EE i5393 UMN
Circuits, Computation, and Biology Spring 2015

Homework # 3

Due April 24, 2015

1. Logical and Iterative Computation with Molecular Reactions

   In the last homework, you implemented some simple functions with molecular reactions. In this homework, you'll implement some more complicated ones.

   (a) **XOR function**

   Design a set of DNA reactions that computes the exclusive-OR function of two inputs X and Y, where a large concentration of the type represents logical 1 and a small (or near zero) concentration of the type represents logical 0. Provide simulation results to validate your solution.

   (b) **Euclid's Algorithm**

   Euclid's algorithm is an efficient method for computing the greatest common divisor (GCD) of two integers, also known as the greatest common factor (GCF) or highest common factor (HCF). It is named after the Greek mathematician Euclid, who described it in Books VII and X of his Elements.

   In its simplest form, Euclid's algorithm starts with a pair of positive integers and forms a new pair that consists of the smaller number and the difference between the larger and smaller numbers. The process repeats until the numbers are equal. That number then is the greatest common divisor of the original pair.

   Design a set of molecular reactions that implements the procedure. Demonstrate that your code works for $x = 66$ and $y = 30$.

   (c) **Collatz Procedure**

   The *Collatz* conjecture is a famous open problem in mathematics, proposed by Lothar Collatz in 1937. Consider the following iterative procedure. For any positive integer x,

   - if $x = 1$ stop;

- else if x is odd, let $x = 3x + 1$;
- else let $x = x/2$.

The conjecture is that, starting with any positive integer x, the procedure always terminates with $x = 1$. For instance, starting with $x = 5$, one follows the sequence 16, 8, 4, 2 and 1. Starting from $x = 27$, one follows the sequence 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, ... (keep going, you'll see that you eventually hit one.)

Proving this is evidently difficult. Paul Erdös said about the conjecture: "Mathematics is not yet ready for such problems". He offered a monetary reward of $500 for its solution.

You are *not* asked to prove the *Collatz* conjecture on this homework. Rather you are asked to design a set of molecular reactions that implements the procedure. The input to the system is a quantity of a type X; the system should iterate through the Collatz sequence until it hits one.

Demonstrate that your code works for $x = 27$.

2. **XOR Function with Logic Gates**

The exclusive-or operation is at the core of all encoding and decoding functions. In class, we discussed at length circuits for computing this operation. We discussed a scheme for implementing $XOR(x_1, \ldots, x_n)$ with

$$\lceil [2.5(n-1)\rceil$$

AND/OR gates, with arbitrary fan-in, and as many NOT gates as are needed.

We also discussed a lower bound for $XOR(x_1, \ldots, x_n)$ with

$$2n - 1$$

AND/OR gates, with arbitrary fan-in, and as many NOT gates as are needed.

As a function of n:

n	$\lceil [2.5(n-1)\rceil$	$2n-1$
2	3	3
3	5	5
4	8	7
5	10	9
6	13	11
$\vdots$	$\vdots$	$\vdots$

We see that there is a gap, starting at $n = 4$: the best known circuit for computing $XOR(x_1, x_2, x_3, x_4)$ requires 8 AND/OR gates, while the strongest known proof is for a lower bound of 7 AND/OR gates.

(a) **Implementing XOR with AND gates**

Recall that the Exclusive-OR (XOR) function is 1 if an odd number of the inputs are 1, and 0 otherwise. Suppose that we have to build an circuit that computes XOR with AND gates, OR gates, and inverters.

i. Draw a circuit to compute the XOR of 13 variables with the fewest possible number of AND and OR gates gates, and as many inverters as you want. (Solutions has 30 AND and OR gates.)

ii. Draw a circuit to compute the XOR of 15 variables with the fewest possible number of AND and OR gates gates, and as many inverters as you want. (Solutions has 35 AND and OR gates.)

iii. Draw a circuit to compute the XOR of 23 variables with the fewest possible number of AND and OR gates gates, and as many inverters as you want. (Solutions has 55 AND and OR gates.)

iv. Draw a circuit to compute the XOR of 27 variables with the fewest possible number of AND and OR gates gates, and as many inverters as you want. (Solutions has 65 AND and OR gates.)

(b) **A Lower Bound (12.5 points)**

Solve any one of the following for full points.

i. Write a computer program that verifies that no circuit with 7 AND/OR gates computes the XOR of 4 variables. (You can limit the number of configurations that it tries up to symmetries, e.g., permuting the variables. But you have to convince me that you have verified all distinct configurations, up to such symmetries.)

ii. Write a program that verifies that no circuit with x AND/OR gates computes the XOR up to y variable. (**Bonus points if you can go above $x = 7$, $y = 4$.)**

iii. Improve the lower bound to $\lceil 2.5n - 1 \rceil$. (I don't know if this is true. In addition to full points for this problem, **I'll give you an immediate A in the course if you solve this**.)

iv. Discuss the implementation of the XOR of n variables in terms of CMOS transistors. What is the minimum number of transistors that you require? Can you prove a lower bound (or prove that your construct is optimal)?

3. **Circuits and Conditional Permutations**

In class, we discussed *conditional permutations.* Consider

```
{x: (1 3 4 2 5), (1 4 5 2 3)}.
```

The first block of 5 numbers specifies the permutation that is taken if $x = 1$, while the second block of 5 numbers specifies the permutation that is taken if $x = 0$.

Let's use `*` to denote the *identity* permutation. Consider the following sequence of conditional permutations.

```
{x: (1 3 4 2 5), *}
{y: (1 4 5 2 3), *}
{x: (1 5 2 4 3), *}
{y: (1 3 2 5 4), *}
```

Note that the two permutations for $x = 1$ are inverses. The same is true for the two permutations for $y = 1$. As you can verify, this sequence of permutations implements the permutation `(1 4 3 5 2)` if $x = 1$ and $y = 1$. It implements the identity permutation if either $x = 0$ or $y = 0$.

We interpret a sequence of permutations as follows: if the overall result is a permutation – any permutation – then the operation corresponds to logical 1; if the overall result is no permutation, then the operation corresponds to logical 0. Accordingly, this sequence implements xy i.e., the AND of x and y.

Now consider the following sequence.

```
{x: (1 3 4 2 5), *}
{y: (1 4 5 2 3), *}
{x: (1 5 2 4 3), *}
{y: (1 3 2 5 4), *}
{*: (1 2 5 3 4), (1 2 5 3 4)}
```

It is the same sequence as above, except that here we have appended the *unconditional permutation* `(1 2 5 3 4)` at the end. (Here "unconditional" means we perform the same permutation regardless, so it is not conditional on the value of a variable.) Note that the unconditional permutation is the inverse of the permutation computed by the sequence above. Accordingly, this sequence implements identity if $x = 1$ and $y = 1$, and it implements `(1 2 5 3 4)` otherwise. So this sequence implements $(xy)'$, i.e., the NAND of x and y.

In fact, we can sneak the unconditional permutation in with the last one, by simply multiplying it in:

```
{x: (1 3 4 2 5), *}
{y: (1 4 5 2 3), *}
{x: (1 5 2 4 3), *}
{y: (1 4 2 3 5), (1 2 5 3 4)}
```

For what follows, define the following five permutations:

```
A = (1 4 3 5 2)
B = (1 4 5 2 3)
C = (1 3 4 2 5)
D = (1 2 4 5 3)
E = (1 4 2 3 5)
```

Now note that:

```
A = CBC'B' ; A' = BCB'C'
B = CDC'D' ; B' = DCD'C'
C = DED'E' ; C' = EDE'D'
D = EBE'B' ; D' = BEB'E'
E = DAD'A' ; E' = ADA'D'
```

Here `X'` means the inverse of `X`. You will constantly be referring back to these identities.

Example

Write a sequence of conditional permutations for

```
F(x, y, z) = (x z)' (x'y')'
```

Begin by writing

```
{(x z)' (x'y')': A, *}
```

We expand this as follows:

```
{(x z)': C, *}
{(x'y')': B, *}
{(x z)': C', *}
{(x'y')': B', *}
```

Now we expand the first line as:

```
{x: E, *}
{z: D, *}
{x: E’, *}
{z: D’C, C}
```

Putting these together:

```
{x: E, *}
{z: D, *}
{x: E’, *}
{z: D’C, C}
{(x’y’)’: B, *}
{(x z)’: C’, *}
{(x’y’)’: B’, *}
```

After we do the next expansion, we will have the following:

```
{x: E, *}
{z: D, *}
{x: E’, *}
{z: D’C, C}
{x’: D, *}
{y’: C, *}
{x’: D’, *}
{y’: C’B, B}
{(x z)’: C’, *}
{(x’y’)’: B’, *}
```

Next, inverting `x` and `y`, we get:

```
{x: E, *}
{z: D, *}
{x: E’, *}
{z: D’C, C}
{x: *, D}
{y: *, C}
{x: *, D’}
```

```
{y: B, C'B}
{(x z)': C', *}
{(x'y')': B', *}
```

Completing all the expansions, we get:

```
{x: E, *}
{z: D, *}
{x: E', *}
{z: D'C, C}
{x: *, D}
{y: *, C}
{x: *, D'}
{y: B, C'B}
{x: D, *}
{z: E, *}
{x: D', *}
{z: E'C', C'}
{x: *, C}
{y: *, D}
{x: *, C'}
{y: B', D'B'}
```

Problem

You have your choice of four subproblems that you can answer. Each earn will earn you full credit for this question.

(a) Write a program that produces a sequence of conditional permutations corresponding to any given Boolean function. Specify the function however you like: as a truth table, as an algebraic expression, etc. (As always, use whatever programming environment you like.)

(b) Write a program that produces the Boolean function corresponding to any sequence of conditional permutations. (As always, use whatever programming environment you like.)

(c) Answer the following questions:

i. Write sequences of conditional permutations that implement the following functions.

A. $f(x, y) = x + y$

B. $f(x, y, z) = (x'y'z')'$

C. $f(x, y, z) = xy + xz + yz$

ii. Find a different sequence of permutations than A, B, C, D, E above that can implement arbitrary Boolean functions. Explain why your choice works.

iii. Prove that no set of cyclic permutations of length less than five will work.

(d) Make progress on any one of the following research topics (and earn significant bonus credit beyond the value of this question):

i. Find a way of effectively minimizing the length of sequences of conditional permutations.

ii. Find a way of effectively testing whether a sequence of conditional permutations implements the Boolean function "identically zero" (or, equivalent "identically one").

iii. Find a way of effectively implementing Boolean functions through conditional permutations at the *circuit level*. (For instance, propose a programmable architecture based on this idea.)

Here "effectively" is entirely subjective. If you find a method that takes polynomial time in the number of inputs for either the first two tasks then: either your method is wrong; or I'm wrong about this whole topic in some fundamental way; or you get the Turing Award.

4. **Counting Networks**

Many fundamental multi-processor coordination problems can be expressed as *counting problems*: processors collectively assign successive values from a given range. This is sometimes referred to as the *Bakery Problem.* Since few North Americans spend time waiting in line in bakeries, we'll use the metaphor of a visit to the Department of Motor Vehicles (DMV): when someone walks into a DMV office, they are assigned a number and then served when it is called. With enough people, merely handing out the numbers can become a bottleneck.

An elegant solution to this problem is a *counting network.* Such a network implements a mod n shared counter with a network of *balancer* objects. Each balancer is implemented with a Boolean variable in shared memory representing its state (0 for "up" and 1 for "down"), as well as two pointers to successor balancers. These pointers are static; the topology of the network is fixed. Processes shepherd *tokens* through the network, toggling balancers through atomic *read-modify-write* (rmw) operations, and moving on to succeeding balancers: to the "top" balancer if the state was "up" and to the "bottom" balancer if the state was "down". This is described by the following pseudo-code:

```
balancer = [toggle: boolean, next: array [0..1] of ptr to balancer]
traverse(b: balancer)
   loop until leaf(b)
      i := rmw(b.toggle := !b.toggle)
      b := b.next[i]
   end loop
end traverse
```

There are designated *input* balancers and *output* balancers. The output balancers have no successors; rather they point to indices (from 0 to $n-1$ for a mod n counter). It convenient to represent a balancer as two dots joined by a vertical bar, as shown in Figure 5, and the full data structure of the counting network by wiring up the balancers, as shown in Figure 2. The numbering of the tokens indicates the order in which they arrived. (This is to illustrate what's going on; in fact, the tokens are not numbered.)

Figure 2 is an example of a counting network called the `Batcher` network, for $n = 4$. Here tokens are placed on randomly chosen input balancers and routed to output balancers. The DMV number that gets assigned is that of the index number of the output wire. Note that the first token arrives at index 0, the second at index 1, and so on. In general, the i-th token arrives at index i mod 4.

This is the defining property of a counting network: it is *counting* the number of input tokens without ever passing them all through a shared computing element!

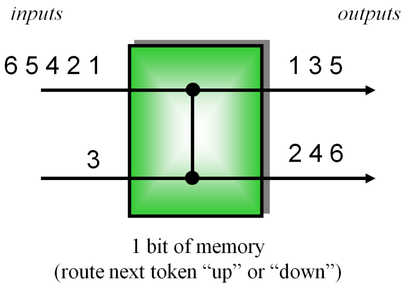

Figure 1: A balancer.

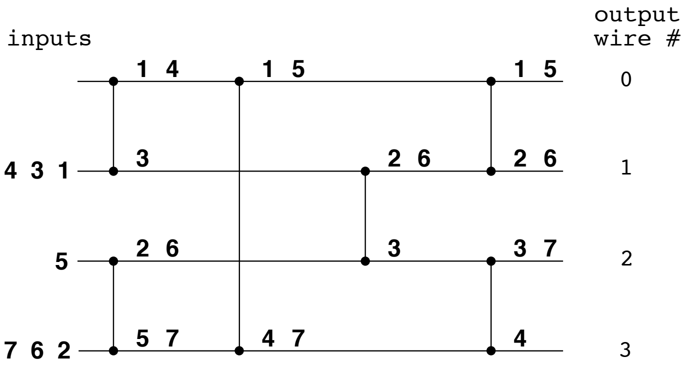

Figure 2: Routing of tokens in a counting network.

The `Batcher` network is constructed with a block called the `Merge` network. The recursive construction of the `Merge[8]` network is shown in Figure 3. The recursive construction of the `Batcher[8]` network, based on the `Merge[8]` network, is shown in Figure 4. `Merge` and `Batcher` networks for larger values of n that are powers of 2 are constructed similarly. At the bottom of the recursion, `Merge[2]` and `Batcher[2]` networks both consist of a single balancer.

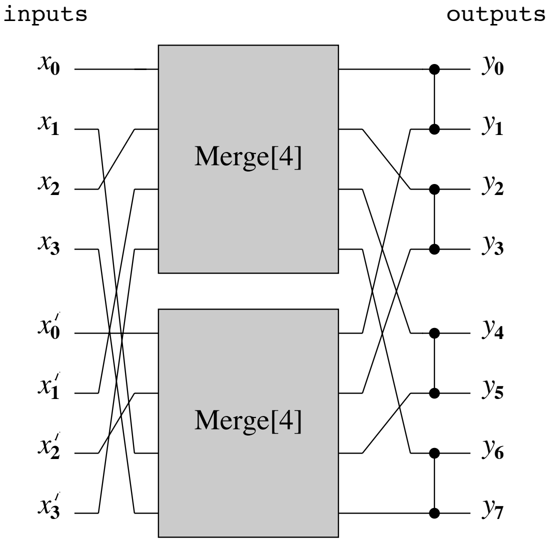

Figure 3: the `Merge[8]` network.

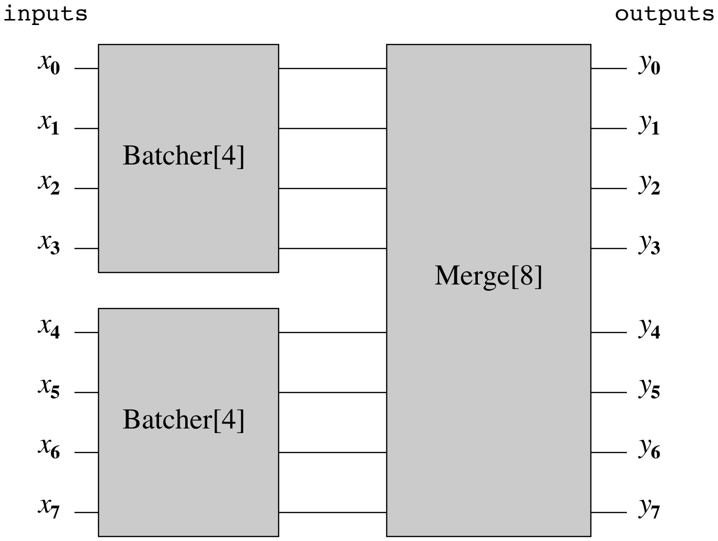

Figure 4: the `Batcher[8]` network.

Questions

The length of a path through the balancing network is the number of balancers that a shepherding process will visit from input to output when routing a token. You are asked to analyze the lengths of such paths. The length of the paths, of course, relates to how long it takes processors to route their tokens, so correlates with the time complexity of the algorithm. (Since the data structures are defined recursively for powers of 2, you might find it helpful to reason in terms of *recurrence relations.*)

(a) What is the length of the paths through a `Merge[`n`]` network where n is some power of 2?
(5 points)

(b) What is the length of the paths through a `Batcher[`n`]` network where n is some power of 2? (10 points)

5. **Counting vs. Smoothing**

An alternate view of a balance is as device that balances two integers x and y, as shown in Figure 5.

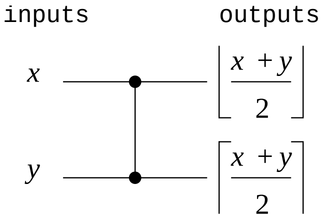
inputs
outputs
x
y
$\left\lfloor \frac{x+y}{2} \right\rfloor$
$\left\lceil \frac{x+y}{2} \right\rceil$

Figure 5: a balancer.

The Batcher counting network presented in class has depth $O(\log^2 n)$. In this problem, we consider a network of balancers with depth $\log n$.[†]

Consider the network shown in Figure 6.

(a) If the inputs x_0, x_1, x_2, and x_3 are 7, 3, 2 and 6, respectively, what are the outputs y_0, y_1, y_2, and y_3?

(b) For an arbitrary input sequence, prove that y_0 is the smallest output value and y_3 is the largest. Prove that the maximum difference between y_0 and y_3 is two.

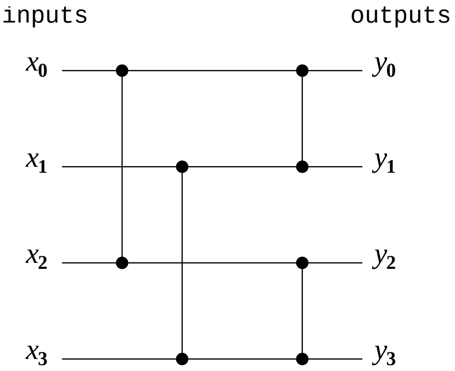

Figure 6: the `Smooth[4]` network.

Consider the `Smooth[`n`]` network, $n \geq 2$, shown in Figure 7. It is constructed recursively with two `Smooth[`$n/2$`]` networks and $n/2$ balancers. The `Smooth[2]` network consists of a single balancer.

[†] All logarithms are base 2.

(c) Draw the `Smooth[8]` network. Prove that the maximum difference between y_0 and y_7 is three.

(d) For the `Smooth[`n`]` network, $n \geq 2$, prove that the maximum difference between y_0 and y_{n-1} is $\log n$.

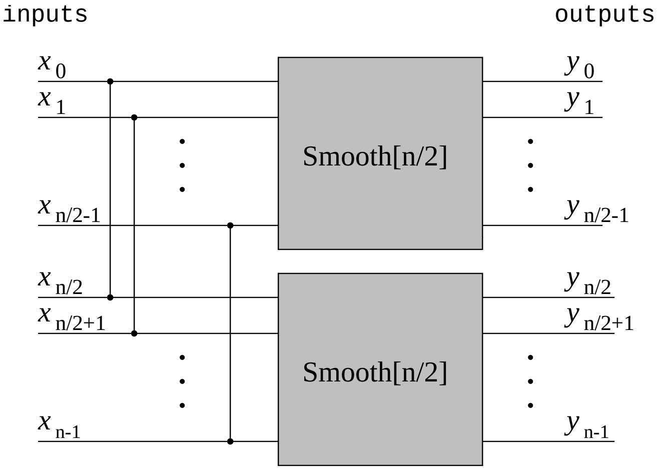

Figure 7: the `Smooth[`n`]` network.