Unscrambling Address Lines

Andrei Broder* Michael Mitzenmacher* Laurent Moll*

Abstract

A writer leaves a message in a write-once memory accessible via address lines. Before the intended recipient has a chance to get the message, the address lines are permuted by an adversary. We provide a simple, nearly optimal algorithm for the reader and writer to communicate over such a channel.

This problem arose in the context of FPGA hardware design. Our algorithm has been implemented and is part of the design tool suite in use within Compaq.

1 Introduction

Consider the following problem regarding the transmission of a message between a writer and a reader facing an adversary. The writer stores logical zeroes and ones in a table of size 2^n stored in consecutive locations in a write-once memory. The memory is accessed through n one bit address lines. After the writing is complete, an adversary permutes the address lines. For example, for $n = 4$ there are sixteen memory locations: if the address lines are set to 0010, before the adversary acts, the memory returns the value stored in location 2. If the adversary permutes the second and third address line, the memory sees a request for location 0100 and returns the value stored in location 4.

The reader does not know the permutation used by the adversary, but can read all the memory locations. The reader's goal is to discover how the address lines were permuted, and, in addition, to obtain a message from the writer. Assuming the reader and writer establish a protocol ahead of time, how many bits can they communicate? More practically, what is a good protocol?

This problem arose in the context of Field-Programmable Gate Arrays (FPGAs) hardware design. An FPGA is a simple reconfigurable hardware device. The first commercial FPGA was introduced in 1986 [1]. For a large part of today's FPGAs, their basic logical element is equivalent to a look-up table [4]. The usual tools for FPGA design lay out a circuit on these logical elements, routing the wiring as appropriate. In particular, one tool currently in use permutes the address lines as appropriate to improve the wiring layout. This process is perfectly reasonable if the FPGA programmer want to use the design as a "black box." However, if the FPGA programmer wants to patch the design, an effective means of determining this permutation is necessary. The number of memory locations in the table dedicated to this end should be as low as possible, so that the rest of the table can be used for other purposes. (Because of the layered structure of the complex software used for wiring layout, keeping track of the permutation through the layers is not feasible.)

We describe a brute-force approach to the problem, as well as a simple algorithmic solution.

2 Brute force: table look-up

For any specific n, the problem can be solved by brute force. We divide all possible settings of table-content bits into equivalence classes; two settings are equivalent if and only if the first yields the same memory output as the second via some address lines permutation. We then count the number of equivalence classes with $n!$ distinct members. If C_n is the number of such classes, then the writer can effectively transmit any value in the range $[0 \dots C_n - 1]$ in such a way that the reader can determine the value plus the permutation used by the adversary. This is accomplished by establishing one representative member from each of the C_n equivalence classes, and sending one of these C_n representatives. The value from $[0 \dots C_n - 1]$ is determined by the reader from the class of the read memory bits; the permutation is similarly determined by which of the $n!$ permutations of the representative appears in the memory. Essentially, then, one can reduce the problem to a large table look-up.

In practice, however, this approach appears infeasible for all but the smallest values of n, as there are 2^{2^n} possible ways to set the memory. Using a brute force table-look up approach rapidly becomes infeasible in terms of memory utilization and preprocessing. The first few values of C_n are $2, 4, 16, 1792, 34339072, \dots$. We have not determined a closed form for C_n; this remains an open problem.

In a similar vein, we might ask how many values D_n can be passed if we do not care whether the reader learns the adversarial permutation. In this case, all the

*Compaq Systems Research Center, Palo Alto, California.
E-mail: {broder,michaelm,moll}@pa.dec.com

equivalence classes (and not just those with $n!$ members) count, as each class determines a possible value from $[0 \ldots D_n - 1]$. The first few values in this case are $2, 12, 80, 3984, 37333248, \ldots$. A closed form for D_n also remains an open problem. We note that neither C_n or D_n appear as sequences in the famous Sloane's list [2, 3].

3 An algorithmic solution

We have devised a simple algorithmic solution which requires at most $n \log_2 n$ memory probes to determine the permutation, and uses only $n \log_2 n$ of the 2^n bits of the memory. These are both within a $1 + o(1)$ factor of optimal, since on average (a) it takes at least $\log_2(n!)$ memory probes to determine the permutation; and (b) the writer cannot transmit more than $2^n - \log_2(n!)$ bits of information if the writer has to specify a permutation as well. (Note that if the reader does not need to determine the permutation, then our algorithm still works, but we can no longer claim that it is within an $1 + o(1)$ factor of optimal. Finding non-trivial bounds for this case remains open.)

We establish the appropriate notation. Initially, we assume that the number of address lines is $n = 2^r$ for some r. We label the memory locations by n-dimensional $\{0, 1\}$ vectors. Originally the writer assigns bit values $f(x) \in \{0, 1\}$ to the vectors (locations) $x \in \{0, 1\}^n$. We denote the permutation chosen by the adversary as π and view it as a permutation of the numbers 0 to $n-1$. We use $\hat{\pi}$ to represent the action of π on vectors in the natural way: for example, if there are 4 address lines, and $\pi(0) = 0, \pi(1) = 2, \pi(2) = 1$, and $\pi(3) = 3$, then $\hat{\pi}(x) = \hat{\pi}(x_3 x_2 x_1 x_0) = x_3 x_1 x_2 x_0$. The values returned by the memory, after the adversary's evil deed, are denoted by $g(x)$, where $g(x) = f(\hat{\pi}(x))$.

The reader learns the permutation π after r rounds. For each round the reader reads the value of $g(x)$ in n distinct locations. These locations are independent of π and different from round to round. As we explain, before the permutation, the writer sets only the locations that eventually will be read. Hence $n \log_2 n$ values in the table are stored and read by our algorithm and the other locations are available for message transmission. We maintain the following invariant: after round k, for each line i, we know $\pi(i)$ modulo 2^k. Note that this invariant is trivially true before round 1. We call this the *bit-by-bit* approach. To simplify exposition, we describe the writing and the reading round by round, although in fact the writer does all the writing before the reading begins.

For the first round (round 1), the writer sets $f(x)$ to be 1 for all unit vectors $x = e_i$ for odd i, and 0 for all unit vectors $x = e_i$ for even i. The reader sets exactly one line j to 1 and all the others to 0. The memory returns 1 if and only if $\pi(j) = 1$, that is, j is mapped to an odd-numbered line.

Similarly, for round k, let the values of z range over $[0 \ldots 2^k - 1]$. The writer sets $f(x)$ for all x with a 1 in *all* positions x_i with $i = z - 1 \bmod 2^k$, exactly one 1 in one of the $n/2^k$ positions x_i with $i = z \bmod 2^k$ (call this position j), and 0's elsewhere. Note that there are n possibilities for x corresponding to the n possible values for j. The writer sets $f(x)$ to 1 if $(j - z)/2^k$ is odd and to 0 otherwise.

The reader, given the information gathered in prior rounds, can determine the permuted position of each line modulo 2^k. Hence it can compute all x such that $\hat{\pi}(x)$ has $\hat{\pi}(x)_i = 1$ in *all* positions with $i = z - 1 \bmod 2^k$, $\hat{\pi}(x)$ has exactly one 1 in one of the $n/2^k$ positions $\hat{\pi}(x)_i$ with $i = z \bmod 2^k$. Let j be the index of this particular position within x. That is, the reader can determine how to set the address bits to read values $g(x) = f(\hat{\pi}(x))$ precisely for the x's that the writer has defined for this round. Again, these reads determine for each j whether the $(k + 1)$'st bit from the right of $\pi(j)$ is 0 or 1. Our invariant is maintained, and hence only $n \cdot r = n \log_2 n$ values are set and read in the memory.

Minor improvements can be made. For example, the reader need not read n values each round, but only $n-1$ values, since the nth value to be read is determined by the other $n - 1$.

When $n = 2^r + a$, where $0 < a < 2^r$, we use an $(r + 1)$'st round for locations which are not determined by the first r bits from the right. The same argument shows that the total number of memory locations that need to be set and read is at most $n \cdot r + 2a = n\lfloor \log_2 n \rfloor + 2a$.

4 Acknowledgement

We wish to thank Mike Burrows, who computed the computable terms of the C_n and D_n sequences.

References

[1] W. S. Carter & Al., *A user programmable reconfigurable logic array*, in Proceedings of the IEEE 1986 Custom Integrated Circuits Conference., May 1986, pp. 233–235.

[2] N. J. A. Sloane, *Sloane's on-line encyclopedia of integer sequences*. Available on-line via `http://www.research.att.com/~njas/sequences/` .

[3] ———, *A Handbook of Integer Sequences*, Academic Press, 1973.

[4] *The programmable logic data book 1998*. Xilinx Inc., San Jose, CA, 1998. Available on line via `http://www.xilinx.com/partinfo/databook.htm` .