lock’s connecting link is solid in this model, rather than being composed of three beams. Regardless, the function is identical. The left and right-side pointed structures (vertically in the center) to which the side links attach would be part of a larger structure, providing an anchor point.

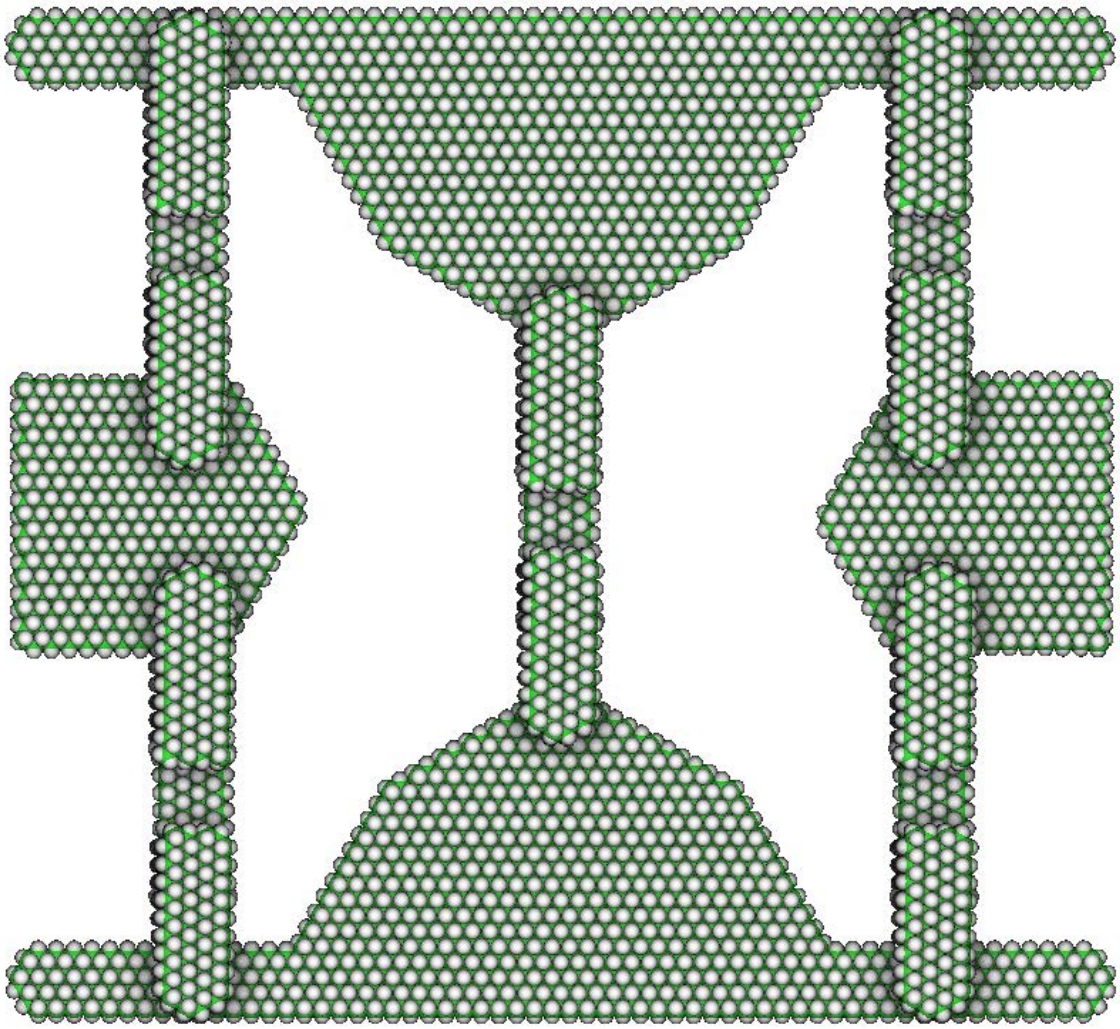

**Figure 15: A molecular model of a diamond-based lock, top view.
Hydrogens are white, Carbons are green.**

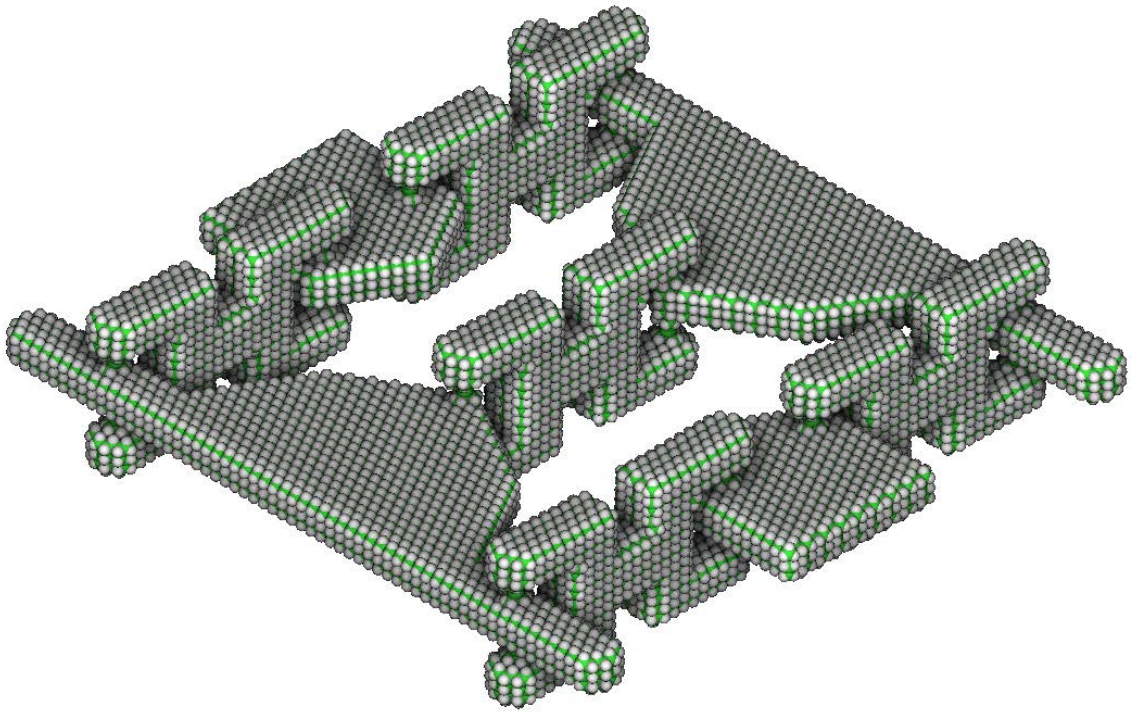

Figure 16: A molecular model of a diamond-based lock, ¾ view

Summary: Why Mechanical Computing?

In "Computing With Only Two Parts" we described a design for mechanical computing far simpler than any previously known, requiring only rotary joints and links. But, that left the question as to why such a finding was of more than just academic interest.

With the description of how rotary joints can be made to rotate around individual atomic bonds with very little friction, and using molecular-scale links as connections between rotary joints, we have shown that the practical applications of this new mechanical computing paradigm are quite important: Energy expenditure can be decreased greatly as compared to any other known system for computing.

However, to be fair, we have thus far ignored an important aspect of energy dissipation during computing: The Landauer Limit. It is to that topic that we now turn.

Limits of Computing Efficiency

For decades it has been known that the only true limits on computational efficiency are those posed by the need to expend energy to erase data [10-12]. It is not actually computation per se that requires energy (the efficiency there is limited only by the device design; computation can theoretically require zero energy), it is the erasure of data.

If we assume that most logical operations involve the input of two bits, which, absent some mechanism to save the input data, are then erased after they are used, most logical operations will expend a minimum of about 4×10^{-21} J (see Appendix, "The Landauer Limit"), regardless of device design. However, in the real world, devices are not perfectly efficient, and this number does not include the energy dissipated by electrical resistance, or friction.

This is quite important because, in conventional computers, losses due to electrical resistance dwarf the thermodynamic penalty for erasing data imposed by the Landauer Limit. It is for this reason that conventional computer designs give no heed to the energetic penalty of erasing data; the penalty is, given the overall inefficiency of conventional logic gates, negligible.

However, if a logic gate could be designed so that it was inherently very efficient, it is possible for the opposite situation to occur: Rather than the energetic penalty associated with the Landauer Limit comprising a negligible portion of the energy dissipated by a gate, the Landauer Limit could be the cause for almost all of the gate's energy dissipation, with the gate's inherent inefficiency approaching 0. This nearly-perfect gate, if erasing 2 bits of data per logical operation, would therefore dissipate about 4×10^{-21} J.

Due to the units used in most computer benchmarks, Joules / logical operation is not an easy metric to use for comparisons with conventional computers. Translating this figure into J/GFLOP (Appendix, "

Translating Between Logical Operations and GFLOPS"), a computer operating at the Landauer Limit requires 8×10^{-8} J/GLFOP. The most efficient current supercomputers use about 0.142 J/GFLOP [9], indicating that they are about 1.77 million times less efficient than a computer operating at the Landauer Limit. Given this, the design of efficient mechanical computers obviously has great value even if all they do is reduce energy dissipation down to the level of the Landauer Limit.

However, the Landauer Limit of 4×10^{-21} J per logical operation is quite large compared to the energy lost to friction via a NAND gate of the link- and rotary-joint style (3.9×10^{-26} J per logical operation). It makes little sense to have such an efficient mechanism, but to then operate it in such a manner that power losses due to bit erasure reduce its effective efficiency by several orders of magnitude.

Reversible Computing

Conventional computers erase bits as a matter of course, being so otherwise inefficient that the energetic penalty associated with bit erasure is meaningless to their operation. However, in a scenario where the underlying mechanisms of the computer are so efficient that this is not true, bit erasure becomes a concern. Luckily, bits do not have to be erased. Reversible computing is a well-known way to circumvent the Landauer Limit. Using reversible computing schemes, bits are stored, even if technically no longer needed to produce the final output, so that the penalty associated with bit erasure is not incurred.

Reversible computing comes in different forms and degrees. In what might be considered the ultimate form of reversible computing, all input, output, and intermediate bits are stored. Then, once the desired computation is complete, rather than just "wiping" the memory (which equates to the erasure of many bits), the whole computation is run backwards (or in reverse, hence "reversible computing"), resetting inputs, outputs, and memory to its original state without erasing any bits.

While this may sound quite unintuitive to those outside the field, reversible computing has been known for decades. There are even programming languages and compilers made specifically to ensure that programs are reversible (granted, they are not currently in widespread use since avoiding the Landauer Limit is, for reasons described above, of mainly academic interest when using conventional hardware).

Reversible Logic Gates

The rotary joint- and link-based system we propose is easily used to implement reversible computing. It is well-known that any gate can be designed to avoid erasing bits by simply carrying the inputs forward as additional outputs, ensuring that both the inputs and outputs are always known. This results in additional "garbage" bits, and requires additional memory, but avoids the energetic penalty inherent in bit erasure. In other words, it would be trivial to modify the described NAND gate to be reversible.

Further, there are well-known gates that are inherently reversible. Such reversible gates include the Toffoli gate and the Fredkin gate. And, not only are these gates reversible, but, like the NAND gate, they are universal. Meaning, given the appropriate configuration of, e.g., Fredkin gates, any desired logic can be implemented. These gates, like the more common irreversible gates already discussed, are readily implemented using only rotary joints and links.

The Fredkin Gate

Figure 17 depicts a Fredkin gate. As previously mentioned, in the same way that a NAND gate is a universal gate for combinatorial logic, the Fredkin gate is as well. However, the additional benefit of the Fredkin gate is that it is reversible. A Fredkin gate does not erase data as it

performs logical operations. As a result, a Fredkin gate is not subject to the Landauer Limit; it can operate as efficiently as the mechanism itself can be made.

While determining the detailed operation of the Fredkin gate will probably be obvious given the earlier example of the NAND gate, some comments may help clarify its workings. First, a Fredkin gate is a 3-input, 3-output gate. Using the two-input system, the inputs are A0, A1, B0, B1, C0, and C1. The outputs are X0, X1, Y0, Y1, Z0, and Z1. Since it may be typical to think of the operation of these mechanisms as proceeding from left to right, the fact that the X0 and X1 links are not on the right might be confusing. This is for diagrammatic clarity only. There is no reason that those outputs could not be on the far right side along with Y0, Y1, Z0, and Z1 – it just requires crossing more lines. Two links are labeled “4-bar linkages” just to make it clear that they are not balances. Rather, as they are pushed on by X0 or X1, they do not rotate with respect to the overall diagram, but instead stay vertical as they are actuated. Where links meet at right angles and there is a solid corner, this indicates that this corner is rigid. This is, again, for diagrammatic clarity. In an actual device, these would probably not be efficient structures for transmitting the desired movement.

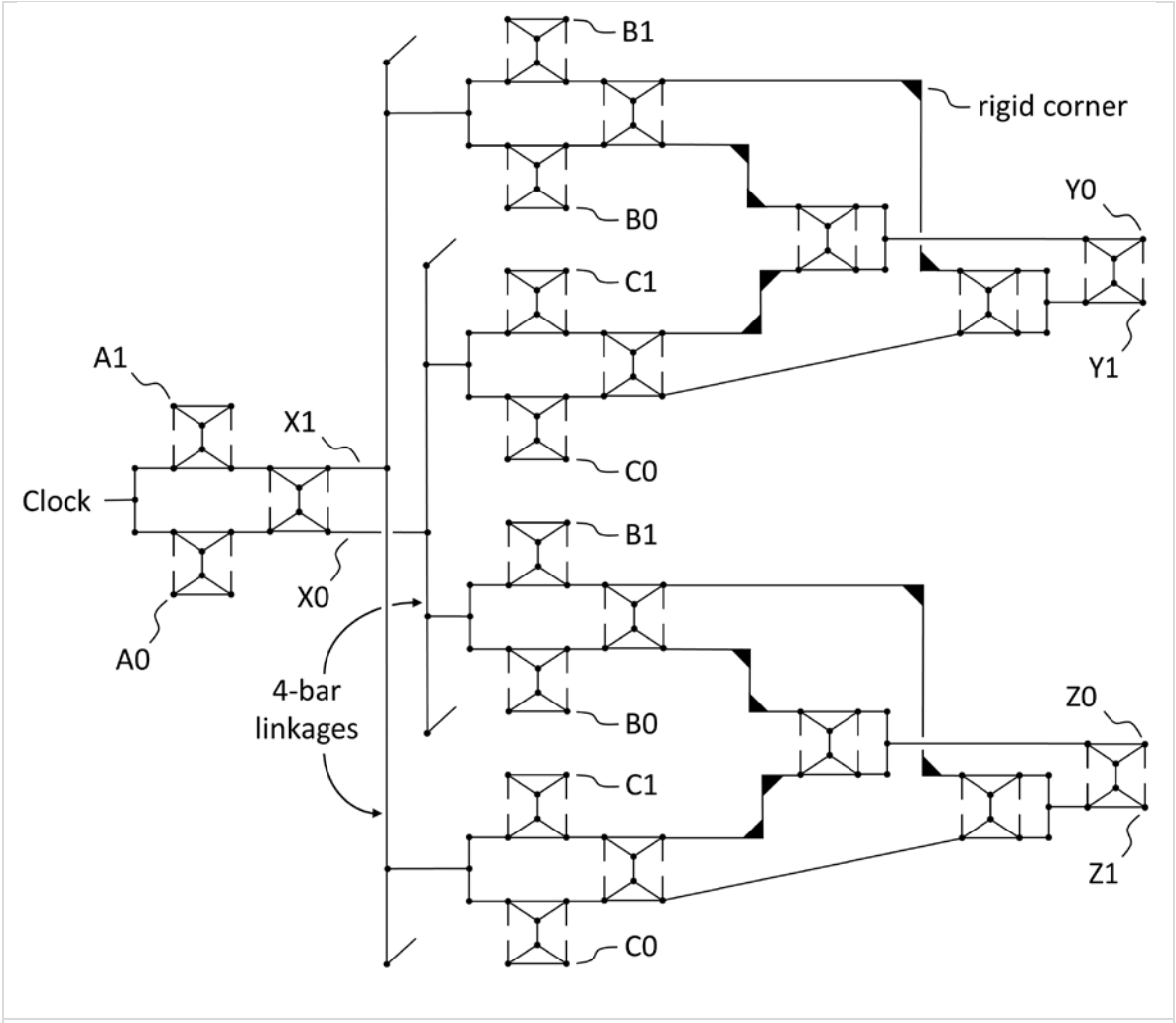

Figure 17: A Fredkin Gate

Note that, in a manner similar to how cells can be connected to form a shift register, Fredkin gates can be connected to form a system of arbitrarily complex, but completely reversible, logic.

Other Reversible Mechanisms

It is worth noting that the same principles that apply to reversible combinatorial logic also apply to sequential logic. Meaning, structures such as shift registers can be operated reversibly (click for an animated example). The same comments apply to data lines and any other miscellaneous structures that may be needed to build a complete system.

Reversible Clocking

Having noted that any logic gate can be made reversible, and having explicitly shown that a well-known, universal, reversible gate, the Fredkin gate, can be created directly from only links and rotary joints, a brief mention of reversible clocking is in order.

Systems that are physically and logically reversible may rely upon particular clock schemes to facilitate this reversibility. There are a variety of reversible clocking schemes, including Bennett Clocking, and Landauer Clocking, which can be used to ensure that a reversible mechanism can be run in reverse, "uncomputing" its previous results so that the system can be reset without having to erase a substantial amount of data. These clocking schemes are not complex, or controversial. They are simply different than the schemes used in irreversible systems, and so it is worth noting that some such scheme would be incorporated into a reversible design.

Summary: Reversible Computing

In the section "Molecular-Scale Implementation," we describe one method of implementing rotary joint- and link-based logic with frictional losses so low that bit erasure becomes, by far, the dominant factor in determining energy dissipation.

In the present section we have described how reversible computing can be used to avoid bit erasure. Reversible computing thereby allows the described logic to attain its full potential, operating with energy losses several orders of magnitude under the Landauer Limit.

Conclusions and Future Work

We have described a method for mechanical computing based on the realization that there are ways to lower friction to the point where it dissipates far less energy than the analogous electrical resistance in a conventional electronic computer. And, as the computations have shown, by “far less” we mean by a factor of billions.

It should be noted that these are early designs; almost arbitrary implementations of the new design paradigm just to show proof of concept. The designs presented have not been optimized, nor have they been vetted at all possible scales. Various avenues of research are still being pursued.

For example, it is assumed that the links in a lock pivot 1 radian when moving between the 0 and 1 positions. Can this movement be substantially reduced while maintaining reliable device functioning? If so, speed and energy benefits result directly from such an improvement.

Can the frequency of the mechanisms be substantially increased? Probably. Compared to the 18 GHz resonance frequency calculated, the assumed 0.1 GHz operating frequency is very low and was chosen to err on the side of caution. The maximum working frequency for the designs shown, or for future, more optimized designs, may be much higher.

Scale-related issues also warrant more investigation. For example, while the functioning of single rotary joints, and small rotary joint- and link-based mechanisms, have been well-characterized, it is possible that, when coupled together into larger devices, unforeseen issues arise (e.g., long-distance resonances that operate at lower frequencies).

Low-temperature behavior remains to be investigated, and may offer substantial additional benefits.

And, failure modes need to be investigated. For example, is it possible (especially when, e.g., trying to minimize the angle which a link traverses between 0 and 1) to have thermal fluctuations cause an erroneous result, registering the link as being a 0 when it should have been a 1, or vice versa? If so, the links may have to be stiffer, move farther, move more slowly, or error correction techniques may have to be introduced. Similar questions might apply to the possibility of a link simply breaking.

That all being said, we are not discussing incremental improvements to existing technology where differences of 10%, or 20%, or even 90%, actually make much difference. Molecular manufacturing should enable a new paradigm in energy-efficient computing millions or billions of times more efficient than current computers.

This is a new area of research, and as such, there are probably many design improvements to be discovered. We therefore liken these early designs to the “Model T” of mechanical computers, and anticipate that future designs will become faster and more efficient still. However, even were this not the case, and we assumed that not only had we bizarrely stumbled upon the perfect design already, but that yet-to-be-discovered problems are going to somehow lower performance 10X, or even 100X – the improvement over the current state of the art in electronic computing still cannot be overstated.

Appendix

Additional Traditional Logic Gate Designs

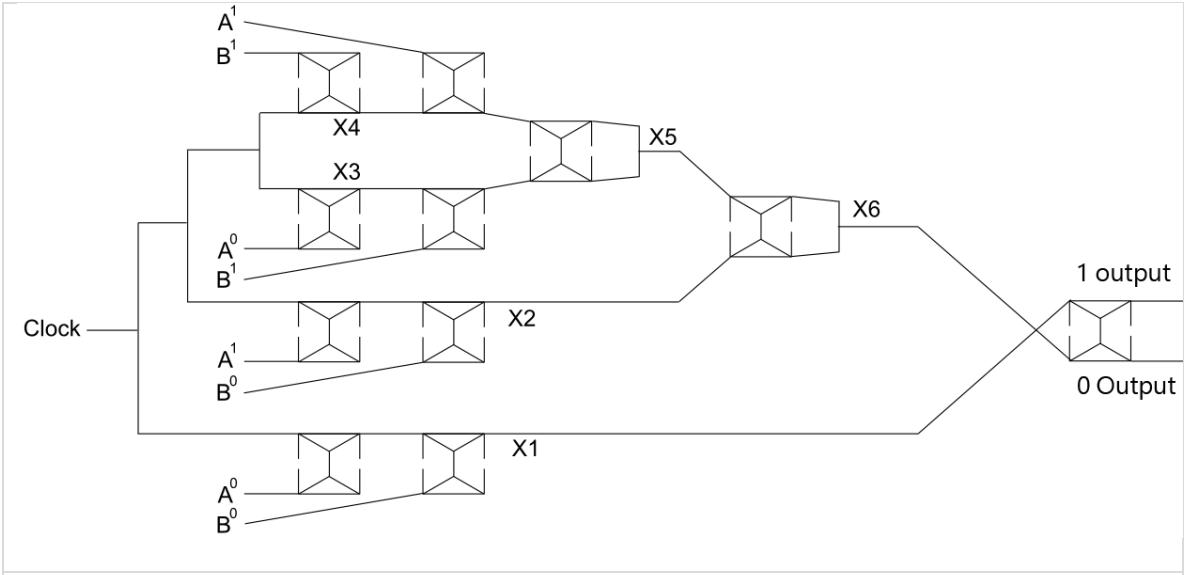

Figure 18: AND Gate

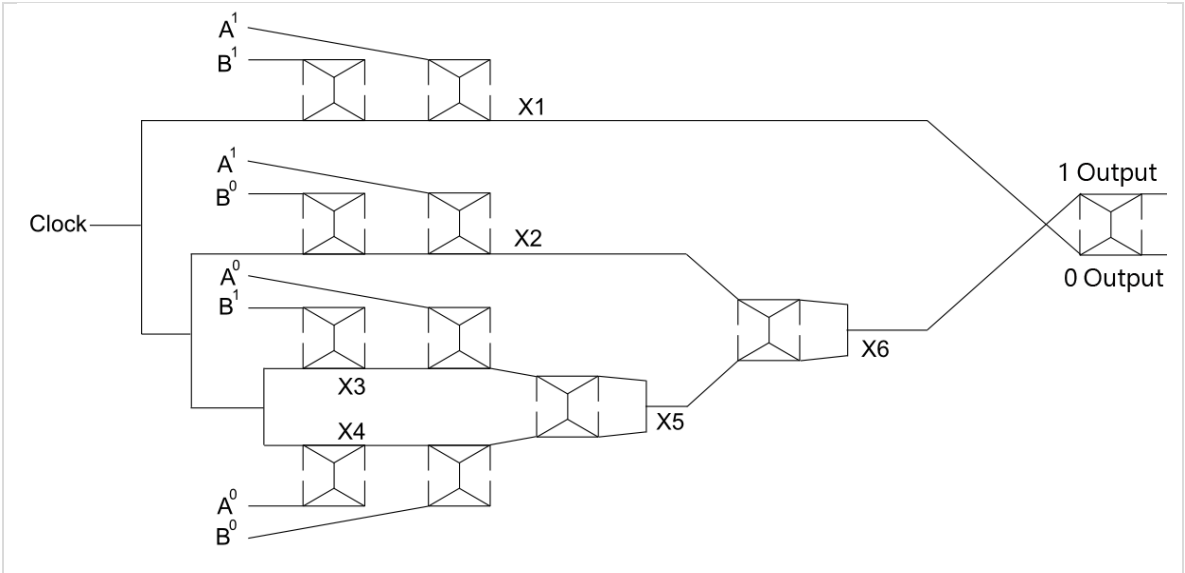

Figure 19: OR Gate

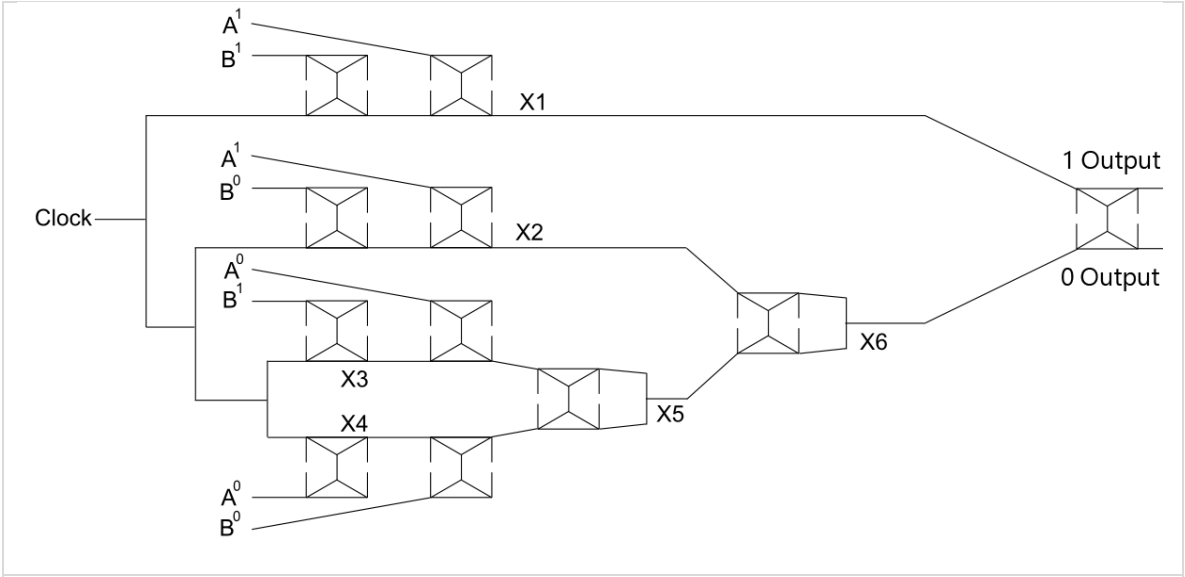

Figure 20: NOR Gate

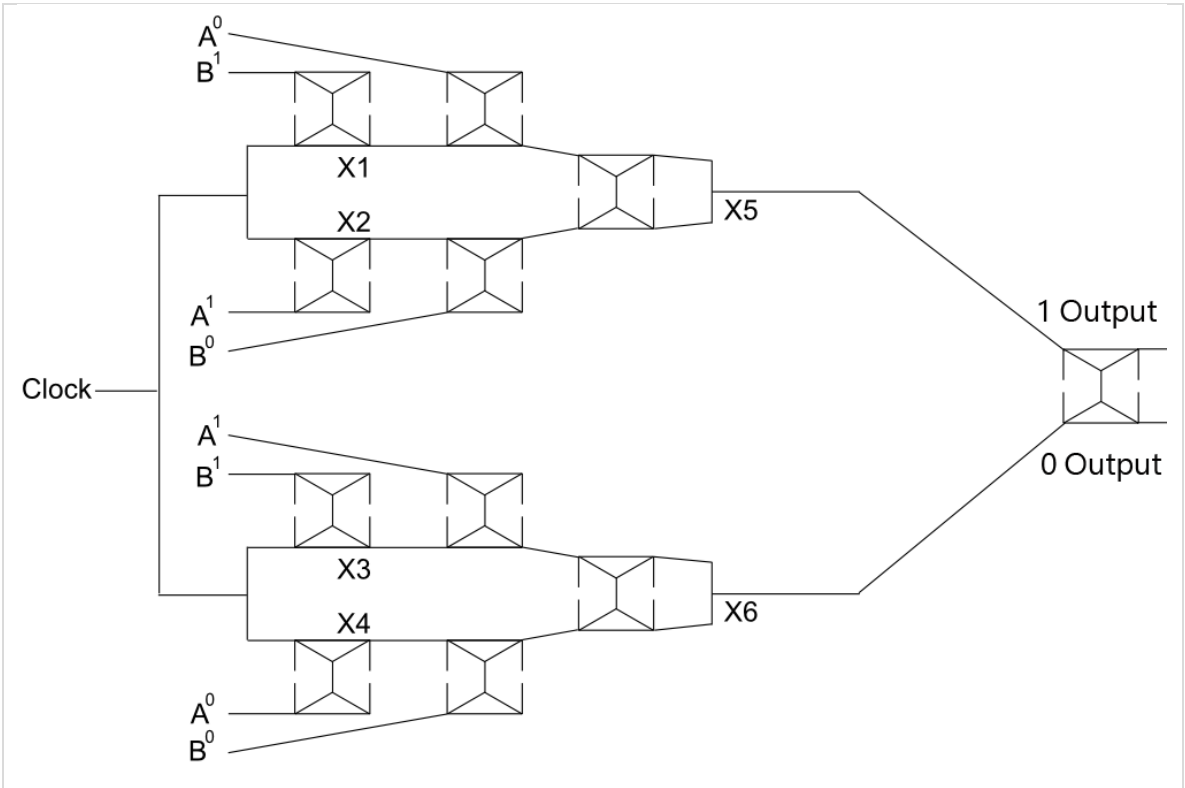

Figure 21: XOR Gate

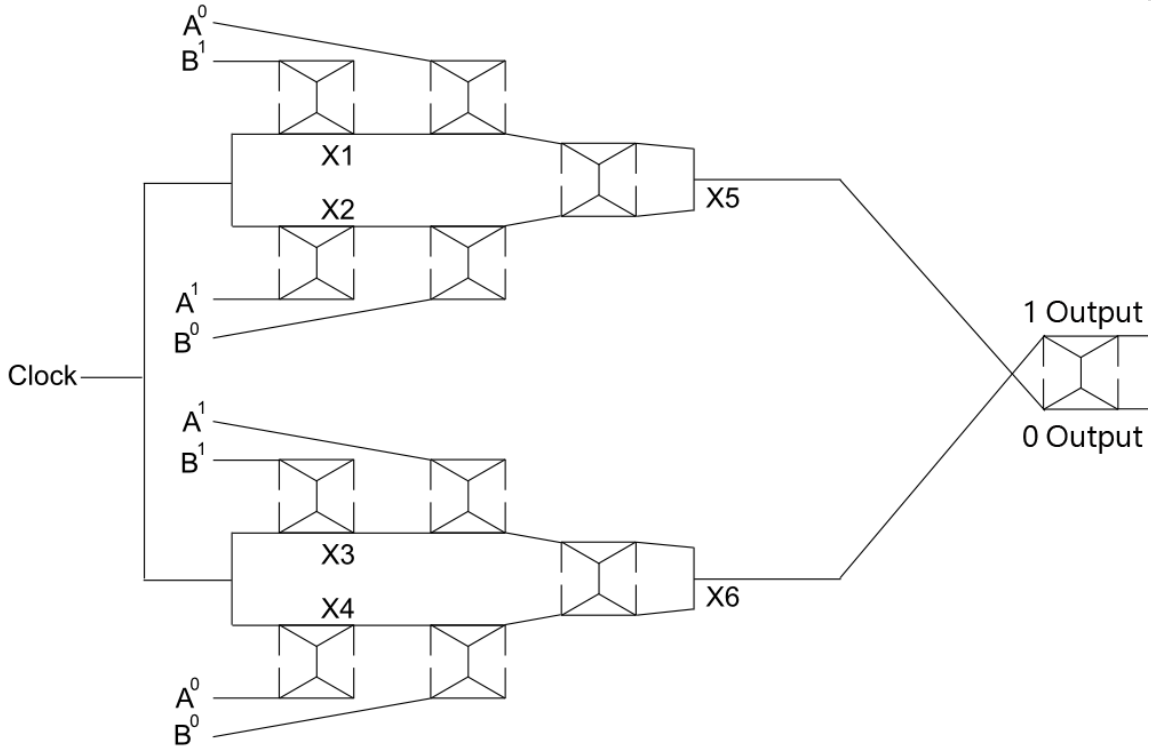

Figure 22: XNOR Gate

Computational Chemistry Simulations

Computational chemistry simulations of the acetylenic rotary joint were used to determine its drag coefficient. [13]

The Landauer Limit

The cost to erase data is kT ln(2) per erased bit, where k is Boltzmann's constant, and T is temperature in Kelvin. This principle is often referred to as the Landauer Limit, and has been measured experimentally [12]. At 205K the Landauer Limit is 2×10^{-21} J / bit erased. At room temperature, the Landauer Limit is about 2.8×10^{-21} J /bit erased.

To be clear, the erasure of a bit is not the same as a logical operation, nor is the erasure of a bit the same as a floating point operation. A bit is erased when a state representing 0 or 1 is irreversibly lost.

For example, assume the existence of a perfectly-efficient mechanical logic gate. This logic gate has no friction whatsoever. A 2-bit input is provided to this gate, the result computed, and then the gate is reset to its original state. No data about the inputs or the output has been saved, meaning that the inputs have been erased. Even using a hypothetical, perfectly-efficient gate, this erasure of 2 bits of information will incur the energetic cost of 2 bits x kT ln(2)/bit, or about 4×10^{-21} J at 205K.

Energy Dissipation Calculations

Computational chemistry simulations indicate that the drag coefficient, K_drag, of a single rotary joint like that shown in Figure 14 is 5.5×10^{-36} J s.

However, most mechanisms use multiple rotary joints to accomplish a single logical operation. The basic lock mechanism uses about 10 rotary joints (only half of which move for a given input). Therefore, the overall K_drag for a lock is about $5 \times 5.5 \times 10^{-36}$ J s, or 2.75×10^{-35} J s.

The NAND gate, being composed of 11 locks plus several balances, consists of 125 rotary joints, about 70 of which move for a given calculation. Therefore, the overall K_drag for a NAND gate is about $70 \times 5.5 \times 10^{-36}$ J s, or 3.9×10^{-34} J s.

How much power does this translate into? The relevant formula is:

$P\_drag = K\_drag \times (V\_drag)^2$	**Equation 1: Rotational Energy Dissipation**

Where V_drag = rotational speed, in radians/second, between the housing and the rotor.

To use this formula, some estimates must be made about how far a link must rotate to unambiguously be able to differentiate between the position representing “0” and the position representing “1”. The frequency must also be chosen.

For the angle of rotation, we choose a conservative round number of 1 radian (about 57 degrees; this number could probably be reduced but this is not a parameter we have attempted to optimize).

To better address limitations on how fast the links can rotate, calculations have been done to determine the amount of force which needs to be applied to the system to achieve a given acceleration, and the resonant frequencies of the system. Force is not the limiting factor; the bonds are easily strong enough to tolerate forces which would allow sub-nanosecond switching time.

However, exciting mechanical resonances is a concern. The major system resonance of an exemplary system using rotary joints as depicted in Figure 14, and links composed of diamond rods which are, on average, 20nm long, was determined to be at 18 GHz. To be extremely conservative, we use a switching time of 10ns (0.1 GHz), which stays very, very far away from the 18 GHz resonance frequency.

Plugging these values into Equation 1, for the lock we have:

$$P\_drag = (2.75 \times 10^{-35} \text{ J s}) \times (1 \text{ radian}/10^{-8} \text{ s})^2 = 2.75 \times 10^{-19} \text{ watts}$$

And for the NAND gate we have:

$$P\_drag = (3.9 \times 10^{-34} \text{ J s}) \times (1 \text{ radian}/10^{-8} \text{ s})^2 = 3.9 \times 10^{-18} \text{ watts}$$

Since Watts = J/s, to find out the energy consumed by a single logical operation, we multiply Watts by the amount of time a single logical operation takes, here assumed to be 10^{-8} s. The final figure will vary somewhat with the particular structure carrying out the logical operation, since each structure may have a different number of moving rotary joints.

For the NAND gate, the result is 3.9 x 10^{-26} J per logical NAND operation. For a single movement of a lock, this same calculation gives a figure of 2.7 x 10^{-27} J.[1]

Obviously, these calculations consider only frictional losses. Any system would need to operate reversibly to operate under the Landauer Limit. This can be done with NAND gates using the proper clocking scheme, or reversible gates can be used (for which the calculations would be slightly different due to varying numbers of rotary joints, but not different to a degree that is meaningful to our conclusions).

Translating Between Logical Operations and GFLOPS

Joules per logical operation are not easy to interpret in terms of conventional computer benchmarks because such benchmarks are normally provided in Watts/GIGAFLOP, or its reciprocal. A FLOP assumes a 64-bit floating point operation.

Using 3.9 x 10^{-26} J per logical NAND operation as the example, to perform the necessary conversion, it is assumed that about 20,000 logical operations (LogOps) are required per FLOP. [14] Therefore, the computation becomes:

(J/LogOps * 20,000 LogOps/FLOP) * (10^9 FLOP/GFLOP)/ = J / GFLOP;

Or, to provide simple conversion factors:

(J/LogOps) x (2x10^{13}) = J/GFLOP; and
(J/GLFOP) / (2x10^{13}) = J/LogOps

Using these conversions, we conclude that using the rotary joint- and link-based designs described herein, NAND-based LogOps consume 7.8 x 10^{-13} J/GFLOP. If we assume a number of NAND gates operating in parallel so that, at a switching speed of 0.1ns per NAND gate, 1 GFLOP of computation is provided in 1 second, then J becomes equal to J/sec, or Watts, so 7.8 x 10^{-13}

[1] These figures assume no loss from bit erasure. If bit erasure was frequently required, the Landauer Limit would dominate the energy loss, no matter how efficient the mechanism was.

J/GFLOP becomes equivalent to 7.8×10^{-13} Watts/GFLOP. Then we can simply take the reciprocal of the previous number, showing that such a system provides 1.28×10^{12} GFLOP/Watt.[1]

As compared to a conventional supercomputer providing 7 GFLOPS/Watt (or 0.142 J/GFLOP at 1 GFLOP/sec), this is $(1.28 \times 10^{12} / 7) = 1.8 \times 10^{11}$ times more efficient.[1]

References

1. Reif, J.H., *Mechanical Computing: The Computational Complexity of Physical Devices*, in *Encyclopedia of Complexity and System Science*, R. Meyers, Editor. 2009, Springer-Verlag. p. 5466-5482.
2. Feynman, R., *Quantum Mechanical Computers.* Optics News, 1985: p. 11-20.
3. Credi, A., S. Silvi, and M. Venturi, *Molecular Machines and Motors*. Topics in Current Chemistry. Vol. 354. 2014: Springer.
4. Joachim, C. and G. Rapenne, *Single Molecular Machines and Motors: Proceedings of the 1st International Symposium on Single Molecular Machines and Motors*. 2013: Springer.
5. Isobe, H., et al., *Molecular bearings of finite carbon nanotubes and fullerenes in ensemble rolling motion.* Chemical Science, 2013. **4**(3): p. 1293.
6. Wang, Y.-L., et al., *Molecular Rotors Observed by Scanning Tunneling Microscopy*, in *Three-Dimensional Nanoarchitectures*, W. Zhou and Z. Wang, Editors. 2011. p. 287-316.
7. Manzano, C., et al., *Step-by-step rotation of a molecule-gear mounted on an atomic-scale axis.* Nat Mater, 2009. **8**(7): p. 576-579.
8. Kottas, G., et al., *Artificial Molecular Rotors.* Chem. Rev., 2005. **105**: p. 1281-1376.
9. *Performance per watt.* Available from: https://en.wikipedia.org/wiki/Performance_per_watt.
10. Bennett, C. and R. Landauer, *The Fundamental Physical Limits of Computation.* Scientific American, 1985.
11. Landauer, R., *Irreversibility and Heat Generation in the Computing Process.* IBM Journal of Research and Development, 1961. **5**: p. 183-191.
12. Berut, A., et al., *Experimental verification of Landauer's principle linking information and thermodynamics.* Nature, 2012. **483**(7388): p. 187-189.
13. Merkle, R., et al., *"Determination of the Drag Coefficient of an Acetylenic Rotary Joint" [manuscript in preparation]* 2016.
14. DeBenedicitis, E., *Reversible Logic for Supercomputing.* ACM, 2005.
15. Drexler, K.E., *Nanosystems: Molecular Machinery, Manufacturing, and Computation*. Wiley, 1992.