A note on unscrambling address lines

Chang-Chun Lu a, Shi-Chun Tsai b, *,1

a Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan
b Department of Computer Science and Info Engineering, National Chiao-Tung University, Hsinchu 30050, Taiwan

Received 6 March 2002; received in revised form 9 August 2002
Communicated by K. Iwama

Abstract

A writer stores some data in memory accessible via address lines. If an adversary permutes the address lines after the writer leaves the message, then how can a reader find the permutation? This is the so-called unscrambling address lines problem of Broder et al. [SODA’99, 1999, pp. 870–871]. By a divide-and-conquer approach, we give a very simple algorithm to recover the permutation. Our method is much easier to understand than Broder et al.’s previous ad hoc solution.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Divide-and-conquer; Permutation; Field programmable gate array (FPGA); Algorithms

1. Introduction

The unscrambling address lines problem [1] arose in the context of FPGA hardware design. An FPGA is a user programmable reconfigurable logic array first introduced in 1986 [2]. The basic logical element of many FPGA is equivalent to a look-up table [3]. There are three parties in the unscrambling address lines problem: a reader, a writer and an adversary. The writer stores logical 0’s and 1’s in memory with n-bit address lines, which define 2n locations for storage. After writing is complete, the adversary permutes the address lines. Therefore, the reader would read the wrong address. Then how can the reader find the permutation? For example, for n = 4 there are 16 locations in the memory: if the address lines are set to x3x2x1x0 = 1001, which indicates the 9th location before the adversary permutes the address lines. If the adversary exchanges the first (x1) and third (x3) lines, then x3x2x1x0 becomes 0011—indicating the third location.

Broder et al. [1] provide a solution and the method has been implemented and used in Compaq Systems Research Center [1]. Their method is ad hoc and short enough to be illustrated in two pages, while it is not clear enough to underline the key idea. We give a structural divide-and-conquer algorithm, which is very easy to understand and still maintains the same performance. The general mechanism of the solution is that: first, we leave some messages at certain addresses in the memory. Then we read the...
memory at these addresses. According to the output (0 or 1) from the memory, we can divide address lines into two groups. Similarly, we can further divide each group into two. If the number of address lines is \( n \), after \( \log n \) rounds we will divide address lines into \( n \) groups and each group contains exactly one line. By collecting the output from the memory in each round, we will know the permutation of each address line. Assume the number of address line is \( n = 2^r \). Then the memory locations can be represented with \( n \)-dimensional 0–1 vectors. The writer assigns 0 or 1 to the locations \( x = x_{n-1} \cdots x_1 x_0 \in [0, 1]^n \). We use \( \pi \) to denote the permutation used by the adversary, where the permutation is on the numbers from 0 to \( n-1 \). For example, let \( n = 4 \) (i.e., there are 4 address lines) and let \( \pi(0) = 1, \pi(1) = 2, \pi(2) = 3, \pi(3) = 0 \). Then \( \pi(x_3 x_2 x_1 x_0) = x_2 x_1 x_0 x_3 \), which means that when reader tries to read the bit located at \( x_3 x_2 x_1 x_0 \) it will actually get the bit stored at \( x_2 x_1 x_0 x_3 \).

We maintain the following invariant: after round \( k \), there are \( 2^k \) groups, each group has \( n/2^k \) address lines and if line \( i \) is in group \((z \mod 2^k)\), then line \( i \) will be in either group \((z \mod 2^{k+1})\) or group \((z + 2^k \mod 2^{k+1})\) after the \((k + 1)\)th round. In each round, groups are independent from each other. I.e., in each round we do not need the information of the other groups when splitting a group of lines. This feature does not hold in Broder et al.’s ad hoc method. Our algorithm makes \( n \log n \) memory probes to determine the permutation and the addresses used are different from the method by Broder et al.

\[ M(x) \]

Let \( M(x) \) refer to the value stored at address \( x \). After the writer and adversary, the reader will get the value \( M(\pi(x)) \) instead of \( M(x) \) when probing at \( x \). For \( i = 0, \ldots, n-1 \), let \( e_i = x_{n-1} \cdots x_1 x_0 \) denote the address with \( x_i = 1 \) and \( x_j = 0 \) for all \( j \neq i \). We define \( S_w \subseteq \{0, \ldots, n-1\} \), for all 0–1 string \( w \) of length at most \( r \), to be the subsets of the labels of address lines. Note that the string \( w \) also denotes a binary representation of the set index. Initially, \( S_w = \{0, \ldots, n−1\} \), where \( \varepsilon \) is the empty string. We abuse the notation a little bit by treating all the 0-strings \( \varepsilon^* \) as zero. Note that \( \varepsilon \) is zero when treated as a number. Let \( u \) be a 0–1 string of length \( k < r \) and \( S_u \) be a subset of the address labels after round \( k \). After round \( k+1 \), \( S_u \) will be evenly split into \( S_{0u} \) and \( S_{1u} \). The splitting depends on what we read from the written bits at the specific addresses. Formally, we define \( S_{w} \) ’s recursively.

**Definition 2.1.** For any positive integer \( n = 2^r \), integer \( k < r \) and permutation \( \pi \), define \( S_k = \{0, \ldots, n−1\} \). For any \( k \)-bit binary string \( w \),

\[ S_{0w} = \{ i | i \in S_w, \pi(i) \mod 2^{k+1} = w \} \]

and

\[ S_{1w} = \{ i | i \in S_w, \pi(i) \mod 2^{k+1} = w + 2^k \}, \]

where we also treat \( w \) as a \( k \)-bit binary number.

**Lemma 2.2.** For any \( i \in S_w \), we have \( \pi(i) \mod 2^{|w|} = w \) and \( |S_w| = n/2^{|w|} \), where \( |w| \) is the length or the number of bit of \( w \) and let \(|\varepsilon| = 0 \).

**Proof.** We prove by induction on \( |w| \). For \( |w| = 0 \) (i.e., \( w = \varepsilon \) and \( S_\varepsilon = \{0, \ldots, n−1\} \)), it is clear that, for any \( i \in S_\varepsilon \), \( \pi(i) \mod 1 = 0 \). Suppose it is true up to \(|w| = k \). By the definition of \( S_{0w} \) and \( S_{1w} \), it is clear for the case of \(|w| + 1 \). Similarly, we have \(|S_w| = n/2^{|w|} \). \( \square \)

Now the problem turns out to be how to find out whether \( i \in S_{0w} \) or \( i \in S_{1w} \) for each \( i \in S_w \) by probing the value at certain memory locations. We need to decide which addresses to set the values. These addresses are independent of the permutation. For the reader, the addresses are decided adaptively round by round.

2. Preliminary

Let \( n \) be the number of address lines. For convenience, let \( n = 2^r \). Let \( \pi \) be the permutation that the adversary uses to rearrange the address lines. We try to find the permutation \( \pi \) by probing at certain addresses with specific settings. Let \( x = x_{n-1} \cdots x_1 x_0 \) be an \( n \)-bit address for memory location and \( \pi(x) = x_{\pi(n-1)} \cdots x_{\pi(1)} x_{\pi(0)} \). We use the convenient notation \( \delta_R \), which is 1 if the relation \( R \) is true; 0 otherwise. A key observation is: for any integer \( j \), if \( j \mod 2^{k-1} = z \) then \( j \mod 2^k = z \) or \( z + 2^{k-1} \). This observation makes our algorithms more straightforward and is implicit in the work of Broder et al.
3. Main results

First we warm up with a trivial case, where we assume that each memory cell can store up to \( n - 1 \) different symbols instead of 0 and 1 only. In this case, the reader and writer only need to access once the addresses \( e_i, i = 0, \ldots, n - 1 \). The writer sets \( M(e_i) = i \) and the reader can determine the permutation immediately after reading at \( e_i \)'s. In reality, we can only store 0 and 1 at each memory cell and thus we will need to probe into more locations.

We will write to a location once and thus each location cannot be rewritten. We illustrate the idea by the following example in Table 1 with \( n = 8 \) and \( \pi = (03526741) \), i.e., \( \pi(7) = 0, \pi(6) = 3, \pi(5) = 2, \pi(4) = 2, \pi(3) = 6, \pi(2) = 7, \pi(1) = 4, \pi(0) = 1 \).

As we defined above \( S_w = \{0, 1, 2, 3, 4, 5, 6, 7\} \). After the first round, we divide \( S_w \) into \( S_0 = \{7, 4, 3, 1\} \) and \( S_1 = \{6, 5, 2, 0\} \). This is done by reading \( M(e_i) \), where \( i \) is added to \( S_0 \) if \( M(e_i) = 0 \); \( S_1 \) otherwise.

In other words, the address lines indexed by \( S_0 \) are permuted to even lines and to odd lines if indexed by \( S_1 \). The address lines labeled by \( S_0 \) can be permuted to 0 or 2 (mod 4) and lines in \( S_1 \) can be permuted to 1 or 3 (mod 4). Thus the further splittings of \( S_0 \) and \( S_1 \) are independent. To split \( S_0 \) we can mask the address lines in \( S_1 \) as 1 and for the lines in \( S_0 \) we allow only one line with 1. In the second round, writer and reader seemingly use different addresses for writing and reading. But the permutation makes the reader read exactly the locations that have been set values by the writer.

After round 2, we have \( S_{00} = \{7, 1\} \) and \( S_{10} = \{4, 3\} \) from \( S_0 \) and \( S_{01} = \{5, 0\} \) and \( S_{11} = \{6, 2\} \) from \( S_1 \). Similarly, we obtain \( S_{000} = \{7\}, S_{100} = \{1\}, S_{010} = \{4\} \), \( S_{110} = \{3\} \), \( S_{001} = \{0\} \), \( S_{101} = \{5\} \), \( S_{011} = \{6\} \), \( S_{111} = \{2\} \). From the above singletons, we recover the permutation. Note that each location is written exactly once.

More specifically, let \( S_w \) be a subset of the address lines. Then by Lemma 2.2, all \( i \in S_w \) have \( \pi(i) \mod 2^{|w|} = w \). Now we need to figure out which addresses to write and to read in order to split \( S_w \) into \( S_{0w} \) and \( S_{1w} \). Let \( u_{a_{n-1}} \cdots u_1 u_0 \) and \( r_{n-1} \cdots r_0 \) indicate the addresses for writer and reader, respectively, where \( u_i \)'s and \( r_i \)'s can be 0 or 1. We are interested in the following sets of addresses:

**Definition 3.1.** First define \( R_w = \{e_i \mid i = 0, \ldots, n - 1\} \). Given \( w \) and \( S_w \), define

\[
R_w = \left\{ r_{n-1} \cdots r_1 r_0 \mid r_j = 1 \text{ for } j \notin S_w; \sum_{j \in S_w} r_j = 1 \right\};
\]

\[
W_w = \left\{ u_{a_{n-1}} \cdots u_1 u_0 \mid u_j = 1, \text{ for } j \mod 2^{|w|} \neq w \right\}
\]

and

\[
\sum_{j \mod 2^{|w|}=w} u_j = 1.
\]

Both \( \sum_{j \in S_w} r_j = 1 \) and \( \sum_{j \mod 2^{|w|}=w} u_j = 1 \) in the definition make sure that exactly one bit is 1 and the others are 0. Note that \( W_w \) has nothing to do with the permutation \( \pi \). Our writer will set values at the addresses in \( W_w \) and reader will probe the addresses in \( R_w \) then split \( S_w \) into \( S_{0w} \) and \( S_{1w} \) with the returned values.

**Lemma 3.2.** Given \( \pi, w, S_w \) and \( n \), if \( r \in R_w \), then \( \pi(r) \in W_w \).

**Proof.** Let \( r = r_{n-1} \cdots r_1 r_0 \in R_w \). Then \( \pi(r) = a_{n-1} \cdots a_1 a_0 \), where \( a_{\pi(i)} = r_i \). With \( w \), we have \( j \in S_w \) iff \( \pi(j) \mod 2^{|w|} = w \). So

\[
\sum_{j \mod 2^{|w|}=w} a_j = \sum_{j \mod 2^{|w|}=w} r_{n-1}(j)
\]

\[
= \sum_{j \mod 2^{|w|}=w} r_{j'},
\]

since \( j' = \pi^{-1}(j) \)

\[
= \sum_{j' \in S_w} r_{j'} = 1.
\]

For \( j \notin S_w \), we have \( \pi(j) \mod 2^{|w|} \neq w \) and so \( a_{\pi(j)} = 1 \), since \( r_j = 1 \). Thus \( \pi(r) \in W_w \). □

The above lemma is crucial for our approach. Once the addresses are decided, for each \( u \in W_w \), the writer sets the corresponding location with 1, if there is a \( j \) such that \( j \mod 2^{|w|} = w, u_j = 1 \) and \( j \mod 2^{|w|+1} = w + 2^{|w|}; 0 \) otherwise. Thus for each \( j \in S_w \), the reader accesses the address \( r \in R_w \), where \( r_i = 0 \) for all \( i \in (S_w - \{j\}) \) and \( r_i = 1 \) for \( i \notin (S_w - \{j\}) \). Then it will return the value at \( \pi(r) \in W_w \) and \( j \) will be put in \( S_{1w} \) if the value is 1; otherwise put in \( S_{0w} \). We list the algorithms in Fig. 1.
Table 1
The values and addresses used

\( (\pi = (03526741) ) \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>0</td>
<td>1</td>
<td>00000001</td>
<td></td>
</tr>
<tr>
<td>00000010</td>
<td>1</td>
<td>0</td>
<td>00000010</td>
<td></td>
</tr>
<tr>
<td>00000100</td>
<td>0</td>
<td>1</td>
<td>00000100</td>
<td></td>
</tr>
<tr>
<td>00001000</td>
<td>1</td>
<td>0</td>
<td>00001000</td>
<td></td>
</tr>
<tr>
<td>00010000</td>
<td>0</td>
<td>0</td>
<td>00010000</td>
<td></td>
</tr>
<tr>
<td>00100000</td>
<td>1</td>
<td>1</td>
<td>00100000</td>
<td></td>
</tr>
<tr>
<td>01000000</td>
<td>0</td>
<td>1</td>
<td>01000000</td>
<td></td>
</tr>
<tr>
<td>10000000</td>
<td>1</td>
<td>0</td>
<td>10000000</td>
<td></td>
</tr>
</tbody>
</table>

\( S_0 = \{0, 1, 2, 3, 4, 5, 6, 7\} \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101011</td>
<td>0</td>
<td>0</td>
<td>01100111</td>
<td></td>
</tr>
<tr>
<td>10101110</td>
<td>1</td>
<td>1</td>
<td>01101101</td>
<td></td>
</tr>
<tr>
<td>10111010</td>
<td>0</td>
<td>1</td>
<td>01110101</td>
<td></td>
</tr>
<tr>
<td>11101010</td>
<td>1</td>
<td>0</td>
<td>11100101</td>
<td></td>
</tr>
</tbody>
</table>

\( S_1 = \{0, 2, 5, 6\} \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010111</td>
<td>0</td>
<td>0</td>
<td>10011011</td>
<td></td>
</tr>
<tr>
<td>01011101</td>
<td>1</td>
<td>1</td>
<td>10011110</td>
<td></td>
</tr>
<tr>
<td>01110101</td>
<td>0</td>
<td>0</td>
<td>10111010</td>
<td></td>
</tr>
<tr>
<td>11010101</td>
<td>1</td>
<td>1</td>
<td>11011010</td>
<td></td>
</tr>
</tbody>
</table>

\( S_0' = \{1, 7\} \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>11101111</td>
<td>0</td>
<td>1</td>
<td>01111111</td>
<td></td>
</tr>
<tr>
<td>11111110</td>
<td>1</td>
<td>0</td>
<td>11111111</td>
<td></td>
</tr>
</tbody>
</table>

\( S_0 = \{7\} \rightarrow \pi(7) = 0 \)
\( S_0 = \{1\} \rightarrow \pi(1) = 4 \)

\( S_1 = \{3, 4\} \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>10111111</td>
<td>0</td>
<td>1</td>
<td>11101111</td>
<td></td>
</tr>
<tr>
<td>11111011</td>
<td>1</td>
<td>0</td>
<td>11111011</td>
<td></td>
</tr>
</tbody>
</table>

\( S_0 = \{4\} \rightarrow \pi(4) = 2 \)
\( S_1 = \{3\} \rightarrow \pi(3) = 6 \)

\( S_1 = \{0, 5\} \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>11011111</td>
<td>0</td>
<td>0</td>
<td>11011111</td>
<td></td>
</tr>
<tr>
<td>11111010</td>
<td>1</td>
<td>1</td>
<td>11111010</td>
<td></td>
</tr>
</tbody>
</table>

\( S_0 = \{0\} \rightarrow \pi(0) = 1 \)
\( S_1 = \{5\} \rightarrow \pi(5) = 5 \)

\( S_1 = \{2, 6\} \)

<table>
<thead>
<tr>
<th>Writer</th>
<th>Value set</th>
<th>Value read</th>
<th>Reader set</th>
<th>Address line</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111111</td>
<td>0</td>
<td>1</td>
<td>10111111</td>
<td></td>
</tr>
<tr>
<td>11110111</td>
<td>1</td>
<td>0</td>
<td>11110111</td>
<td></td>
</tr>
</tbody>
</table>

\( S_0 = \{6\} \rightarrow \pi(6) = 3 \)
\( S_1 = \{2\} \rightarrow \pi(2) = 7 \)

It is worth mentioning that our method is highly parallel in nature. Once an \( S_w \) is available we can further split it into two sets without any information from the other sets. While the method by Broder et al. needs information from another set to split a set. For example, to split a set \( S_w \) with their approach, it still
Input: $n$ is the number of address lines.
Output: Assign proper values to specific locations.

**Writer**(n)
1. for $i = 0$ to $n - 1$ do $M(e_i) \leftarrow \delta_i \mod 2 = 1$
2. for all binary string $w$ with $|w| = 1$ to $\log n$ do **WriteHelper**(w);

**WriteHelper**(w)
1. for $i = 0$ to $n - 1$ do $u_i \leftarrow \delta_i \mod 2 \neq |w|$;
2. $j \leftarrow w$;
3. for $i = 0$ to $\lfloor \log n \rfloor - 1$ do
4. $M(u) \leftarrow \delta_j \mod 2 = 1$
5. $u_j \leftarrow 0$;
6. $j \leftarrow j + 2^{|w|}$;

Input: $w$: 0–1 string; $S_w$: each $j \in S_w$ has $j \mod 2^{|w|} = w$.
Output: A permutation.

**Reader**(w, $S_w$)
1. if $S_w$ has only one element then print($\pi(j) = w.$);
2. else
3. for $i = 0$ to $n - 1$ do $r_i \leftarrow \delta_i \in S_w$;
4. for all $j \in S_w$ do
5. $r_j \leftarrow 1$;
6. if ($M(r) == 1$) then add $j$ to $S_{1w}$;
7. else add $j$ to $S_{0w}$;
8. $r_j \leftarrow 0$;
9. Reader(0w, $S_{0w}$);
10. Reader(1w, $S_{1w}$);

Fig. 1. Writer and reader with write-once memory model.

needs the input $S_w \cdot 1$ in order to decide the addresses for reader [1]. Thus, it takes two sets to split a set with their approach.

It is clear that line 2 of **Writer** dominates the algorithm. Together with **WriteHelper**, the time complexity is

$$O\left(\sum_{|w| \leq \log n} n/2^{|w|}\right) = O(n \log n).$$

For **Reader**, line 3 can be handled in a step with bit manipulation instruction or, as suggested by one of the referees, by using a new variable storing the value of $u$ used in the previous step. Hence, the time complexity $T(n)$, starting with $S$, can be written with a recurrence relation: $T(n) = 2T(n/2) + O(n)$, which has the solution $T(n) = O(n \log n)$. The correctness of our algorithm can be proved formally by induction. The recursive structure is very similar to Merge Sort and Fast Fourier Transformation. There is no such simple structure in the work by Broder et al. We conclude with the following theorem.

**Theorem 3.3.** **Writer** and **Reader** probe $O(n \log n)$ locations and correctly return the permutation.

Based on the independence on splitting the sets $S_w$‘s, we can parallelize **Reader** (i.e., by allowing parallel memory access) and achieve $O(\log n)$ time complexity.

**Acknowledgements**

We thank the anonymous referees for comments on the paper.

**References**

