

# The Wonderful Exclusive-Or: Error Detection and Correction

Jayadev Misra  
The University of Texas at Austin

July 1, 2003

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Properties of Exclusive-or . . . . .	2
1.2	Dependant set . . . . .	3
<b>2</b>	<b>Small Applications</b>	<b>4</b>
2.1	Complementation . . . . .	4
2.2	Toggling . . . . .	4
2.3	Exchange . . . . .	5
2.4	Storage for Doubly-Linked Lists . . . . .	6
2.5	Hamming Distance . . . . .	6
2.6	The Game of Nim . . . . .	9
<b>3</b>	<b>Secure Communication</b>	<b>11</b>
<b>4</b>	<b>RAID Architecture</b>	<b>12</b>
<b>5</b>	<b>Error Detection</b>	<b>12</b>
5.1	Parity Check Code . . . . .	13
5.2	Horizontal and Vertical Parity Check . . . . .	14
<b>6</b>	<b>Error Correction</b>	<b>14</b>
6.1	A Naive Error-correcting Code . . . . .	14
6.2	Hamming Code . . . . .	17
6.3	Reed-Muller Code . . . . .	19

## 1 Introduction

One of the most useful logical operators is *exclusive-or*, written as  $\oplus$  in these notes. It is a binary operator, and its truth table is shown in Table 1. Encoding *true* by 1 and

	F	T
F	F	T
T	T	F

Table 1: Truth Table of Exclusive-Or

	0	1
0	0	1
1	1	0

Table 2: Exclusive-Or as addition modulo 2

*false* by 0, we get Table 2, which shows that the operator is addition modulo 2, i.e., addition in which you discard the carry.

In most of the applications we will be applying  $\oplus$  to bit strings of equal lengths, which we call *words*. The effect is to apply  $\oplus$  to the corresponding bits independently. Thus,

$$\begin{array}{r}
 0110 \\
 \oplus \\
 1011 \\
 = \\
 1101
 \end{array}$$

## 1.1 Properties of Exclusive-or

In the following expressions  $x$ ,  $y$  and  $z$  are words of the same length, 0 is a word of all zeroes, and 1 is a word of all ones.  $\bar{x}$  denotes the word obtained from  $x$  by complementing each of its bits.

- $\oplus$  is commutative:  $x \oplus y = y \oplus x$
- $\oplus$  is associative:  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- zero and complementation:  $x \oplus 0 = x$ ,  $x \oplus 1 = \bar{x}$
- inverse:  $x \oplus x = 0$ ,  $x \oplus \bar{x} = 1$

From the inverse property, we can regard  $\oplus$  as subtraction modulo 2.

Exclusive-or has many interesting relationships with other logical operators. We note a few.

- Opposite of equivalence:  $(x \equiv y) = \overline{(x \oplus y)}$
- De Morgan's law:

$$\overline{(x \oplus y)} = (\bar{x} \equiv \bar{y})$$

The right side is same as  $x \equiv y$ . So,  $\overline{(x \oplus y)} = (x \equiv y)$ .

- $\wedge$  distributes over  $\oplus$ :  $(x \oplus y) \wedge z = (x \wedge z) \oplus (y \wedge z)$ .

## 1.2 Dependant set

A nonempty set of words,  $W$ , is *dependant* iff  $\hat{W} = 0$ , where  $\hat{W}$  is the exclusive-or of all the words in  $W$ . Dependant sets are used in two applications later in these notes, in sections 4 and 6.2.

**Observation**  $W$  is dependant iff for any partition of  $W$  into subsets  $X$  and  $Y$ ,  $\hat{X} = \hat{Y}$ .

Proof:

$$\begin{aligned}
 & \hat{W} = 0 \\
 \equiv & \{W \text{ can be partitioned into } X \text{ and } Y, \text{ and using the properties of } \wedge\} \\
 & \hat{X} \oplus \hat{Y} = 0 \\
 \equiv & \{\text{add } \hat{Y} \text{ to both sides of this equation}\} \\
 & \hat{X} \oplus \hat{Y} \oplus \hat{Y} = \hat{Y} \\
 \equiv & \{\hat{Y} \oplus \hat{Y} = 0 \text{ and } \hat{X} \oplus 0 = \hat{X}\} \\
 & \hat{X} = \hat{Y}
 \end{aligned}$$

The proof of the following observation is similar to the one above and is omitted.

**Observation** If there is a partition of  $W$  into  $U$  and  $V$  such that  $\hat{U} = \hat{V}$ , then  $W$  is dependant.

Note: The two observations above say different things. The first one says that if  $W$  is dependant then for *all* partitions into  $X$  and  $Y$  we have  $\hat{X} = \hat{Y}$ , and, conversely, if for *all* partitions into  $X$  and  $Y$  we have  $\hat{X} = \hat{Y}$ , then  $W$  is dependant. The second observation is a stronger result than the later part of the first observation: if there exists any (not all) partition into  $U$  and  $V$  such that  $\hat{U} = \hat{V}$ , then  $W$  is dependant.

### Exercise 1

1. Show that  $\overline{x \oplus y} = x \oplus y$ , and  $\overline{x \oplus y} = \overline{x} \oplus \overline{y}$ .
2. Show that  $(x \oplus y = x \oplus z) \equiv (y = z)$ . As a corollary, prove that  $(x \oplus y = 0) \equiv (x = y)$ .
3. What is the condition on  $x$  and  $u$  so that  $(x \oplus u) < x$ , where  $x$  and  $u$  are numbers written in binary?
4. Let  $W'$  be a set obtained from a dependant set  $W$  by either removing an element or adding an element. Given  $W'$  determine  $W$ .

$x$	$=$	$\alpha$	1	$\beta$
$x \oplus u$	$=$	$\alpha$	0	$\gamma$
$u$	$=$	0s	1	$\beta \oplus \gamma$

Table 3: Computing  $x \oplus x \oplus u$

**Solution to Exercise 1.2[part 3]** Since  $(x \oplus u) < x$ , there is a bit position where  $x$  has a 1 and  $x \oplus u$  has a 0, and all bits to the left of this bit are identical in  $x$  and  $x \oplus u$ . So,  $x$  is of the form  $\alpha 1 \beta$  and  $x \oplus u$  is of the form  $\alpha 0 \gamma$ . Then, taking their exclusive-or, see Table 3, we find that  $u$  has a string of zeroes followed by a single 1 and then another string  $(\beta \oplus \gamma)$ . Comparing  $x$  and  $u$  in that table,  $x$  has a 1 in the position where the leading 1 bit of  $u$  appears. This is the only relevant condition. Note that  $x$  could be smaller than  $u$ .

## 2 Small Applications

### 2.1 Complementation

To complement some bit of a word is to flip it, from 1 to 0 or 0 to 1. To selectively complement the bits of  $x$  where  $y$  has a 1, simply do

$$x := x \oplus y$$

From symmetry of the right side, the resulting value of  $x$  is also a complementation of  $y$  by  $x$ . If  $y$  is a word of all 1s, then  $x \oplus y$  is the complement of (all bits of)  $x$ ; this is just an application of the law:  $x \oplus 1 = \bar{x}$ .

Suppose we want to construct a word  $w$  from  $x$ ,  $y$  and  $u$  as follows. Wherever  $u$  has a 0 bit choose the corresponding bit of  $x$ , and wherever it has 1 choose from  $y$ , see the example below.

$$\begin{aligned} u &= 0\ 1\ 0\ 1 \\ x &= 1\ 1\ 0\ 0 \\ y &= 0\ 0\ 1\ 1 \\ w &= 1\ 0\ 0\ 1 \end{aligned}$$

Then  $w$  is, simply,  $((x \oplus y) \wedge u) \oplus x$ .

### Exercise 2

Prove this result.

### 2.2 Toggling

Consider a variable  $x$  that takes two possible values,  $m$  and  $n$ . We would like to *toggle* its value from time to time: if it is  $m$ , it becomes  $n$  and vice versa. There is a neat way to do it using exclusive-or. Define a variable  $t$  which is initially set to  $m \oplus n$  and it never changes.

toggle::  $x := x \oplus t$

To see why this works, check out the two cases: before the assignment, let the value of  $x$  be  $m$  in one case and  $n$  in the other. For  $x = m$ , the toggle sets  $x$  to  $m \oplus t$ , i.e.,  $m \oplus m \oplus n$ , which is  $n$ . The other case is symmetric.

### Exercise 3

Variable  $x$  assumes the values of  $p$ ,  $q$  and  $r$  in cyclic order, starting with  $p$ . Write a code fragment to assign the next value to  $x$ , using  $\oplus$  as the primary operator in your code. You will have to define additional variables and assign them values along with the assignment to  $x$ .  $\square$

**Solution** Define two other variables  $y$  and  $z$  whose values are related to  $x$ 's by the following invariant:

$$x, y, z = t, t \oplus t', t \oplus t''$$

where  $t'$  is the next value in cyclic order after  $t$  (so,  $p' = q$ ,  $q' = r$  and  $r' = p$ ), and  $t''$  is the value following  $t'$ . The invariant is established initially by letting

$$x, y, z = p, p \oplus q, p \oplus r$$

The cyclic assignment is implemented by

```

 $x := x \oplus y;$ 
 $y := y \oplus z;$ 
 $z := y \oplus z$ 

```

Show that if  $x, y, z = t, t \oplus t', t \oplus t''$  before these assignments, then  $x, y, z = t', t' \oplus t'', t' \oplus t$  after the assignments (note:  $t''' = t$ ).

## 2.3 Exchange

Here is a truly surprising application of  $\oplus$ . If you wish to exchange the values of two variables you usually need a temporary variable to hold one of the values. You can exchange *without* using a temporary variable. The following assignments exchange the values of  $x$  and  $y$ .

```

 $x := x \oplus y;$ 
 $y := x \oplus y;$ 
 $x := x \oplus y$ 

```

To see that this program actually exchanges the values, suppose the values of  $x$  and  $y$  are  $X$  and  $Y$  before the exchange. The following annotated program shows the values they have at each stage of the computation; I have used backward substitution to construct this annotation. The code is to the left and the annotation to the right in a line.

$$\begin{array}{ll}
& y = Y, (x \oplus y) \oplus y = X, \text{i.e., } x = X, y = Y \\
x := x \oplus y; & \\
& x \oplus (x \oplus y) = Y, (x \oplus y) = X, \text{i.e., } y = Y, (x \oplus y) = X \\
y := x \oplus y; & \\
& x \oplus y = Y, y = X \\
x := x \oplus y & \\
& x = Y, y = X
\end{array}$$

## 2.4 Storage for Doubly-Linked Lists

Each node  $x$  in a doubly-linked list stores a data value, a left pointer,  $x.left$ , to a node and a right pointer,  $x.right$ , to a node. One or both pointers may be *nil*, a special value. The doubly-linked list has the property that for each node  $x$

$$\begin{array}{l}
x.left \neq nil \Rightarrow x.left.right = x \\
x.right \neq nil \Rightarrow x.right.left = x
\end{array}$$

Typically, each node needs storage for the data value and for two pointers. The storage for two pointers can be reduced to the storage needed for just one pointer; store  $x.left \oplus x.right$  at  $x$ . How do we retrieve the two pointer values from this one value? During a computation, node  $x$  is reached from either the left or the right side; therefore, either  $x.left$  or  $x.right$  is known. Applying  $\oplus$  to the known pointer value and  $x.left \oplus x.right$  yields the other pointer value; see the treatment of toggling in section 2.2. Here, *nil* should be treated as 0.

We could have stored  $x.left + x.right$  and subtracted the known value from this sum; exclusive-or is faster to apply and it avoids overflow problems.

Sometimes, nodes in a doubly-linked list are reached from some node outside the list; imagine an array each of whose entries points to a node in a doubly-linked list. The proposed pointer compression scheme is not useful then because you can reach a node without knowing the value of any of its pointers.

**Note:** These kinds of pointer manipulations are often prevented by the compiler of a high-level language through type checks. I don't advocate such manipulations except when you are programming in an assembly language, and you need to squeeze out the last drop of performance. Even then see if there are better alternatives; often a superior data structure or algorithm gives you far better performance than clever tricks!<sup>1</sup>

## 2.5 Hamming Distance

The *Hamming distance*—henceforth, simply called *distance*—between two words is the number of positions where they differ. Thus the distance between 1 0 0 1 and 1 1 0 0 is 2. This is the number of 1s in  $1\ 0\ 0\ 1 \oplus 1\ 1\ 0\ 0$ , which is 0 1 0 1.

<sup>1</sup>“The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague”. From “The Humble Programmer” by Edsger W. Dijkstra, 1972 Turing Award lecture[1].

Distance is a measure of how similar two words are, smaller the distance greater the similarity. Observe the following properties of distance. Below,  $x$ ,  $y$  and  $z$  are words and  $d(x, y)$  is the distance between  $x$  and  $y$ .

- $(d(x, y) = 0) \equiv (x = y)$
- $d(x, y) \geq 0$
- $d(x, y) = d(y, x)$
- (Triangle Inequality)  $d(x, y) + d(y, z) \geq d(x, z)$

In order to prove these properties, you need some facts about the number of 1s in a word. Let  $\text{count}(x)$  be the number of 1s in  $x$ . Then,  $d(x, y) = \text{count}(x \oplus y)$ . We have the following properties of  $\text{count}$ .

$$\begin{aligned} \text{count}(x) &\geq 0 \\ (\text{count}(x) = 0) &\equiv (x = 0) \\ \text{count}(x) + \text{count}(y) &= \text{count}(x \oplus y) + 2f, \text{ for some nonnegative } f. \end{aligned}$$

The first two properties are easy to see. For the last property, let  $f$  be the number of positions where  $x$  and  $y$  both have 1s. In  $x \oplus y$ , the bit values at these positions are 0, and at every other position the number of 1s in  $x \oplus y$  is same as the number of 1s in that position of  $x$  and  $y$  combined. Therefore,  $x \oplus y$  has  $2f$  fewer ones than  $x$  and  $y$  combined.

It follows from the last property that

$$\begin{aligned} \text{count}(x) + \text{count}(y) &\geq \text{count}(x \oplus y) \\ \text{count}(x) + \text{count}(y) &\text{ has the same parity (even or odd) as } \text{count}(x \oplus y) \end{aligned}$$

The proof of Triangle Inequality is now immediate.

$$\begin{aligned} &\text{count}(x) + \text{count}(y) \geq \text{count}(x \oplus y) \\ \Rightarrow &\text{ \{replace } x \text{ by } x \oplus y \text{ and } y \text{ by } y \oplus z\} \\ &\text{count}(x \oplus y) + \text{count}(y \oplus z) \geq \text{count}(x \oplus y \oplus y \oplus z) \\ \equiv &\text{ \{simplify\} } \\ &\text{count}(x \oplus y) + \text{count}(y \oplus z) \geq \text{count}(x \oplus z) \\ \equiv &\text{ \{ } d(x, y) = \text{count}(x \oplus y) \text{; similarly for the other terms} \} } \\ &d(x, y) + d(y, z) \geq d(x, z) \end{aligned}$$

Hamming distance is essential to the study of error detection (section 5) and error correction (section 6).

#### Exercise 4

A word  $x$  has *even parity* if  $\text{count}(x)$  is even, otherwise it has *odd parity*. Show that two words of identical parity (both even or both odd) have even distance, and words of different parity have odd distance.  $\square$

**Solution** In the following proof we start with a property of *count*.

$$\begin{aligned}
& \text{count}(x) + \text{count}(y) \text{ has the same parity (even or odd) as } \text{count}(x \oplus y) \\
\Rightarrow & \{ \text{writing } \text{even}(n) \text{ to denote that number } n \text{ is even} \} \\
& \text{even}(\text{count}(x) + \text{count}(y)) \equiv \text{even}(\text{count}(x \oplus y)) \\
\equiv & \{ \text{for any two integers } p \text{ and } q, \text{even}(p + q) = (\text{even}(p) \equiv \text{even}(q)); \\
& \text{let } p \text{ be } \text{count}(x) \text{ and } q \text{ be } \text{count}(y) \} \\
& (\text{even}(\text{count}(x)) \equiv \text{even}(\text{count}(y))) \equiv \text{even}(\text{count}(x \oplus y)) \\
\equiv & \{ \text{count}(x \oplus y) = d(x, y) \} \\
& (\text{even}(\text{count}(x)) \equiv \text{even}(\text{count}(y))) \equiv \text{even}(d(x, y))
\end{aligned}$$

The term  $\text{even}(\text{count}(x))$  stands for “ $x$  has even parity”. Therefore, the first term in the last line of the above proof,  $(\text{even}(\text{count}(x)) \equiv \text{even}(\text{count}(y)))$ , denotes that  $x$  and  $y$  have identical parity. Hence, the conclusion in the above proof says that the distance between  $x$  and  $y$  is even iff  $x$  and  $y$  have identical parity.  $\square$

### Exercise 5

Let  $w_1, w_2, \dots, w_N$  be a set of *unknown* words. Let  $W_i$  be the exclusive-or of all the words except  $w_i$ ,  $1 \leq i \leq N$ . Given  $W_1, W_2, \dots, W_N$ , can you determine the values of  $w_1, w_2, \dots, w_N$ ? You can only apply  $\oplus$  on the words. You may prefer to attack the problem without reading the following hint.

Hint:

1. Show that the problem can be solved when  $N$  is even.
2. Show that the problem can not be solved when  $N$  is odd.

A more general problem:

Investigate how to solve a general system of equations which use  $\oplus$  as the only operator. For example, the equations may be:

$$\begin{aligned}
w_1 \oplus w_2 \oplus w_4 &= 1\ 0\ 0\ 1\ 1 \\
w_1 \oplus w_3 &= 1\ 0\ 1\ 1\ 0 \\
w_2 \oplus w_3 &= 0\ 0\ 0\ 0\ 1 \\
w_3 \oplus w_4 &= 1\ 1\ 0\ 1\ 1
\end{aligned}
\quad \square$$

**Solution** Let  $S$  denote the exclusive-or of all the unknowns, i.e.,  $S = w_1 \oplus w_2 \oplus \dots \oplus w_N$ . Then  $W_i = S \oplus w_i$ .

1. For even  $N$ :

$$\begin{aligned}
& W_1 \oplus W_2 \oplus \dots \oplus W_N \\
= & \{ W_i = S \oplus w_i \} \\
& (S \oplus W_1) \oplus (S \oplus W_2) \oplus \dots \oplus (S \oplus W_N) \\
= & \{ \text{Regrouping terms} \} \\
& (S \oplus S \oplus \dots \oplus S) \oplus (w_1 \oplus w_2 \oplus \dots \oplus w_N) \\
= & \{ \text{the first operand has an even number of } S \} \\
& 0 \oplus (w_1 \oplus w_2 \oplus \dots \oplus w_N) \\
= & \{ \text{the last operand is } S \} \\
& S
\end{aligned}$$



Once  $S$  is determined, we can compute each  $w_i$  because

$$\begin{aligned}
& S \oplus W_i \\
= & \{W_i = S \oplus w_i\} \\
& S \oplus S \oplus w_i \\
= & \{S \oplus S = 0\} \\
& w_i
\end{aligned}$$

2. For odd  $N$ : We show that any term that we compute is exclusive-or of some subset of  $w_1, w_2, \dots, w_N$ , and *the subset size is even*. Therefore, we will never compute a term which represents, say,  $w_1$  because then the subset size is odd.

To motivate the proof, suppose we have  $N = 5$ , so  $W_1 = w_2 \oplus w_3 \oplus w_4 \oplus w_5$ ,  $W_2 = w_1 \oplus w_3 \oplus w_4 \oplus w_5$ ,  $W_3 = w_1 \oplus w_2 \oplus w_3 \oplus w_4$ ,  $W_4 = w_1 \oplus w_2 \oplus w_3 \oplus w_5$ ,  $W_5 = w_1 \oplus w_2 \oplus w_3 \oplus w_4$ . Initially, each of the terms,  $W_1, W_2$  etc., is represented by a subset of unknowns of size 4. Now, suppose we compute a new term,  $W_1 \oplus W_4$ ; this represents  $w_2 \oplus w_3 \oplus w_4 \oplus w_5 \oplus w_1 \oplus w_2 \oplus w_3 \oplus w_5$ , which is same as  $w_1 \oplus w_4$ , again a subset of even number of terms.

The proof is as follows. Initially the proposition holds because each  $W_i$  is the exclusive-or of all but one of the unknowns, namely  $w_i$ ; so the corresponding subset size is  $N - 1$ , which is even since  $N$  is odd.

Whenever we apply  $\oplus$  to any two terms: (1) either their subsets have no common unknowns, so the resulting subset contains all the unknowns from both subsets, and its size is the sum of both subset sizes, which is even, or (2) the subsets have some number of common unknowns, which get cancelled out from both subsets, again yielding an even number of unknowns for the resulting subset.  $\square$

## 2.6 The Game of Nim

The game of Nim is a beautiful illustration of the power of the exclusive-or operator.

The game is played by two players who take turns in making moves. Initially, there are several piles of chips and in a move a player may remove any positive number of chips from a single pile. A player loses when he can't make a move, i.e., all piles are empty. We develop the conditions for a specific player to win.

Suppose there is a single pile. The first player wins by removing all chips from that pile. Now suppose there are two piles, each with one chip, call this initial state (1,1). The first player is forced to empty out one pile, and the second player then removes the chip from the other pile, thus winning the game. Finally, consider two piles, one with one chip and the other with two chips. If the first player removes all chips from either pile, he loses. But if he removes one chip from the bigger pile, he creates the state (1,1) which leads to a defeat for the second player, from the previous argument.

**The Underlying Mathematics** Consider the number of chips in a pile as a word (a bit string) and take the exclusive-or of all the words. Call the state *losing* if the result is 0, *winning* otherwise. Thus, the state (1,1) results in 0, a losing state, whereas (1,2) gives  $0\ 1 \oplus 1\ 0 = 1\ 1$ , which is a winning state. The final state, where all piles are

empty, is a losing state. The mnemonics, *losing* and *winning*, signify the position of a player: a player who has to make a move in a winning state has a winning strategy, i.e., if he makes the right moves he wins no matter what his opponent does; a player in a losing state will definitely lose provided his opponent makes the right moves. So, one of the players has a winning strategy based on the initial state. Of course, either player is allowed to play stupidly and squander a winning position.

The proof of this result is based on the following state diagram. We show that any possible move in a losing state can only lead to a winning state, thus a player who has to move in this state can not do anything but hope that his opponent makes a mistake! A player in a winning state has at least one move to transform the state to losing; of course, he can make a wrong move and remain in the winning state, thus handing his opponent the mistake he was hoping for. Next, we prove the claims made in this diagram.

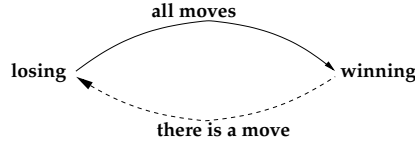


Figure 1: State transitions in the game of Nim

A move reduces a pile of  $p$  chips to  $q$  chips,  $0 \leq q < p$ . Let the exclusive-or of the remaining piles be  $s$ . Before the move, exclusive-or of all piles was  $s \oplus p$ . After the move it is  $s \oplus q$ . First, we show that in a losing state, i.e.,  $s \oplus p = 0$ , all possible moves establish a winning state, i.e.,  $s \oplus q \neq 0$ .

$$\begin{aligned}
 & s \oplus q \\
 = & \{p \oplus p = 0\} \\
 & s \oplus q \oplus p \oplus p \\
 = & \{s \oplus p = 0\} \\
 & p \oplus q \\
 \neq & \{p \neq q\} \\
 & 0
 \end{aligned}$$

Now, we show that there is a move in the winning state to take it to losing state. Let the exclusive-or of all piles be  $u$ ,  $u \neq 0$ . Let  $x$  be any pile which has a 1 in the same position as the leading 1 bit of  $u$ . So,

$$\begin{aligned}
 u &= 0s \ 1 \ \gamma \\
 x &= \alpha \ 1 \ \beta
 \end{aligned}$$

The winning move is to replace  $x$  by  $x \oplus u$ . We show that (1)  $x \oplus u < x$ , and (2) the exclusive-or of the resulting set of piles is 0, i.e., the state after the move is a losing state.

Proof of (1):  $x \oplus u = \alpha 0(\beta \oplus \gamma)$ . Comparing  $x$  and  $x \oplus u$ , we have  $x \oplus u < x$ .

Proof of (2): The exclusive-or of the piles before the move is  $u$ ; so, the exclusive-or of the piles except  $x$  is  $x \oplus u$ . Hence, the exclusive-or of the piles after the move is  $(x \oplus u) \oplus (x \oplus u)$ , which is 0.

### Exercise 6

In a winning state let  $y$  be a pile which has a 0 in the same position as the leading bit of  $u$ . Show that removing any number of chips from  $y$  leaves a winning state.

**Solution** The forms of  $u$  and  $y$  are as follows.

$$\begin{aligned} u &= 0s \ 1 \ \gamma \\ y &= \alpha \ 0 \ \beta \end{aligned}$$

Suppose  $y$  is reduced to  $y'$  and the exclusive-or of the resulting set is 0. Then  $u \oplus y \oplus y' = 0$ , or  $y' = u \oplus y$ . Hence,  $y' = \alpha \ 1 \ (\beta \oplus \gamma)$ . So,  $y' > y$ ; that is, such a move is impossible.

## 3 Secure Communication

The problem in secure communication is for a sender to send a message to a receiver so that no eavesdropper can read the message during transmission. It is impossible to ensure that no one else can see the transmission; therefore, the transmitted message is usually encrypted so that the eavesdropper can not decipher the real message. In most cases, the sender and the receiver agree on a transmission protocol; the sender encrypts the message in such a fashion that only the receiver can decrypt it.

In this section, I describe a very simple encryption (and decryption) scheme whose only virtue is simplicity. Usually, this form of transmission can be *broken* by a determined adversary. There are now very good methods for secure transmission, see Rivest, Shamir and Adelman [2].

The sender first converts the message to be sent to a bit string, by replacing each symbol of the alphabet by its ascii representation, for instance. This string is usually called the *plaintext*. Next, the plaintext is broken up into fixed size blocks, typically around 64 bits in length, which are then encrypted and sent. For encryption, the sender and the receiver agree on a key  $k$ , which is a bit string of the same length as the block. To send a string  $x$ , the sender transmits  $y$ , where  $y = x \oplus k$ . The receiver, on receiving  $y$ , computes  $y \oplus k$  which is  $(x \oplus k) \oplus k$ , i.e.,  $x$ , the original message. An eavesdropper can only see  $y$  which appears as pure gibberish. The transmission can only be decrypted by some one in possession of key  $k$ .

There are many variations on this simple scheme. It is better to have a long key, much longer than the block length, so that successive blocks are encrypted using different strings. When the bits from  $k$  run out, wrap around and start reusing the bits of  $k$  from the beginning. Using a longer key reduces the possibility of the code being broken.

This communication scheme is simple to program; in fact, encryption and decryption have the same program. Each operation is fast, requiring time proportional to a

block length for encryption (and decryption). Yet, the scheme has significant drawbacks. Any party who has the key can decode the message. More important, any one who can decode a single block can decode all blocks (assuming that the key length is same as the block length), because given  $x$  and  $y$  where  $y = x \oplus k$ ,  $k$  is simply  $x \oplus y$ . Also, the sender and the receiver will have to agree on a key before the transmission takes place, so the keys have to be transmitted first in a secure manner, a problem known as *key exchange*. For these and other reasons, this form of encryption is rarely used in high security applications.

### Exercise 7

The following technique has been suggested for improving the security of transmission. The sender encrypts the first block using the key  $k$ . He encrypts subsequent blocks by using the previous encrypted block as the key. Is this secure? How about using the plaintext of the previous block as the key? Suppose a single block is deciphered by the eavesdropper; can he then decipher all blocks, or all subsequent blocks?

## 4 RAID Architecture

The following scenario is common in corporate data centers. A large database, consisting of millions of records, is stored on a number of disks. Since disks may fail, data is stored on backup disks also. One common strategy is to partition the records of the database and store each partition on a disk, and also on a backup disk. Then, failure of one disk causes no difficulty. Even when multiple disks fail, the data can be recovered provided both disks for a partition do not fail.

There is a different strategy, known as RAID, that has gained popularity because it needs only one additional disk beyond the primary data disks, and it can tolerate failure of any one disk.

Imagine that the database is a matrix of bits, where each row represents a record, and each column a specific bit in all records. Store each column on a separate disk and store the exclusive-or of all columns on a backup disk. Let  $c_i$  denote the  $i^{th}$  column,  $1 \leq i \leq N$ , in the database. Then the backup column,  $c_0$  is given by  $c_0 = c_1 \oplus \dots \oplus c_N$ . Therefore, the set of columns,  $c_0 \dots c_N$ , is a dependant set, see section 1.2. Then, any column  $c_i$ ,  $0 \leq i \leq N$ , is the exclusive-or of the remaining columns. Therefore, the contents of any failed disk can be reconstructed from the remaining disks.

## 5 Error Detection

Message transmission is vulnerable to noise, which may cause portions of a message to be altered. For example, message 1 1 0 0 1 may become 1 0 1 0 1. In this section, we study methods by which a receiver can determine that the message has been altered, and thus request retransmission. In the next section, we discuss methods by which a receiver can correct (some of) the errors, thus avoiding retransmission.

Typically, a long message is broken up into fixed size blocks (extra bits, which can be distinguished from the real ones, are added at the end of the message so that the

string fits exactly into some number of blocks; typically, the extra bits carry information about the length and composition of the block). Henceforth, each block is transmitted independently, and we concentrate on the transmission of a single block.

## 5.1 Parity Check Code

Consider the following input string where spaces separate the blocks.

011 100 010 111

The sender appends a bit at the end of each block so that each 4-bit block has an even number of 1s. This additional bit is called a *parity* bit, and each block is said to have *even parity*. After addition of parity bits, the input string shown above becomes,

0110 1001 0101 1111

This string is transmitted. Suppose two bits are flipped during transmission, as shown below; the flipped bits are underlined.

0110 1000 0101 0111

Note that the flipped bit could be a parity bit or one of the original ones. Now each erroneous block has odd parity, and the receiver can identify all such blocks. It then asks for retransmissions of those blocks.

If two bits (or any even number) of bits of a block get flipped, the receiver cannot detect the error. This is a serious problem, so simple parity check is rarely used. In practice, the blocks are much longer (than 3, shown here) and many additional bits are used for error detection.

**Is parity coding any good?** How much is error probability reduced if you add a single parity bit? The analysis here uses elementary probability theory. Let  $p$  be the probability of error in the transmission of a single bit<sup>2</sup>. The probability of correct transmission of a single bit is  $q$ , where  $q = 1 - p$ . The probability of correct transmission of a  $b$  bit block is  $q^b$ . Therefore, the probability that there is an undetected error in the block is  $1 - q^b$ . For  $p = 10^{-4}$  and  $b = 12$ , this probability is around  $1.2 \times 10^{-3}$ .

With the addition of a parity bit, we have to send  $b + 1$  bits. But we can detect one error if it arises. So, the probability of undetected error is  $1 -$  the probability that there is no error in transmission — the probability that there is a single error in transmission. Probability that there is no error in transmission is  $q^{b+1}$ . Probability that there is a single error can be computed as follows: it is the probability that the first bit is incorrectly transmitted and the remaining bits are correctly transmitted ( $p \times q^b$ ) plus the probability that the second bit is incorrectly transmitted and the remaining bits are correctly transmitted (same probability) plus . . . . Therefore, probability of single error is  $(b+1) \times p \times q^b$ . Hence, the probability of undetected error is  $1 - q^{b+1} - (b+1) \times p \times q^b$ . Setting  $b, p, q = 12, 10^{-4}, 1 - 10^{-4}$ , this probability is around  $9.4 \times 10^{-7}$ , several orders of magnitude smaller than  $1.2 \times 10^{-3}$ .

<sup>2</sup>I am assuming that all errors are independent, a thoroughly false assumption when burst errors can arise.

1	0	1	1	1
0	1	1	1	1
1	1	1	0	1
0	0	1	1	0
0	0	0	1	1

Table 4: Adding parity bits to rows and columns

## 5.2 Horizontal and Vertical Parity Check

A simple generalization of the simple parity check scheme is described next. We regard the data as a matrix of bits, not just a linear string. For instance, we may break up a 16 bit block into 4 subblocks, each of length 4. We regard each subblock as the row of a matrix, so, column  $i$  is the sequence of  $i^{th}$  bits from each subblock. Then we add parity bits to each row and column, and a single bit for the entire matrix. In Table 4, 4 subblocks of length 4 each are transformed into 5 subblocks of length 5 each.

We can now detect odd number of errors in rows *or* columns. If two adjacent bits in a row get altered, the row parity remains the same but the column parities for the affected columns are altered.

The most common use of this scheme is in transmitting a sequence of ascii characters. Each character is a 8-bit string, which we regard as a row.

### Exercise 8

Show a small error pattern in Table 4 which will not be detected by this scheme. □

## 6 Error Correction

In many practical situations, retransmission is expensive or impossible. For example, when the sender is a spacecraft from a distant planet, the time of transmission can be measured in days; so, retransmission adds significant delay, and the spacecraft will have to store a huge amount of data awaiting any retransmission request. Even more impractical is to request retransmission of the music on a CD whose artist is dead.

### 6.1 A Naive Error-correcting Code

When retransmission is not feasible, the sender encodes the messages in such a way that the receiver can detect and correct some of the errors. As an example, suppose that the sender plans to send a 2 bit message. Adding a parity bit increases the block length to 3. Repeating the original 2-bit message after that gives a 5-bit block, as shown Table 5.

Each of the possible blocks—in this case, 5 bit blocks—is called a *codeword*. Codewords are the only possible messages (blocks) that will be sent. So, if the sender plans to send 11, he will send 11011. In the example of Table 5, there are only four 5-bit codewords, instead of 32 possible ones. This means that it will take longer to

Original	With Parity	Additional Bits
00	00 <b>0</b>	00 <b>000</b>
01	01 <b>1</b>	01 <b>101</b>
10	10 <b>1</b>	10 <b>110</b>
11	11 <b>0</b>	11 <b>011</b>

Table 5: Coding for error correction; parity bits are in bold

Codeword	Received Word	Hamming Distance
00000	11010	3
01101	11010	4
10110	11010	2
11011	11010	<u>1</u>

Table 6: Computing Hamming Distance to Codewords

transmit a message, because many redundant bits will be transmitted. The redundancy allows us to detect and correct errors.

For the given example, we can detect two errors and correct one error in transmission. Suppose 11011 is changed to 11010. The receiver observes that this is not a codeword, so he has detected an error. He corrects the error by looking for the *nearest* codeword, the one which has the smallest Hamming distance from the received word. The computation is shown in Table 6. As shown there, the receiver concludes that the original transmission is 11011.

Now suppose two bits of the original transmission are altered, so that 11011 is changed to 10010. The computation is shown in Table 7. The receiver will detect that there is an error, but based on distances, he will assume that 10110 was sent. We can show that this particular encoding can correct one error only. The number of errors that can be detected/corrected depends on the Hamming distance among the codewords, as given by the following theorem.

**Theorem 1** Let  $h$  be the Hamming distance between the nearest two codewords. It is possible to detect any number of errors less than  $h$  and correct any number of errors less than  $h/2$ .

Codeword	Received Word	Hamming Distance
00000	10010	2
01101	10010	5
10110	10010	<u>1</u>
11011	10010	2

Table 7: Hamming Distance when there are two errors

Block length	$h = 3$	$h = 5$	$h = 7$
5	4	2	-
7	16	2	2
10	72-79	12	2
16	2560-3276	256-340	36-37

Table 8: Number of codewords for given block lengths and  $h$

Proof: The statement of the theorem is as follows. Suppose codeword  $x$  is transmitted and string  $y$  received.

1. if  $d(x, y) < h$ : the receiver can detect if errors have been introduced during transmission.
2. if  $d(x, y) < h/2$ : the receiver can correct the errors, if any. It picks the closest codeword to  $y$ , and that is  $x$ .

Proof of (1): The distance between any two distinct codewords is at least  $h$ . The distance between  $x$  and  $y$  is less than  $h$ . So, either  $x = y$  or  $y$  is not a codeword. Therefore, the receiver can detect errors as follows: if  $y$  is a codeword, there is no error in transmission, and if  $y$  is not a codeword, the transmission is erroneous.

Proof of (2): We show that the closest codeword to  $y$  is  $x$ , i.e., for any other codeword  $z$ ,  $d(y, z) > d(x, y)$ .

$$\begin{aligned}
& d(x, y) + d(y, z) \\
\geq & \quad \{\text{triangle inequality}\} \\
& d(x, z) \\
\geq & \quad \{x \text{ and } z \text{ are codewords, and their distance is at least } h\} \\
& h \\
> & \quad \{d(x, y) < h/2. \text{ Hence, } h > 2d(x, y)\} \\
& 2d(x, y)
\end{aligned}$$

From this proof

$$\begin{aligned}
d(x, y) + d(y, z) &> 2d(x, y), \text{ or} \\
d(y, z) &> d(x, y)
\end{aligned}$$

### Exercise 9

Compute the nearest distance among the codewords in Table 5. □

It is clear from Theorem 1 that we should choose codewords to maximize  $h$ . But with a fixed block length, the number of codewords decreases drastically with increasing  $h$ . Table 8 shows the number of codewords for certain values of the block length and  $h$ ; an entry like 72-79 denotes that the exact value is not known, but it lies within the given interval. Note that the decrease along a row is quite dramatic.



0	0	1	1	1	1	0	1	0	1	1	0	1
d	d	d	d	d	c	d	d	d	c	d	c	c
13	12	11	10	9	8	7	6	5	4	3	2	1
		*	*	*	*		*		*	*		*

Table 9: Hamming Code Transmission

The coding scheme described in this section can correct one error, as follows. Suppose the sender wishes to send a bit string  $b$ ; then it sends  $bpb$ , where  $p$  is the parity bit. At most one error is introduced during transmission, so the receiver sees  $b'p'b''$  where the first  $b$  could be altered to  $b'$ ,  $p$  altered to  $p'$  and the second copy of  $b$  to  $b''$ . The receiver can deduce the following. If both  $b'p'$  and  $p'b''$  have even parity, then there is no error and  $b' = b'' = b$ . If one side, say  $b'p'$  has odd parity and  $p'b''$  has even parity then  $b'' = b$ ; the error occurs in  $b'$ . If both  $b'p'$  and  $p'b''$  have odd parity, then the parity bit has been corrupted and  $b' = b'' = b$ .

This scheme is extremely wasteful in the use of transmitted bits; only one error can be corrected at the expense of transmitting more than double the number of bits. We will see a more efficient scheme in the next section.

#### Exercise 10

Prove that the parity check code of section 5.1 can be used to detect at most one error, but can not be used to correct any error.

## 6.2 Hamming Code

The coding scheme described in this section was developed by Hamming, a pioneer in Coding theory who introduced the notion of Hamming distance. It requires only  $\log n$  extra bits, called *check bits*, and it corrects at most one error in an  $n$ -bit transmission. The novel idea is to transmit in the check bits the *positions* where the data bits are 1. Since it is impractical to actually transmit all the positions, we will instead transmit an encoding of them, using exclusive-or. Also, since the check bits can be corrupted as easily as the data bits, we treat them symmetrically, and also send the positions where the check bits are 1. More precisely, we regard each position number in the transmitted string as a word, and encode the check bits in such a way that:

- HC:: the set of position numbers where the data bits and check bits are 1 form a dependant set, i.e., the exclusive-or of these positions, regarded as words, is 0 (see section 1.2).

Let us look at an example where HC rule has been applied.

**Example** Consider transmission of a 13 bit string, as shown in Table 9. The data bits are labeled d and the check bits c. The positions where 1s appear are labeled by \*. They form a dependant set; check that

$$\begin{array}{rcl}
& 1\ 0\ 1\ 1 & (=11) \\
\oplus & & \\
& 1\ 0\ 1\ 0 & (=10) \\
\oplus & & \\
& 1\ 0\ 0\ 1 & (=9) \\
\oplus & & \\
& 1\ 0\ 0\ 0 & (=8) \\
\oplus & & \\
& 0\ 1\ 1\ 0 & (=6) \\
\oplus & & \\
& 0\ 1\ 0\ 0 & (=4) \\
\oplus & & \\
& 0\ 0\ 1\ 1 & (=3) \\
\oplus & & \\
& 0\ 0\ 0\ 1 & (=1) \\
= & 0\ 0\ 0\ 0 & \quad \square
\end{array}$$

The question for the sender is where to store the check bits (we have stored them in positions 8, 4, 2 and 1 in the example above) and how to assign values to them so that the set of positions is dependant. The question for the receiver is how to decode the received string and correct a possible error.

**Receiver** Let  $P$  be the set of positions where the transmitted string has 1s and  $P'$  where the received string has 1s. From the assumption that there is at most one error, we have either  $P = P'$ ,  $P' = P \cup \{t\}$ , or  $P = P' \cup \{t\}$ , for some position  $t$ ; the latter two cases arise when the bit at position  $t$  is flipped from 0 to 1, and 1 to 0, respectively. From rule HC,  $\hat{P} = 0$ , where  $\hat{P}$  is the exclusive-or of the words in  $P$ .

The receiver computes  $\hat{P}'$ . If  $P = P'$ , he gets  $\hat{P}' = \hat{P} = 0$ . If  $P' = P \cup \{t\}$ , he gets  $\hat{P}' = \hat{P} \oplus \{t\} = 0 \oplus \{t\} = t$ . If  $P = P' \cup \{t\}$ , he gets  $\hat{P}' = \hat{P} \oplus \{t\}$ , or  $\hat{P}' = \hat{P} \oplus \{t\} = 0 \oplus \{t\} = t$ . Thus, in both cases where the bit at  $t$  has been flipped,  $\hat{P}' = t$ . If  $t \neq 0$ , the receiver can distinguish error-free transmission from erroneous transmission and correct the error in the latter case.

**Sender** We have seen from the previous paragraph that there should not be a position numbered 0, because then error-free transmission can not be distinguished from one where the bit at position 0 has been flipped. Therefore, the positions in the transmitted string are numbered starting at 1. Each position is an  $n$ -bit word. And, we will employ  $n$  check bits.

Check bits are put at every position which is a power of 2 and the remaining bits are data bits. In the example given earlier, check bits are put at positions 1, 2, 4 and 8, and the remaining nine bits are data bits. So the position of any check bit as a word has a single 1 in it. Further, no two check bit position numbers have 1s in the same place.

Let  $C$  be the set of positions where the check bits are 1s and  $D$  the positions where the data bits are 1s. We know  $D$ , but we don't know  $C$  yet, because check bits have not been assigned values. We show next that  $C$  is uniquely determined from rule HC.

From rule HC,  $\hat{C} \oplus \hat{D} = 0$ . Therefore,  $\hat{C} = \hat{D}$ . Since we know  $D$ , we can compute  $\hat{D}$ . For the example considered earlier,  $\hat{D} = 1101$ . Therefore, we have to set the check bits so that  $\hat{C} = 1101$ . This is done, simply, by assigning the bit string  $\hat{C}$  to the check bits in order from higher to lower positions; for the example, assign 1 to the check bit at positions 8, 4 and 1, and 0 to the check bit at position 2. The reason this rule works is that assigning a value  $v$  to the check bit at position  $2^i$ ,  $i \geq 0$ , in the transmitted string has the effect of setting the  $i^{th}$  bit of  $\hat{C}$  to  $v$ .

How many check bits do we need to transmit a given number of data bits? With  $n$  check bits, we can encode  $2^n$  positions, i.e., 0 through  $2^n - 1$ . Since we have decided not to have a position numbered 0 (see the discussion at the end of the “Receiver” and the beginning of the “Sender” paragraphs), the number of positions is at most  $2^n - 1$ . Of these  $n$  positions are taken up by the check bits, so the number of data bits is no more than  $2^n - n - 1$ .

### 6.3 Reed-Muller Code

You have probably emailed photographs or send faxes. Such transmissions are always digital; text, image, audio, video are all converted first to bit strings and then transmitted. The receiver converts the received string to its original form. For text strings, conversion to and from bit strings is straightforward. For a still image, like a photograph or scanned document, the image is regarded as a matrix: a photograph, for instance may be broken up into 200 rows, each a strip, and each row may again be broken up into columns. It is not unusual to have over a million elements in a matrix for a photograph the size of a page. Each matrix element is called a *pixel* (for picture element). Each pixel is then converted to a bit-string and the entire matrix is transmitted in either row-major or column-major order.

The conversion of a pixel into a bit-string is not entirely straightforward; in fact, that is the subject matter of this section. In the most basic scheme, each pixel in a black and white photograph is regarded as either all black or all white, and coded by a single bit. This representation is acceptable if there are a large number of pixels, i.e., the resolution is fine, so that the eye can not detect minute variations in shade within a pixel. If the resolution is low, say, an image of the size of a page is broken up into a  $80 \times 110$  matrix, each pixel occupies around .01 square inch; the image will appear grainy after being converted at the receiver. Here are two images scanned at different resolutions.

The Mariner 4 spacecraft, in 1965, sent 22 photographs of Mars, each one represented by a  $200 \times 200$  matrix of pixels. Each pixel encoded 64 possible levels of brightness, and was transmitted as a 6 bit string. A single picture, consisting of  $200 \times 200 \times 6$  bits, was transmitted at the rate of slightly over 8 bits per second, thus requiring around 8 hours for transmission. The subsequent Mariners, 6, 7 and 9, did a much better job. Each picture was broken down to  $700 \times 832$  pixels (i.e., 582,400 pixels per picture vs. 40,000 of Mariner 4) and each pixel of 6 bits was encoded by 32 bits, i.e., 26 redundant bits were employed for error detection and correction. The transmission rate was 16,200 bits per second. This takes around 18 minutes of transmission time per picture of much higher quality, compared to the earlier 8 hours.

Our interest in this section is in transmitting a single pixel so that any error in

$$H_1 = \left[ \begin{array}{c|c} 1 & 1 \\ \hline 1 & 0 \end{array} \right]$$

Table 10: Hadamard Matrix  $H_1$

$$H_2 = \left[ \begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{array} \right]$$

Table 11: Hadamard Matrix  $H_2$

transmission can be detected and/or corrected. The emphasis is on correction, because retransmission is not a desirable option in this application. We study the simple Reed-Muller code employed by the later Mariners.

To motivate the discussion let us consider how to encode a pixel which has 8 possible values. We need only 3 bits, but we will encode using 8 bits, so as to permit error correction. As pointed out in section 6.1, error correcting capability depends on the Hamming distance between the codewords. The 8-bit code we employ has distance 4 between *every* pair of codewords; so, we can detect 3 errors and correct 1. Error correction capability is low with 8-bit codewords. The Mariners employed 32-bit codewords, where the inter-word distance is 16; so, 15 errors could be detected and 7 corrected.

The codewords for the 8-bit Reed-Muller code are shown as rows of the matrix in Table 12. The rest of this section is devoted to the construction of  $2^n$  codewords,  $n \geq 1$ , where the Hamming distance between any two codewords is exactly  $2^{n-1}$ .

## Hadamard Matrix

We will define a family of 0, 1 matrices  $H$ , where  $H_n$  is a  $2^n \times 2^n$  matrix,  $n \geq 0$ . In the Reed-Muller code, we take each row of the matrix to be a codeword. We showed  $H_3$  in Table 12.

The family  $H$  is defined recursively.

$$H_0 = \begin{bmatrix} 1 \end{bmatrix}$$

$$H_{n+1} = \begin{bmatrix} H_n & H_n \\ H_n & \overline{H_n} \end{bmatrix}$$

where  $\overline{H_n}$  is the bit-wise complementation of  $H_n$ . Matrices  $H_1$  and  $H_2$  are shown in Tables 10 and 11, and  $H_3$  appears in Table 12.

Hadamard matrices have many pleasing properties. The two that are of interest to us are: (1)  $H_n$  is symmetric for all  $n$ , and (2) the Hamming distance between any two distinct rows in  $H_n$ ,  $n \geq 1$ , is  $2^{n-1}$ . Since the matrices have been defined recursively,

$$H_3 = \left[ \begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array} \right]$$

Table 12: Hadamard Matrix  $H_3$ : 8-bit Simple Reed-Muller code

it is no surprise that the proofs employ induction. I will leave the proof of (1) to you. Let us prove (2), using induction on  $n$ .

For  $n = 1$ : The result can be verified for  $H_1$ , shown in Table 10. The distance between the two rows is  $2^{1-1}$ .

Before we prove the inductive case, we state a simple result. In the following,  $d(x, y)$  stands for the Hamming distance between  $x$  and  $y$ .

**Observation** Let  $p$  and  $q$  be words of even length, say  $2 \times t$ . Then

$$\begin{aligned} d(p, q) = t &\equiv d(\overline{p}, q) = t \\ d(p, q) = t &\equiv d(p, \overline{q}) = t \\ d(p, q) = t &\equiv d(\overline{p}, \overline{q}) = t \end{aligned} \quad \square$$

For  $n + 1, n \geq 1$ : Consider any two distinct rows  $x$  and  $y$  of  $H_{n+1}$ . Let  $x_l$  and  $x_r$  denote the left and right halves of row  $x$  (similarly,  $y_l$  and  $y_r$  are defined); see figure 2. Then,  $d(x, y) = d(x_l, y_l) + d(x_r, y_r)$ . Now, both  $x_l$  and  $y_l$  are rows of  $H_n$ , because the left half of  $H_{n+1}$  has only such rows. But,  $x_r$  and  $y_r$  may be rows of  $H_n, \overline{H_n}$  or both. We consider two cases:

(1)  $x_l$  and  $y_l$  are the same row of  $H_n$ : (See Case 2 in figure 2.) Then  $d(x_l, y_l) = 0$ . Also  $x_r = \overline{y_r}$ . So,  $d(x_r, y_r) = 2^n$ . Hence,  $d(x, y) = 2^n$ . (Note: the inductive hypothesis has not been used in this proof.)

(2)  $x_l$  and  $y_l$  are distinct rows of  $H_n$ : Using the inductive hypothesis,  $d(x_l, y_l) = 2^{n-1}$ . The computation of  $d(x_r, y_r)$  requires a case analysis. Either, (a)  $x_r \in H_n, y_r \in H_n$ , (b)  $x_r \in H_n, y_r \in \overline{H_n}$ , (c)  $x_r \in \overline{H_n}, y_r \in H_n$ , or (d)  $x_r \in \overline{H_n}, y_r \in \overline{H_n}$ . Therefore, either  $x_r$  or  $\overline{x_r}$  is from  $H_n$  and so is  $y_r$  or  $\overline{y_r}$ . In all cases, they correspond to different rows of  $H_n$ . By the inductive hypothesis, the two distinct rows of  $H_n$  they correspond to have distance  $2^{n-1}$ . Using the observation above,  $d(x_r, y_r) = 2^{n-1}$ . So,  $d(x, y) = d(x_l, y_l) + d(x_r, y_r) = 2^{n-1} + 2^{n-1} = 2^n$ .

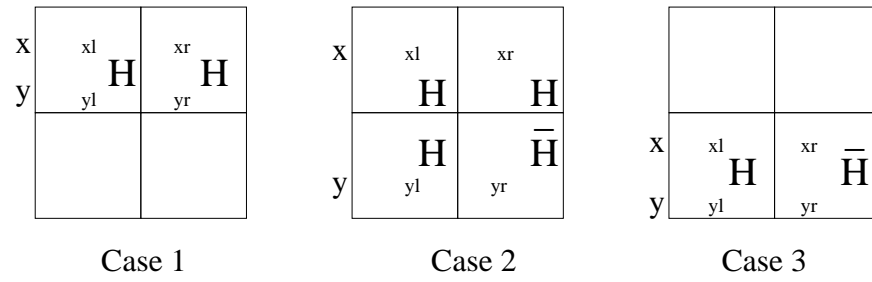


Figure 2: Proof of Distance property

## References

- [1] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.
- [2] R.L. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.