Defect-Tolerant Interconnect to Nanoelectronic Circuits: Internally-Redundant Demultiplexers Based on Error-Correcting Codes

Philip J. Kuekes, Warren Robinett, Gadiel Seroussi, R.Stanley Williams
Quantum Science Research
HP Laboratories Palo Alto
HPL-2004-121
July 13, 2004*

E-mail: stan.williams@hp.com

We describe a family of defect-tolerant demultiplexers based on error-correcting codes. A conventional demultiplexer with a k-bit input address and 2^k-bit output may be fortified against certain defect types by widening its address bus to $n > k$ bits to permit an encoded address to be used within the demultiplexer. The redundant address is computed by an encoder that guarantees a minimum Hamming distance d between addresses, which sparsely populate an expanded address space. The increased Hamming distances between addresses are especially tolerant of stuck-open defects (and broken wires, which are equivalent to multiple stuck-open defects). For each address width k, there are a series of demultiplexer designs with increasing internal redundancy, increasing d, and increasing capability for defect tolerance. These circuit designs are especially suitable for nano-scale crossbars; in particular, they may be realized at the interface where the CMOS wires of conventional microelectronics cross nano-wires to form a mixed-scale interconnect crossbar. Thus, a small number ($2n$) of CMOS wires may be used to control a much larger number (2^k) of nano-wires; the family of encoded demultiplexer designs provides a robust interface to the nano-circuitry, giving significant protection from manufacturing mistakes at the cost of a relatively small amount of area overhead $f_A \cong \frac{n}{k}$. This is a qualitatively new application of error-correcting codes, the analysis of which combines elements of the conventional coding-theoretic notions of full-error and erasure correction. In particular, a code with minimum distance d guarantees tolerance to up to d-1 defects per nano-wire, in analogy to conventional erasure correction.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2004

Abstract

We describe a family of defect-tolerant demultiplexers based on error-correcting codes. A conventional demultiplexer with a k-bit input address and 2^k-bit output may be fortified against certain defect types by widening its address bus to $n > k$ bits to permit an encoded address to be used within the demultiplexer. The redundant address is computed by an encoder that guarantees a minimum Hamming distance d between addresses, which sparsely populate an expanded address space. The increased Hamming distances between addresses are especially tolerant of stuck-open defects (and broken wires, which are equivalent to multiple stuck-open defects). For each address width k, there are a series of demultiplexer designs with increasing internal redundancy, increasing d, and increasing capability for defect tolerance. These circuit designs are especially suitable for nano-scale crossbars; in particular, they may be realized at the interface where the CMOS wires of conventional microelectronics cross nano-wires to form a mixed-scale interconnect crossbar. Thus, a small number ($2n$) of CMOS wires may be used to control a much larger number (2^k) of nano-wires; the family of encoded demultiplexer designs provides a robust interface to the nano-circuitry, giving significant protection from manufacturing mistakes at the cost of a relatively small amount of area overhead $f_A \cong \frac{n}{k}$.

This is a qualitatively new application of error-correcting codes, the analysis of which combines elements of the conventional coding-theoretic notions of full-error and erasure correction. In particular, a code with minimum distance d guarantees tolerance to up to d-1 defects per nano-wire, in analogy to conventional erasure correction.

1. Introduction

Manufacturing perfect electronic circuits is expensive, and the cost is increasing steeply as the requirements for improved precision become more stringent with shrinking feature sizes. The cost of perfection may very well become prohibitive long before the feature sizes in integrated circuits shrink to the point where quantum phenomena dominate their properties. Thus, *defect tolerance* – defined as the ability of a circuit to perform perfectly even with broken components that result from manufacturing mistakes – will be essential in the electronics of the future.

A practical defect-tolerant computing architecture was demonstrated by the Teramac experimental supercomputer, which worked reliably even though 3% of its components were defective [1]. To configure Teramac for a particular computation, a discovery process was first used to identify bad components, which were then avoided at the time a program was compiled onto the hardware. This was possible because of the high-bandwidth, redundant routing network that linked the components within the machine; at the defect rate that actually occurred (3%), working components could be efficiently linked together (through reconfigurable signal paths) to build a perfect machine.

The simplest example of such a high-communications-bandwidth structure is the crossbar, which is illustrated schematically in Fig. 1. Such structures can serve as both memory [2] and configurable logic arrays [3] if the switches can be toggled electrically [4]. Demonstrations of nanoscale crossbars have utilized monolayer molecular films trapped between the upper and lower wires [5,6] that provide the switching function.

A switch can be set to its high-impedance state by putting a relatively large positive voltage across it, and can be set to its low impedance state by putting a large negative voltage across it [7] — we consider this to be configuring the switch. At lower absolute voltages, the switch will "remember" its setting to high or low impedance and simply act as a resistor in the electrical circuit. Thus, the topology of the crossbar, as an electrical circuit, consists of N vertical wires crossing M horizontal wires, with the wires linked at each crosspoint by a voltage-controlled switch.

The Teramac defect-tolerant computing architecture provides guidance in creating defect-tolerant designs for the crossbar structure because of the high-bandwidth, redundant, and configurable data paths present in the crossbar. These are simply the wires and the switches. Thus, it should be possible to repeat the strategy used successfully with Teramac – to first locate bad components through a search, and then route around the bad components when configuring the crossbar to perform a computation. We call this method for defect tolerance the "locate and avoid" strategy.

A practical design for a nano-scale memory system must be defect tolerant, and will probably contain several different types [8] and hierarchies [9] of defect tolerance. We will focus in this paper on one of the main subsystems needed for building a nano-scale crossbar memory – a demultiplexer (demux) [10, 11]. In particular, we will describe how to use error-correcting codes [13] to build an encoder-demultiplexer system that can provide a high degree of defect tolerance to a crossbar memory with only a small amount of redundancy in the circuit. The method described does not use the "locate and avoid" strategy, but can enable the use of the strategy at higher levels of the system by significantly decreasing the effective defect rate "seen" by those levels.

1.1 Some Obstacles

We first survey some of the problems we face: the unreliability of components, limitations on the geometric complexity of structures that can be fabricated, and the difficulty in achieving interconnect between nanocircuitry and conventional microcircuitry. Since conventional, lithographically-fabricated microcircuitry is today primarily implemented with Complementary Metal-Oxide-Semiconductor (CMOS) circuits, we will use the term CMOS to mean conventional circuitry manufactured at whatever scale is available.

The manufacturing yield of nano-components will be worse than that of CMOS components. The defect rate for CMOS components is measured in parts per million (ppm)—for example, in an early 1990's RAM chip designed with defect-tolerant techniques [12], the component defect rate was approximately 22 ppm. In contrast, in nano-circuitry, we may have to deal with defect rates in the range of 1% to 10%, either because that is the best we can do in manufacturing, or because it is too expensive to manufacture with extremely precise tolerances, and we are forced to loosen the tolerances for economic reasons.

Another problem is that, in comparison with CMOS lithography, we are limited in the geometric complexity of the nano-structures we can fabricate, primarily by alignment precision. Crossbar structures have the advantage of being highly regular, and thus fairly simple and inexpensive to fabricate.

A third problem is the difficulty in achieving interconnect between nanocircuitry and CMOS-scale microcircuitry. To be of any use, we must be able to get electrical signals containing data into and out of nano-circuitry. We must find methods of electrically connecting CMOS wires to nano-wires and techniques that allow a small number of CMOS wires to control

many nano-wires – otherwise the input and output busses would be bigger than the nano-structure being accessed.

2. Nano-scale Crossbar Memory and its Interconnect to CMOS

Our solution for overcoming these obstacles is a defect-tolerant demultiplexer that fits efficiently onto the crossbar. In a demux, a relatively small number (k) of input lines controls a much large number (2^k) of output lines. The job of a demux is to turn on exactly one of its output lines (corresponding to the address on the input lines) while keeping all the other output lines turned off.

A demultiplexer is one of the primary subsystems needed to build a nano-scale crossbar memory. In the normal structure of a memory system based on a 2D grid of memory cells, we wish to use some address bits to indicate a single row and a single column to be activated in the memory grid (as in a standard CMOS DRAM memory chip), so that we can read or write the bit stored in the junction where those two wires cross, as shown in Fig. 2.

We have fabricated a nano-scale crossbar memory as an experimental proof-of-principle system [6]. This circuit was built on a 8×8 crossbar structure, which was configured so that a 4×4 subarray functioned as a memory grid, and two other 4×4 subarrays functioned as row and column selectors (demultiplexers). The entire 8×8 crossbar occupied 1 μm^2 of chip area. This small crossbar had so few components that it could be made to work without defect-tolerant techniques. But for realistically-sized memories with large numbers of nanowires, defect tolerance will be essential.

Note that in a nano-scale crossbar memory, the demultiplexers are at the interface to the CMOS circuitry. The demux is therefore a very vulnerable part of a system, since if it is

defective, all the downstream nano-circuitry is lost. Thus, it is especially valuable to improve the operating capability of this crucial component. What we need is an interconnect design that is defect-tolerant, that is compatible with the crossbar structure, and that allows a small number of CMOS wires to control or access a much larger number of nano-wires.

3. A Defect-Tolerant Demultiplexer

Before showing the layout of the demux circuit on the crossbar, we present the analog circuit that implements an AND gate in the crossbar. We focus on one nano-wire in Fig. 3 to illustrate how it implements a 2-bit AND function.

3.1 Layout of Demultiplexer onto Crossbar

Fig. 4 shows how an ordinary (non-defect-tolerant) demux can be laid out on the crossbar. For a k-bit address, there are 2^k possible addresses, and thus, if we have 2^k nano-wires, there is a one-to-one correspondence between addresses and nano-wire output lines. The traditional demux circuit is designed so that (1) each wire has a unique address, and (2) the addresses of the wires are dense – they completely fill the address space.

To keep the diagram simple, we have shown a demux in Fig. 4 with a 2-bit address and 4 output lines, but a similar layout is valid for larger, realistically-sized demux circuits. The circuit layout conforms to the crossbar with no wasted space. The demux activates a single wire, as determined by the input address, and is suitable for use as a row or column recognizer in implementing a nano-scale crossbar memory. However, a major problem is that this design is vulnerable to manufacturing defects.

3.2 The Effect of Defects

In the demux of Fig. 4, a single defective connection on any nanowire will cause that nanowire to give an erroneous output (Fig. 5). The AND gate on each nano-wire may be thought of as a specialized circuit that recognizes only its own address, and thus we call it a *recognizer*. For example, in Fig. 5, wire S11 should respond only to address "11". The defect shown on wire S11 causes the recognizer to become less selective, and it responds to the class of addresses "1X", where "X" is a "don't care" bit. That is, wire S11 now turns on for both address "11" (as it should) and address "10" (as it should not). We therefore give this defective nanowire the label "1X". Since all the address bits are needed to uniquely identify a wire in this densely-populated address space, a defective connection (a stuck-open defect) on any nano-wire will cause that wire to erroneously turn on for some other input address.

This illustrates the failure mechanism when stuck-open defects occur as manufacturing mistakes. Before presenting a technique for dealing with this problem, we note that the types of defects we have encountered so far in fabricating crossbars are stuck-open connections, stuck-closed connections, shorts between adjacent wires and broken wires. The coding technique in this paper applies primarily to stuck-open defects. Methods for handling other defect types will be presented elsewhere. The manufacturing process can be adjusted to make tractable defect types more likely than intractable ones.

3.3 A Defect-Tolerant, Internally-Redundant Demultiplexer Circuit

A design for a demultiplexer (2-bit address, 4 output lines) that can perform correctly in the presence of stuck-open defects is shown in Fig. 6, and may be compared with the normal design shown in Fig. 4b. The basic idea is to add a few more address lines (vertical wires) to the

demux circuit, forming an address with redundant bits for each nano-wire in an expanded address space, so that a single-bit error in an address (caused by a defect) will not cause the defective wire to interfere with a wire having an "adjacent" address (in address space). The "sparsely-populated" address space, in which neighboring addresses are no longer adjacent, allows the circuit to tolerate some defects.

3.4 Distance Between Addresses: Hamming Distance

The *Hamming distance* between addresses is defined as the number of address-bits that are different in two equal-length addresses. For example, the 4-bit addresses "1110" and "0111" are separated by 2 units of Hamming distance – they differ in the first and last bit-positions. (Hamming distance applies only to a pair of bit-strings of the same length, and is undefined between bit-strings of different lengths.) Given two arbitrary k-bit addresses **a** and **a'**, the maximum possible Hamming distance is k, meaning that **a** and **a'** differ in every single bit, and are thus complements of each other (e.g., "1110" and "0001"). The minimum possible Hamming distance is 0, meaning that **a** and **a'** match in every bit-position and are thus equal (e.g., "1110" and "1110"). What we loosely called "adjacent addresses" above could be defined more precisely as two addresses separated by 1 unit of Hamming distance – in other words, addresses that differ in only a single address-bit (e.g., "1110" and "1111").

In a normal demux, such as that of Fig. 4, the address space is densely filled: every possible address occurs – namely {00, 01, 10, 11}. For any particular address (11), flipping one bit yields another valid address (10), at a Hamming distance of 1 from the original address. Thus, in the dense address space of a normal demux circuit, every output line has exactly k nearest neighbors at Hamming distance 1.

The defect-tolerant demux circuit of Fig. 6 is designed so that in the expanded (3-bit) address space, the addresses of the nano-wires are all separated by 2 units of Hamming distance. The addresses of the output lines are {000, 011, 101, 110}. At least 2 bits must be flipped to transform any of these addresses to any other in the set.

3.5 Coding Theory

The challenge in designing demultiplexers that can withstand one or more bad connections is to generate an appropriate set of supplemental address bits that increases the minimum Hamming distance between nano-wire addresses without adding too much additional circuitry. This is a classical problem in the area of information theory known as *coding theory* [13,14], which we will use to systematically add redundant bits and obtain good Hamming distances between addresses. These codes are called *error-correcting codes*, from the application for which they were originally invented, namely, correcting errors in noisy communication channels. We first introduce some terminology from coding theory.

For the set of addresses {000, 011, 101, 110} just discussed, each member of the set is called a *codeword*, and the entire set is considered to be a particular *code*. A *binary code* uses the set {0,1} as its *alphabet*, and thus a *binary vector*, or *binary matrix*, contains only 0's and 1's. So a k-bit address, for example, can be considered to be a length-k binary vector. A binary *code* C, of length n, is a set of binary vectors (*codewords* $\mathbf{u}_i$) of length n. The cardinality, or *size* of C, will be denoted M. The *dimension* of C, denoted by k, is given by

$$k = \log_2 M,$$

while its *minimum distance*, denoted by d, is defined by

$$d = \min_{\mathbf{u}_1, \mathbf{u}_2 \in C,\ \mathbf{u}_1 \neq \mathbf{u}_2} \operatorname{dist}(\mathbf{u}_1, \mathbf{u}_2),$$

where $\mathrm{dist}(\mathbf{u}_1, \mathbf{u}_2)$ denotes the Hamming distance between two binary vectors. We refer to *C* as an (*n*, *M*, *d*) code, or an [*n*, *k*, *d*] code. The latter notation, which emphasizes dimension rather than size, is generally used in the important special case of linear codes.

In a binary *linear code C,* the codewords can be represented by binary *n*-vectors, and the encoding operation can be realized by means of a linear transformation, $\mathbf{u}=\mathbf{a}\cdot G$, where **a** is the input (a binary *k*-vector), **u** is its encoded counterpart (a binary *n*-vector), and *G* is a $k\times n$ binary matrix. *G* is referred to as the *generator matrix* of the code. The dot operation (·) is matrix multiplication defined over the finite field [13] $\mathbf{F}_2$, namely, the set {0, 1} together with Boolean XOR and AND as addition and multiplication operations, respectively. Equivalently, $\mathbf{F}_2$ can be regarded as the set {0,1} with arithmetic operations modulo 2.

A binary linear code *C* is closed under bitwise XOR operations of its codewords. A binary linear code always contains the all-zero codeword **0,** and its number of codewords is always a power of 2, i.e., $M = 2^k$. In fact, assuming that the matrix *G* has full rank *k*, running all possible binary *k*-vectors (*messages*) **a** through the linear transformation $\mathbf{u} = \mathbf{a}\cdot G$ produces the 2^k codewords comprising the code. An *encoder* for a code *C* is an implementation of the linear transformation $\mathbf{u} = \mathbf{a}\cdot G$. The ease of this operation makes linear codes attractive in practice.

In our application, the "messages" are the unencoded addresses of the nano-wires we wish to access (signal vector **a** in Fig. 6). The codewords (signal vector **u** in Fig. 6) are the encoded addresses for the nano-wires, which are redundant representations of their payload messages. With linear codes, it is usual to employ a *systematic* encoding, where **u** contains **a** as a prefix, with *s* supplemental *check-bits* appended. The generator matrix for such systematic encoding assumes the form $G = [\, I \mid A \,]$, where *I* is a $k\times k$ identity matrix, and *A* is a $k\times s$ binary matrix defining the check bits. Thus, $n = k + s$, and the matrix *G* has full rank *k*. Notice that in a

systematic encoding, only the check bits require computation circuitry; the "message" bits of **u** are passed to the encoded address untouched. The size of the encoding circuitry is proportional to the number of non-zero entries in A.

The $[n,k,d]$ nomenclature for a linear binary code summarizes the most important parameters of a code:

- n bits in each codeword (binary vector **u** in Fig. 6);
- k bits in the bitstrings to be encoded (binary vector **a** in Fig. 6);
- d units of Hamming distance as the minimum distance between different codewords.

3.6 Example Codes

Some examples will help to clarify these mathematical definitions and properties. In the small example we have been considering, the code $C = \{000, 011, 101, 110\}$ has size $M = 4$, dimension k=2, length n=3, and minimum distance d=2. Thus, using the $[n,k,d]$ terminology, it is a [3,2,2] code. Its encoding function is shown in Fig. 7, along with its systematic generator matrix $G = [\ I\ \mid A\ ]$, with $A = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. The encoder circuit of Fig. 8b precisely implements the computation $\mathbf{u} = \mathbf{a} \cdot G$, with the generator matrix G "hard-wired" into the circuit as the interconnect pattern, and the XOR gate performing the required summation to compute the last component of the output vector.

3.7 Densely-Populated Versus Sparsely-Populated Address Spaces

Fig. 9 shows visualizations for how the nano-wire addresses fill their address spaces for the two demux designs we have considered so far – the normal, non-defect-tolerant demux of Fig. 4, and the defect-tolerant demux of Fig. 6. Thus the code of Fig. 9a is a [2,2,1] code (2-bit

output, 2-bit input, 1 unit of separation between codewords), while the code of Fig. 9b is a [3,2,2] code (3-bit output, 2-bit input, 2 units of separation between codewords).

3.8 Finding Codes

It can be difficult to construct good codes; but once a code is known, it can be used simply by knowing its generator matrix. The systematic generator matrix immediately yields a digital circuit implementing the encoding function for the code. Thus, we can capitalize on over 50 years of mathematical research in coding theory simply by looking up codes that have already been discovered. For the applications in which we are interested (demultiplexers used to build crossbar memories), the number of nanowires addressed in one demultiplexer block will be fewer than ten thousand in the foreseeable future, which means that k is bounded (say, $k \leq 14$). In the context of coding theory, codes of this size are considered relatively short. Tables of the best known short codes can be found in the literature [13, Appendix A] or on the web [15]. For a given set of parameters, the "mathematically" best code (e.g., largest minimum Hamming distance given the code length and size) might be non-linear and relatively complex to encode. We prefer linear codes for the simplicity of their encoding circuitry.

The parity codes, which have d=2, are the simplest. A parity code adds a single redundant bit to an input bitstring, computed as the XOR of all the input bits. Parity codes have been traditionally used in magnetic tapes and other memory systems to detect single-bit errors. The encoder of Fig. 8 uses a parity code.

The family of *Hamming codes* and their variants provides a range of useful codes, with minimum distance d=3 or d=4. Hamming codes are, in a sense, the simplest non-trivial codes and a full treatment of their properties can be found in almost every coding theory book (cf.

[13]). For every code *order m*, there is a Hamming code with parameters [2^m-1, 2^m-1-m, 3]. Appending an overall parity check to each codeword, this code yields an "extended" Hamming code with parameters [2^m, 2^m-1-m, 4]. These codes, in turn, can be shortened to obtain families of codes with parameters [2^m-1-r, 2^m-1-m-r, 3], and [2^m-r, 2^m-1-m-r, 4], respectively, for integers r, $0 \le r < 2^{m-1}$. For m=4, and r=4, for example, there is a [11,7,3] shortened Hamming code and a [12,7,4] shortened extended Hamming code. For simplicity, we will omit the qualifier "shortened" in the discussion to follow, letting it be assumed from the context.

3.9 Operation of the Defect-Tolerant Demultiplexer: How It Tolerates Defects

An important area of design latitude in a circuit that includes (unreliable) nano-circuitry on the same substrate with (reliable) CMOS circuits is that we can decide which parts of the circuit we wish to protect from defects simply by choosing those that are implemented in CMOS. Fig. 10 shows that for the defect-tolerant demux, we have chosen to implement the encoder and code balancer (NOT gates) in CMOS, and only the AND operations (recognizer array) at the nano-level. This choice still leaves the bulk of the circuitry at the nano level, since there are 2^k n-input AND gates, but only n NOT gates and n-k XOR gates, of k or less inputs each. Thus, only the AND gates (and their associated interconnect) in the circuit are required to tolerate defects. Downstream of the reliable CMOS, the first place that defects can occur is in the recognizer array, where the connections between the larger CMOS wires and the narrow nano-wires may be defective.

We can now describe how the defect-tolerant demux of Fig. 10 is able to tolerate defects. Recall that the effect of a stuck-open defect on a particular nano-wire, as illustrated in Fig. 5, is to "disconnect" one of the inputs to the AND gate circuit on that wire. The result is that the

nano-wire becomes less selective, and responds, in the example of Fig. 5, to address "1X", and thus responds to both addresses "10" and "11".

Note that an address label with a single "X" in it will respond to an address at most 1 unit of Hamming distance away from its correct address. In general, a nano-wire with e stuck-open defects on that same wire will have a label containing e X's, and will be activated by 2^e different addresses, the most distant of which will be at Hamming distance e from the wire's correct address. The addresses of the nano-wires in the defect-tolerant demux of Fig. 10 are the codewords {000, 011, 101, 110}. A single stuck-open defect on a nano-wire will introduce a single "X" into the wire's label, which will not be damaging enough to get it to interfere with another nano-wire, since the nearest neighbors are 2 units of Hamming distance away. Thus, the circuit of Fig. 10 can tolerate a single stuck-open defect on each nano-wire. If two defects occur on the same wire, then it will interfere with a neighboring wire. This defect-tolerant demux is certainly better than the ordinary demux of Fig. 4, which cannot tolerate any defects at all; but it also has its limits, being unable to tolerate two or more defects on a single nano-wire.

3.10 Family of Defect-Tolerant Demultiplexer Designs

The defect-tolerant demux of Fig. 10 is actually a prototype for a family of circuit designs. Any $[n,k,d]$ code can be used to construct a demux with these properties:

- it has a k-bit input address;
- it uses n-bit encoded addresses, and has $2n$ vertical wires (the **v** signal vector of Fig. 6) going from the CMOS circuitry to the nano-level;
- it has 2^k nano-wire output lines, each implementing an n-input AND gate;
- the addresses of the nano-wires are the codewords of the code;

- the minimum distance between codewords (and thus nano-wire addresses) is d;
- the demux can tolerate up to d-1 stuck-open defects on each nano-wire;
- if d or more defects occur on the same nano-wire, an error will occur (the wire will interfere with some other nano-wire for some input address).

These properties may be summarized by saying that for any $[n,k,d]$ code, we can build a corresponding defect-tolerant demultiplexer that can tolerate up to d-1 stuck-open defects on each nano-wire. When we need a demux of a particular size (k) and degree of defect-tolerance (d), we may search the mathematical archives to find a code that is efficient (low n) and therefore requires as few redundant address lines as possible.

3.11 Four Example Demultiplexer Designs, of Increasing Defect-Tolerance

To explore the behavior of this family of demux designs, considered over a range of plausible defect rates, we will now develop four example circuits, based on codes with minimum Hamming distances d=2, d=3, and d=4; and an uncoded (d=1) demux design for comparison. We will use more realistically-sized (k=7) demuxes in these examples. All the demuxes will therefore have 7-bit input addresses and 128 nano-wire output lines. They will differ in the degree of internal redundancy that is added to the circuit, with more (judiciously chosen) redundancy conferring more defect-tolerance.

The four example demultiplexers are based on these codes:

- Example D1: [7,7,1] code. A normal, non-defect-tolerant demux (d=1) against which the other examples may be compared.
- Example D2: [8,7,2] code. This demux is based on a simple parity code (d=2).
- Example D3: [11,7,3] code. This demux is based on a Hamming code (d=3).

- Example D4: [12,7,4] code. This demux is based on an extended Hamming code (d=4).

The respective generator matrices are shown in Fig. 11 and Fig. 12a.

Together with the explanations of Fig. 8, the demultiplexer of Fig. 6 may be used as a prototype to show how to use the generator matrix for an arbitrary [n,k,d] code to construct a defect-tolerant demux circuit. For realistically-sized cases, there will be more bits in each of the signal vectors:

- k bits in the input **a** to the encoder;
- n bits in the encoded address **u**;
- $2n$ vertical wires (signal vector **v**) linking the CMOS circuitry to the nano-wires;
- 2^k nano-wire output lines wires (signal vector **s**).

For example, the generator matrix for the [12,7,4] code (Fig. 12a) gives rise to a demux (Example D4) in which the lengths of the signal vectors are 7 bits (**a**), 12 bits (**u**), 24 bits (**v**), and 128 bits (**s**). The encoder circuit for this code is shown in Fig. 12b.

4. Evaluating the Four Demultiplexer Designs

Bearing in mind that these demuxes are intended to be used as subsystems (row and column selectors) within nano-scale crossbar memories, we evaluate their quality by asking the question: How many of the nano-wire output lines can be used to select a row or column in a memory grid? A nanowire (with address **u**) can fail to be usable for either of two reasons:

- The nanowire has too many defective connections, and is erroneously selected when another nanowire is addressed. We call wire **u** a *villain* in this case.

- Some other nanowire has too many defective connections, and is selected when **u** is addressed. We call this **u** a *victim*, because it may be completely defect-free – but its address conflicts with a wire that is defective (a villain).

We assume that both nanowires are disabled when a conflict occurs. Stuck-open defects make the nano-wire respond to an overly-broad set of addresses. When enough defects occur on the same nano-wire, it will interfere with some other nano-wire, thus becoming a villain (and causing the other wire to become a victim). For a demux based on a particular code *C*, we can compute the (normalized) expected number of usable lines $E_{usable}(p)$, as a function of the defect probability *p*. An exact formula for this expectation is derived in Appendix A. It is based on the combinatorial structure of the code *C*, and takes the form of a polynomial function of p.

The probability analysis leading to the formulas in Appendix A assumes an idealized model where defects occur independently of each other, and with a uniform probability. Although this model is a reasonable first-order approximation, it is clear that "real life" implementations will not necessarily obey it precisely, and the yield and probability calculations will require some combination of physical measurements, modeling, simulation, and mathematical derivations. With this need in mind, a circuit simulator was developed, with defects generated pseudo-randomly with probability *p*, followed by evaluation of the circuit's performance, and counting up the usable lines (Appendix B). In the case of the idealized probability model, the simulator could be checked against the analytic formulas of Appendix A, and verified to produce accurate results to within expected statistical deviations. The simulator is described in Appendix B.

We can now use the graphs of the function $E_{usable}(p)$ to evaluate each of the four demux circuits. The four plots in Fig. 13 show how much each demux degrades (by having fewer and

fewer usable lines) as the defect rate *p* increases. We see that, for the case of the normal demux, the expected number of usable nanowires decreases rapidly as the defect rate *p* increases. A single supplemental bit provided by the [8,7,2] parity code creates substantial defect tolerance by allowing each nano-wire to tolerate just one defect. We see a further improvement by utilizing an [11,7,3] Hamming code, which requires an additional three address bits to be able to tolerate two bad connections per nano-wire. The [12,7,4] extended Hamming code, which can tolerate up to three defects per nano-wire, represents a further significant improvement.

It is clear from the plots of Fig. 13 that codes of increasing *d* yield increasing defect-tolerance. If the defect rate is known to have some fixed value, then as *d* increases, the fraction of usable lines improves. To express it another way, if some other part of the system required that at least, say, 95% of the lines were usable (perhaps a requirement of a higher-level defect-tolerance mechanism), then codes of increasing *d* can tolerate greater component defects rates while still satisfying the requirement. The probability calculations of Table 1 show that having multiple defects on the same nano-wire is not unlikely, even for a case in which there are only 8 CMOS connections to that nano-wire. At a *p*=10% defect rate, for example, 0.5% of the output lines will have defects at 4 or more of their 8 connections.

4.1 Cost-Benefit Analysis

To properly evaluate this family of defect-tolerant demultiplexer designs, we need to consider not only the benefits, but also the cost of the defect-tolerant circuitry introduced into these demultiplexers. Some principal benefits are:

- More usable resources (demux output lines). In the crossbar memory application, more usable output lines (in the row and column recognizers) lead directly to more usable bits in the memory system.
- Ability to tolerate higher defect rates. This can lead to greatly improved manufacturing economics. The expense of the fabrication process will decrease significantly with increasing p that can be tolerated in a circuit, and thus we want to design a system that can tolerate as large a defect rate as possible.

But these benefits incur certain costs:

- More wires are required in the address bus linking the CMOS circuits and nano-circuit.
- Some kind of higher-level defect-tolerant system is required to identify and keep track of the addresses of the unusable nano-wires.

Thus, the benefits must be judged against the costs, and furthermore, the magnitude of both the benefits and costs is strongly dependent on the underlying defect rate p. Our approach here, offered as an example of how to assess such a situation, will be to focus on one benefit (more usable demux output lines, leading to more bits in the crossbar memory) and one cost (chip area consumed due to wider address busses). Combining these two will give us a numerical measure of goodness (usable bits per chip unit area) that will allow us to judge which of the four example demux designs is best, for any given defect rate p. A more comprehensive analysis would include the area consumed by the higher-level defect-tolerant system in the cost calculation, and the relation between manufacturing tolerance (as reflected by the value of p) and actual manufacturing costs. However, these parameters are not known at present.

4.2 Evaluation Function: Bits Per Unit Area

One way to view the economic consequences of appending supplemental bits to the nano-wire addresses is to examine the effect of the supplemental bits on a "yield" quantity, defined as the number of addressable bits per unit of chip area. In the early stages of nano-electronics, the chip area, and thus the cost of a chip, will still be dominated by the CMOS components. Thus, we can ask how much area will be sacrificed by widening the address bus with supplemental address bits. We will consider the specific case of a large crossbar memory with an architecture as illustrated in Fig. 14, constructed of several 128×128 nano-scale crossbar blocks, each containing 2^{14} cross-point junctions. Each crossbar block has two arrays of 128 nanowires crossed over each other, with each array of nanowires addressed by a demultiplexer.

In this example, only ~10% of the circuit area contains nanocircuitry – the other ~90% is taken up by CMOS circuitry and recognizer arrays, in which CMOS wires cross nano-wires [17]. Each encoder can drive many recognizer arrays simultaneously, with specific recognizers activated by a signal generated from a "master" encoder. The great advantage of this scheme is that a relatively small amount of CMOS can be used to control a large number of nanocircuit elements. Thus, even if the blocks of nanocircuitry only cover 10% of the total circuit area, the total functionality of a hybrid electronic circuit can be much larger than that constructed from the conventional circuitry alone.

To calculate areas, we will start with realistic values for the pitch of nano-wires (λ_N = 30 nm) and the pitch of CMOS wires (λ_C = 130 nm), based on plans for DRAM production in 2007 [18]. We can then calculate the width of the nano-scale crossbar circuit, composed of 2^k nano-wires ($w_N = 2^k \lambda_N$), and the width of the address bus, composed of $2n$ CMOS wires ($w_A = 2n\lambda_C$).

The defect-tolerant demux has three subsystems (Fig. 6c): a CMOS encoder (*n*-*k* XOR gates), a CMOS balancer (*n* NOT gates), and a mixed-length-scale recognizer array (2^k AND gates). For reasonable values of *n* and *k*, most of the demux circuitry is in the recognizer array. The balancer is so small in area that it is negligible. In our application, the encoder is shared by many recognizer arrays, so that its effective area is small. Thus, the areas of the encoder and recognizer array are, respectively,

$$A_E(k,n) \cong 0 \qquad \text{(encoder area amortized across many recognizer arrays)}$$

$$A_D(k,n) = w_N \times w_A = 2^k \lambda_N \times 2n\lambda_C$$

We may therefore calculate the redundancy of the circuit as an area overhead factor f_A for the defect-tolerant demux (with respect to an unencoded $[k,k,1]$ demux, which has no encoder) as

$$f_A = \frac{\textit{area of } [n,k,d]\, recognizer_array + \text{area of } [n,k,d]\, encoder}{\text{area of } [k,k,1]\, recognizer_array} = \frac{A_D(k,n) + A_E(k,n)}{A_D(k,k)}$$

This yields $f_A \cong \frac{n}{k}$, which expresses the fact that when the encoder's area is negligible, the area of the recognizer array is controlled by the address bus width: $2n$ wires (coded) versus $2k$ wires (uncoded).

We can calculate the area of one of the crossbar blocks of Fig. 14 (including the two associated recognizers) by putting the crossbar width and the address bus width together and squaring:

$$A_B(k,n) = (w_A + w_N)^2 = \left(2^k \lambda_N + 2n\lambda_C\right)^2$$

To assess the additional area consumed by widening the address busses, we note that *k*=7 for all four of the codes, whereas n takes on the values 7, 8, 11, and 12. Taking the *n*=7 (uncoded) demux as a baseline, we find that the area penalty is a factor of 1.09, 1.39 and 1.51 for the codes with n = 8, 11 and 12, respectively. Defining a unit of area as that needed to address one bit in a

defect-free uncoded system, the normalized expected number of addressable bits per unit of area in a crossbar memory using code-based demux circuits is:

$$B(p) = \frac{A_B(k,k)}{A_B(k,n)}\left(E_{usable}(p)\right)^2,$$

where $E_{usable}(p)$ is the (normalized) expected number of usable lines, in a defect-tolerant demux based on an [n,k,d] code. An exact expression for the function $E_{usable}(p)$, given the underlying code structure, is derived in Appendix A. Using the derived expression, we plot the function $B(p)$, for each of the four codes under consideration, in Figure 15.

For the case where there are no supplemental bits, the number of usable bits per unit area decreases rapidly with defect rate. Appending a single parity bit ([8,7,2] code) to supplement the address of the nanowires improves the number of usable bits substantially for any defect rate $p \geq$ 1%. The [11,7,3] modified Hamming code imposes a significant penalty on the density of addressable bits if the defect rate is < 5%, but it enables a reasonably high density of bits for a defect probability in the range of 5 to 12%. The [12,7,4] modified Hamming code provides better defect tolerance for defect probabilities greater than 12%. Although we have shown only four examples (based on codes with d=1,2,3 and 4) in this paper, there are more powerful known codes (with d>4) that could be used to define demuxes with even higher degrees of defect-tolerance, if the defect rate was high enough to justify the wider address bus, and other costs.

If we know the defect rate of our manufacturing process (perhaps we have driven it as low as we are able), then we can use Fig. 15 to tell us which demux design is optimal. For example, if p=10%, then the demux of example D3 (based on the [11,7,3] code) is the best choice – as shown by the [11,7,3] curve being above the others at p=10%. The demux designs based on the [8,7,2] or [7,7,1] codes would be insufficiently powerful to cope with that defect

rate, and would deliver fewer bits per unit area. On the other hand, the demux design based on a [12,7,4] code would be more powerful than needed, and its overhead would make it less efficient than the d=3 demux design.

The plots of Figure 15 can be interpreted as showing a *coding gain* in the defect-tolerant demultiplexers, analogous to the notion used in coding for communications [14]. For a given acceptable density of usable memory per unit area (yield), a coded system can tolerate a higher defect rate. For example, one observes in Figure 15 that a 50% yield requires manufacturing to a defect rate of 2% with an uncoded system, whereas the same yield can be obtained with a defect rate of 21% when the [12,7,4] code is used. This gap in defect tolerance can translate to significant economic gains, or even feasibility, since low defect rates might be very difficult or even impossible to attain.

4.3 Higher-Level Defect-Tolerant System

The basis of comparison we have used for the four k=7 defect-tolerant demux designs has been to ask how many of the 128 output lines we can expect to be usable. We have assumed that it is acceptable to have some bad lines, since the coding scheme cannot compensate for all defects in the demux. This implies that the overall system contains a higher-level defect-tolerant mechanism [17] to deal with the bad lines in the demux; this is the only way the overall system can achieve error-free performance.

Note that each bad output line in the demux will have a fixed address. This makes it possible to use the “locate and avoid” strategy – first locate the bad output lines in a demux, store their addresses in some kind of memory, and provide a mechanism for avoiding them thereafter.

It is also possible to use other defect-tolerance strategies, such as error-correcting codes, to deal with the bad lines in a demux.

5. Conclusions

A defect-tolerant demultiplexer allows a circuit designer to build an interface to a significant number of nano-wires, from a much smaller set of CMOS wires, by assigning addresses to the nano-wires. By adding redundant bits to the address, each wire can be uniquely addressed, even in the presence of manufacturing defects in the connections between the nano-wires and the CMOS driving circuits. We have described a systematic method for generating a set of supplemental address bits and designing the circuitry required to implement the new addressing scheme. This generates a family of demultiplexer designs, each based on an $[n,k,d]$ error-correcting code, such that increasingly powerful codes (higher d) yield demultiplexer circuits that can tolerate higher defect rates p. In comparison with normal, uncoded demultiplexers, these defect-tolerant demultiplexer circuits have a redundancy (area penalty) factor of approximately $f_A \cong \frac{n}{k}$.

To design an efficient and reliable demultiplexer, it is necessary to know approximately what the component defect rate is. If the defect rate is significantly greater than planned, the system reliability can plummet drastically; while if the defect rate is significantly over-estimated, unnecessary circuitry will reduce the efficiency of the design. In the coded demultiplexer design presented here, a Hamming distance d guarantees tolerance to $e=d-1$ defects per nano-wire (Fig. 16). This is akin to erasure correction in coding theory, i.e., a situation where the locations of the errors are known, and the remaining task of the error correcting decoder is to reconstruct the

transmitted values in these location [13]. In *full error correction*, on the other hand, the locations of the errors are unknown, and an $[n,k,d]$ code can only guarantee correction of $e = floor\left(\frac{d-1}{2}\right)$ errors in this case (Fig. 16). Interestingly, the probability analysis for our demultiplexer design, presented in Appendix A, does not correspond to either conventional case in coding theory, but contains a combination of elements of both cases.

As feature sizes shrink, it becomes increasingly more expensive to avoid mistakes in manufacturing. At some stage, only circuits designed to tolerate manufacturing defects will be practical to build. This design for an internally-redundant demultiplexer circuit has shown how a standard digital building block (a demultiplexer) can be re-designed to tolerate manufacturing defects.

7. Appendix A.

Exact Calculation of the Expected Value of the Number of Usable Output Lines

We will now derive an exact expression, based on the structure of the code, of the probability p_{usable} of a nano-wire output line of the demux being usable. This probability takes the form of a polynomial function of the stuck-open defect rate p.

Let **u** be a codeword of C, which is therefore the address of a nanowire. The nanowire can fail to be usable for either of two reasons:

- The nanowire has too many defective connections, and is erroneously selected when another nanowire is addressed. The probability of this event is denoted $p_{villain}$.
- Some other nanowire has too many defective connections, and is selected when **u** is addressed. The probability of this event is denoted p_{victim}.

We assume that both nanowires are disabled when a conflict occurs, and the overall probability of a nanowire being usable is the probability that the wire is neither a villain nor a victim. Thus, we can write

$$p_{usable} \quad = \quad (1 - p_{villain}) \cdot (1 - p_{victim}),$$

since defects are assumed to be statistically independent.

We compute the probability p_{victim} first. Let **y** be an address (codeword) different from **u**. Address **y** disables address **u** if the bits of **y** in its *non-defective* positions coincide with the bits of **u** in those positions, or equivalently, **y** is defective in all positions where its bits differ from those of **u** and it can be either defective or non-defective in other positions . The probability of this event is $p^{\mathrm{dist}(\mathbf{u},\mathbf{y})}$. Since we assume that defects are statistically independent, the overall probability that **u** is *not* disabled by any other address is given by

$$1 - p_{victim} \quad = \quad \prod_{\mathbf{y} \in C \setminus \{\mathbf{u}\}} (1 - p^{dist(\mathbf{u},\mathbf{y})}) \quad = \quad \prod_{i=1}^{n} (1 - p^{i})^{W_C(i)}, \qquad 2.$$

where $W_C(i)$ denotes the number of codewords of Hamming weight i in C (since C is linear, the number of codewords at distance i from **u** is independent of **u**; the weight corresponds to taking **u**=**0**). When C is the identity code, we have $W_C(i) = \binom{n}{i}$, a binomial coefficient. We observe that the probability of a nanowire being disabled depends on the *entire distance profile* of the code, and not just the minimum distance, although the latter is likely to be dominant when p is small.

We now compute the probability $p_{villain}$ of a nanowire being a villain. As before, since the code is linear, we can assume that the nanowire corresponds to the zero codeword. Let **e** be the characteristic vector of an error pattern, i.e., e_i=1 if there is a disconnect in position *i,* or 0 otherwise. The set of locations where e_i=1 is called the *support* of **e**. We say that **e** *dominates* an n-vector **u** if the support of **e** includes the support of **u**. An error pattern **e** on address **0** disables

address **u** if and only if **e** dominates **u**. Contrary to the previous case, however, events are not independent, and an error pattern **e** can disable more than one nonzero address. Let $D(C)$ denote the set

$$D(C) = \{ \mathbf{e} \mid \mathbf{e} \text{ dominates } \mathbf{u} \text{ for some } \mathbf{u} \in C \}.$$

Then, the probability that **e** acting on **0** disables some other address is equal to $\mathrm{Prob}(\mathbf{e} \in D(C))$, or, equivalently, the probability that **0** is disabled because it disables some other address is given by

$$p_{villain} = \mathrm{Prob}(\mathbf{e} \in D(C)) = \sum_{i=1}^{n} W_{D(C)}(i) p^i (1-p)^{n-i}. \qquad 3.$$

The complement of this expression for $p_{villain}$ occurs in coding theory also as the probability of success of a code C in correcting an erasure event in an erasure channel with probability p, when the events are taken over blocks of length n.

The weight profiles of our four example codes, and those of the corresponding sets of dominating vectors, can be obtained by explicit enumeration, since the codes are small. The profiles are shown in Table 2. Therefore, we can compute $p_{villain}$ and p_{victim} precisely for these codes.

Clearly, the normalized expected number of usable wires in a demux is

$$E_{usable}(p) = p_{usable}(p)$$

The expression derived for $p_{usable}(p)$ assumes full knowledge of the profiles W_C and W_D for the code. For longer codes for which these profiles might be difficult to characterize completely, partial profiles could be used, which would yield fairly accurate approximations of $p_{usable}(p)$.

8. Appendix B. The Digital Circuit Simulator

A simulator was created to synthesize digital circuits for the family of defect-tolerant demultiplexers described in this paper, to randomly generate simulated defects, to simulate the operation of the demux circuits, and to evaluate their performance. The simulator also computes the exact function $p_{usable}(p)$ described in Appendix A, and compares the results for the simulated sample to the exact expectation. The results agree within expected statistical variations (Fig. 17).

Some goals of the simulator are to:

- analyze demuxes based on any arbitrary binary linear error-correcting code, defined by a generator matrix G;
- evaluate error-correcting codes by calculating their distance profile and other code properties;
- explicitly simulate, at the digital level, the operation of the circuits of the demux in the presence of randomly generated defects (digital circuit model) and assess the performance of the simulated demux with appropriate measures;
- simulate circuits for which the errors are correlated or otherwise violate the assumptions of the model originally posed for the exact calculation of Appendix A.

The simulator is written in the J programming language [19], a dialect of the language APL. J is an interpreted, mathematically-oriented, vector language that is well-suited for this task.

The simulator accepts as inputs a generator matrix G defining a specific binary linear code and a defect probability p. The simulator uses the input G to construct the connection network for the digital encoder, as described in Figure 12. The recognizer array is represented by the ideal connection matrix Ci, which represents the interconnect pattern of the AND gates that compute the output address. The Ci matrix is synthesized as a list of all 2^k n-bit codewords

in the code, with the complement of each codeword appended (by the balancer), resulting in a $2^k \times 2n$ matrix.

The simulator then calculates a $2^k \times 2n$ defect matrix E (a binary matrix in which each entry has a one, meaning defect with probability p, or else a zero, meaning no defect) using a random number generator and the input defect rate p, The Ci and E matrices are combined bitwise to define the recognizer with defective connections. All possible 2^k addresses are run through this circuit, and the outputs are tabulated. From this complete set of inputs and outputs, we can detect the "villains" and "victims" described earlier, and thus calculate, for this particular defect matrix, how many bad output lines there are. This procedure is applied Q times for each of a sequence of values of the defect rate p. Figure 17 shows the results obtained for the [7,7,1] and [12,7,4] codes with Q = 100.

9. Acknowledgements

We gratefully acknowledge R. Roth, G. Snider and J. Straznicky for valuable discussions, and the Defense Advanced Research Projects Agency of the United States for partial support.

10. References

[1] Heath J R, Kuekes P J, Snider G S and Williams R S 1998 A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology *Science* **280** 1716.

[2] Kuekes P J, Williams R S, and Heath J R 2000 Molecular wire crossbar memory *US Patent* #6,128,214

[3] Collier C P, Wong E W, Belohradsky M, Raymo F M, Stoddart J F, Kuekes P J, Williams R S, and Heath J R 1999 Electronically Configurable Molecular-Based Logic Gates *Science* **285** 391-4

[4] Heath J R, Kuekes P J, and Williams R S 2002 Chemically Synthesized and Assembled Electronic Devices *US Patent* #6,459,095

[5] Luo Y, Collier C P, Jeppesen J O, Nielsen K A, Delonno E, Ho G, Perkins J, Tseng H-R, Yamamoto T, Stoddart J F, and Heath J R 2002 Two-Dimensional Molecular Electronic Circuits *Chem Phys Chem* **3** 519-25

[6] Chen Y, Jung G-Y, Ohlberg D A A, Li X, Stewart D R, Jeppesen J O, Nielsen K A, Stoddart J F and Williams R S 2003 Nanoscale Molecular-Switch Crossbar Circuits *Nanotechnology* **14** 462-8

[7] Chen Y, Ohlberg D A A, Li X, Stewart D R, Williams R S, Jeppesen J O, Nielsen K A, Stoddart J F, Olynick D L and Anderson E 2003 Nanoscale molecular switch devices fabricated by imprint lithography *Appl Phys Lett* **82** 1610-12

[8] Nikolic K, Sadek A, and Forshaw M 2002 Fault-tolerant Techniques for NanoNanocomputers *Nanotechnology* **13** 357-362 ; Sadek A, Nikolic K, and Forshaw M 2004 Parallel Information and Computation with Restitution for Noise-Tolerant Nanoscale Logic Networks *Nanotechnology* **15** 192-210

[9] Han J and Jonker P 2003 A defect- and fault-tolerant architecture for nanocomputers *Nanotechnology* **14** 224-30

[10] Zhong Z, Wang D, Cui Y, Bockrath M W and Lieber C M 2003 Nanowire Crossbar Arrays as Address Decoders for Integrated Nanosystems *Science* **302** 1377-79

[11] DeHon A 2003 Array-Based Architecture for FET-Based, Nanoscale Electronics *IEEE Transactions on Nanotechnology* **2** 23-32

[12] Stapper C H, Fifield J A, Kalter H L, and Klassen W A 1993 High-Reliability Fault-Tolerant 16-MBit Memory Chip *IEEE Transactions on Reliability* **42** 596-603

[13] MacWilliams F J and Sloane N J A 1990 *The Theory of Error-Correcting Codes* (North- Holland, New York)

[14] Wicker S B 1995 *Error Control Systems for Digital Communication and Storage* (Prentice Hall, Upper Saddle River)

[15] Jaffe D 2004 Information about binary linear codes http://www.math.unl.edu/~djaffe/codes/webcodes/codeform.html

[16] Dugan J B, Bavuso S J, and Boyd M A 1992 Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems *IEEE Transactions on Reliability* **41** 363-77

[17] DeHon A, Goldstein S C, Kuekes P and Lincoln P 2004 Non-Photolithographic Nanoscale Memory Density Prospects (in preparation)

[18] International Technology Roadmap for Semiconductors 2003 Edition http://public.itrs.net.

[19] Iverson K E 2003 J programming language *J Software Inc.* http://www.jsoftware.com.

11. Competing Financial Interests

The authors declare that they have no competing financial interests.

Tables

Table 1. Probabilities of various numbers of defects on a single nanowire with eight connections for defect rate p=10%.

# defects e per nano-wire	Formula for exactly e defects	Probability of exactly e defects (for p=10%)	Cumulative probability of e or more defects (for p=10%)
8	p^8	0.000001%	0.000001%
7	$8(1-p)p^7$	0.000072%	0.000073%
6	$28(1-p)^2p^6$	0.0022%	0.0023%
5	$56(1-p)^3p^5$	0.041%	0.043%
4	$70(1-p)^4p^4$	0.46%	0.50%
3	$56(1-p)^5p^3$	3.3%	3.8%
2	$28(1-p)^6p^2$	15%	19%
1	$8(1-p)^7p$	38%	57%
0	$(1-p)^8$	43%	100%

Note that the formulas in the second column of the table are the terms in the binomial expansion of $(p + (1-p))^8$, and thus sum to 1. The probability of a nanowire having e or more defects, out of n connections, is given by the formula

$$F_n^e(p) = \sum_{i=e}^{n} \binom{n}{i} (1-p)^{n-i} p^i .$$

Table 2. Weight profiles for codes and dominating sets.

	code [7,7,1]		code [8,7,2]		code [11,7,3]		code [12,7,4]	
i	$W_C(i)$	$W_{D(C)}(i)$	$W_C(i)$	$W_{D(C)}(i)$	$W_C(i)$	$W_{D(C)}(i)$	$W_C(i)$	$W_{D(C)}(i)$
0	1	1	1	1	1	1	1	1
1	7	7	0	0	0	0	0	0
2	21	21	28	28	0	0	0	0
3	35	35	0	56	13	13	0	0
4	35	35	70	70	26	130	39	39
5	21	21	0	56	24	462	0	312
6	7	7	28	28	24	462	48	924
7	1	1	0	8	26	330	0	792
8	0	0	1	1	13	165	39	495
9	0	0	0	0	0	55	0	220
10	0	0	0	0	0	11	0	66
11	0	0	0	0	1	1	0	12
12	0	0	0	0	0	0	1	1

Figure Captions

Fig. 1. Circuit topology of a crossbar of switches, each of which connects one horizontal wire to one vertical wire.

Fig. 2. Schematic layout diagram of a crossbar memory, showing the role of the row selector and column selector subsystems. (The selectors are demultiplexers.)

Fig. 3. Diode-logic circuit with pull-up resistor that implements an AND gate. Blue symbolizes a low voltage ("logic 0") and red symbolizes a high voltage ("logic 1"). Two cases are shown: (a) input 11 producing output 1, and (b) input 10 producing output 0. A low voltage (Ground) on any input will force the output low (because of the low-resistance path to Ground through the forward-biased diode), and only if all inputs are high (V_{DD}) does the output go high. With more input lines, this same circuit design implements a k-input AND gate.

Fig. 4. (a) Layout of a normal (non-defect-tolerant) demux circuit onto the crossbar. It has a 2-bit input (A0, A1) and a 4-bit output (S00, S01, S10, S11). The thick vertical lines represent wires from the CMOS circuit level, while the thin horizontal lines represent nano-wires. (b) The digital circuit implemented by this layout. (c) Logic equations for the demux digital circuit. (d) The operation of the demux expressed in mathematical form, showing that exactly one of the 2^k wires in the output array **s** will turn on, corresponding to the k-bit address **a** coming into the demux as input. Note that the output wires are labeled in binary to emphasize how they correspond with the set of possible addresses that may occur as inputs on **a** (00, 01, 10, 11).

Each output line has its own address, and the purpose of the AND gate is to "recognize" the wire's address when it appears on the CMOS lines.

Fig. 5. (a) A stuck-open defect (green) causes an AND circuit to give an erroneous output (purple) – a high voltage ("logic 1") when it should have been low ("logic 0"). Compare with Fig. 3, which shows the correct behavior. (b) The erroneous output in the context a demux – two output lines are activated by the address illustrated (10).

Fig. 6. (a) Defect-tolerant demultiplexer. The 2-bit input address **a** passes through an encoder to produce a 3-bit address **u**. Signal vector **u** and its complement are appended to give the 6-bit signal vector **v**, a redundant representation of the input address, which drives the vertical wires in the crossbar. Finally, each horizontal nano-wire computes an AND function, which recognizes the wire's own (encoded) address when it occurs on the vertical wires. The outputs of these four AND gates or recognizers form the 4-bit output vector **s**. This circuit can tolerate one defect on each nanowire output line, and still perform perfectly. (b) Example of the demux circuit with a defect. A single stuck-open defect (green) on output line S11 does not cause an erroneous output. The input address signal is **a**=10, the encoded address is **u**=101, the balanced codeword is **v**=101010, and the (correct) output word is **s**=0010. Because of the internal redundancy of the encoded address, there are two signal lines to pull output line S11 down to zero (blue) for this particular input signal (**a**=10). Either signal line can pull the output low by itself, and so even if one of these two inputs is disconnected (dotted line) by a defect, output line S11 still produces the correct result . Note that the other two unselected output lines (S00 and S01) for this input signal each have two low (blue) voltage connections, and therefore one defect could be tolerated

on each of these nanowires as well. The selected output line (S10) has all of its inputs driven by ones (red), and disconnecting any of these inputs would not affect its output. Single defects on every output line can be tolerated simultaneously. (c) Subsystems of the defect-tolerant demultiplexer. The encoder maps the k-bit input address **a** to an n-bit codeword **u**. The balancer maps every bit of **u** to a pair of bits (one off, one on), resulting in a $2n$-bit balanced codeword **v** (*balanced* means equal numbers of ones and zeroes). The purpose of the balancer is to provide an active-high signal for both the "1" and "0" states of each address bit, as required by the nano-wire AND gates. The signal vector **v** drives the recognizer array, in which the 2^k nano-wires each compute an AND function, which collectively make up the 2^k-bit output vector **s**.

Fig. 7. Two representations for an encoding function for the code {000, 011, 101, 110}. (a) tabulation of inputs and outputs; (b) linear transformation with the given generator matrix G.

Fig. 8. Equivalent representations of the encoding function for the code {000, 011, 101, 110}: (a) expanded version of $\mathbf{u}=\mathbf{a}\cdot G$ using the definition of matrix multiplication; (b) digital circuit implementing this encoding function. The interconnect pattern in the encoder circuit comes directly from the pattern of 1's and 0's in the generator matrix G of the code.

Fig. 9. Geometrical visualizations of two codes, depicting the addresses in the demuxes of Fig. 4a and Fig. 6b. The set of all connected vertices represents an address space. The Hamming distance between a pair of addresses is the number of edges that connect them. The green nodes indicate codewords. It can be seen that in code (a), all the codewords have neighbors at distance

1; whereas in code (b), the nearest neighbor for each codeword is at distance 2.

Fig. 10. Choosing which parts of the demux circuit to implement in nano-circuitry (the recognizer array), versus at the CMOS level (the encoder and balancer circuits).

Fig. 11. Generator matrices for three k=7 codes.

Fig. 12. (a) Generator matrix G for the [12,7,4] code; and (b) the corresponding encoder circuit. The pattern of ones in the generator matrix determines the interconnect pattern in the encoder. Each output bit is computed as the binary sum (XOR) of a subset of the input signals, as specified by the ones in the corresponding column of G. If five input signals must be summed, a 5-input XOR is required; however, when there is only one term (as in the first seven columns), no XOR is required – the single input is wired directly to the output wire.

Fig. 13. The expected percentage of nanowires that can be addressed by different k=7 demultiplexers as a function of the probability p of a defective connection between a nano-wire and a CMOS signal wire. Shown are the cases for demultiplexers based on the codes with d=1, 2, 3 and 4. The d=1 code is the identity function and corresponds to the uncoded demultiplexer, which serves as the baseline in the comparison.

Fig. 14. A schematic illustration of a hybrid electronic circuit that involves blocks of crossbars and demultiplexers (Fig. 2) to enable data from the outside world to be routed into and out of the nanocircuitry. The balancer circuits are included with the encoders in this diagram. Separating

the encoder from the recognizer arrays allows each encoder to control many recognizers; a separate enable signal selects the crossbar to which the data is routed. The black lines denote CMOS wires; the smaller nano-wires are not shown. The encoders are CMOS; the recognizers contain both CMOS wires and nanowires; and the crossbar circuits contain only nanowires.

Fig. 15. Normalized number of expected usable memory bits per chip unit area (as a function of defect rate p) in a crossbar memory system using repeated 128×128 nanowire crossbar blocks. The total circuit area is calculated as in the text, with the encoders shared by many recognizers (as in Fig.14). Shown are the functions for demuxes based on the codes with d = 1, 2, 3 and 4. Note that to achieve a 50% yield, the d=4 coded demux can tolerate up to a p=21% defect rate, whereas an uncoded (d=1) demux can tolerate only 2% defects. We refer to this improvement as the net *coding gain* of the design.

Fig. 16. Similarly to the case of erasure correction in a communication channel, the defect-tolerant demux (based on a particular code) can tolerate more mistakes than can be corrected when the same code is used for full error-correction. The small black dots are points in the address space, separated according to Hamming distance, and the dots shaded blue are codewords for the d=7 code represented. The circles represent r-spheres, the set of all addresses within r units of Hamming distance from a given address. (a) For the full error correction case, a received (noise-corrupted) message can land anywhere in the address space, and we must find the closest codeword to it. The sphere of radius r=3 around the codewords shows how far a corrupted codeword can stray from its actual value and still be correctable. (b) For the demux circuit, the radius r=6 sphere around a codeword shows that nano-wires occur only at codewords,

and that a pair of nano-wires with neighboring addresses can each have up to 6 defects and still not interfere with each other.

Fig. 17. Plots of (normalized) expected addressable nanowires vs. defect probability *p* for two 128-wire demuxes --a [7,7,1] uncoded demux (green) and a demux based on a [12,7,4] code (red). The solid curves show the expected value calculated using the exact expression derived in Appendix A. The black dots and error bars show the results of runs with the digital circuit simulator. The dots and error bars show the mean and variance (one standard deviation) for 100 simulation runs at each value of *p*, where *p* was stepped through the values *p* = 0.00, 0.05, 0.10, ... 1.00.

Fig. 1.

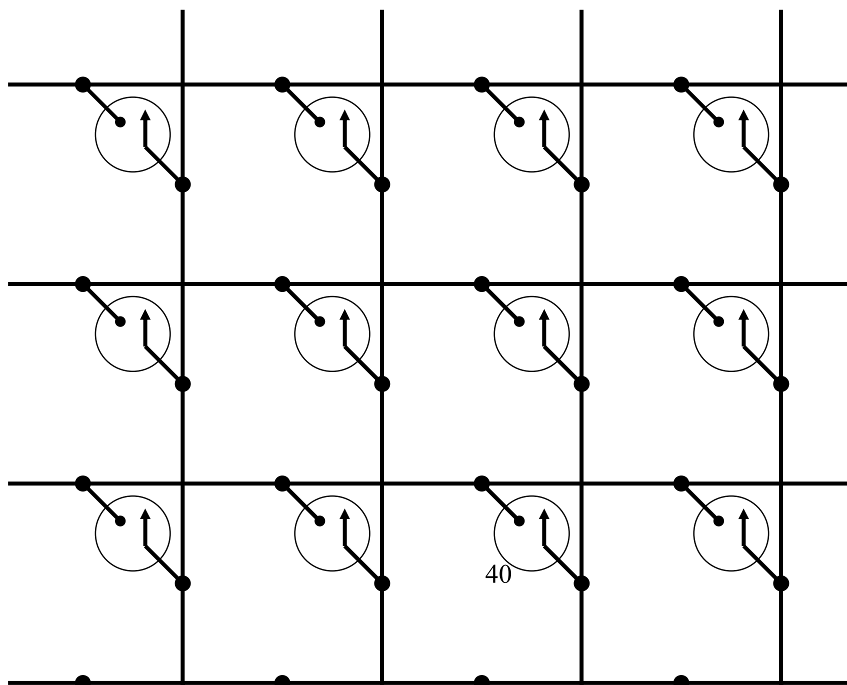

Fig. 2.

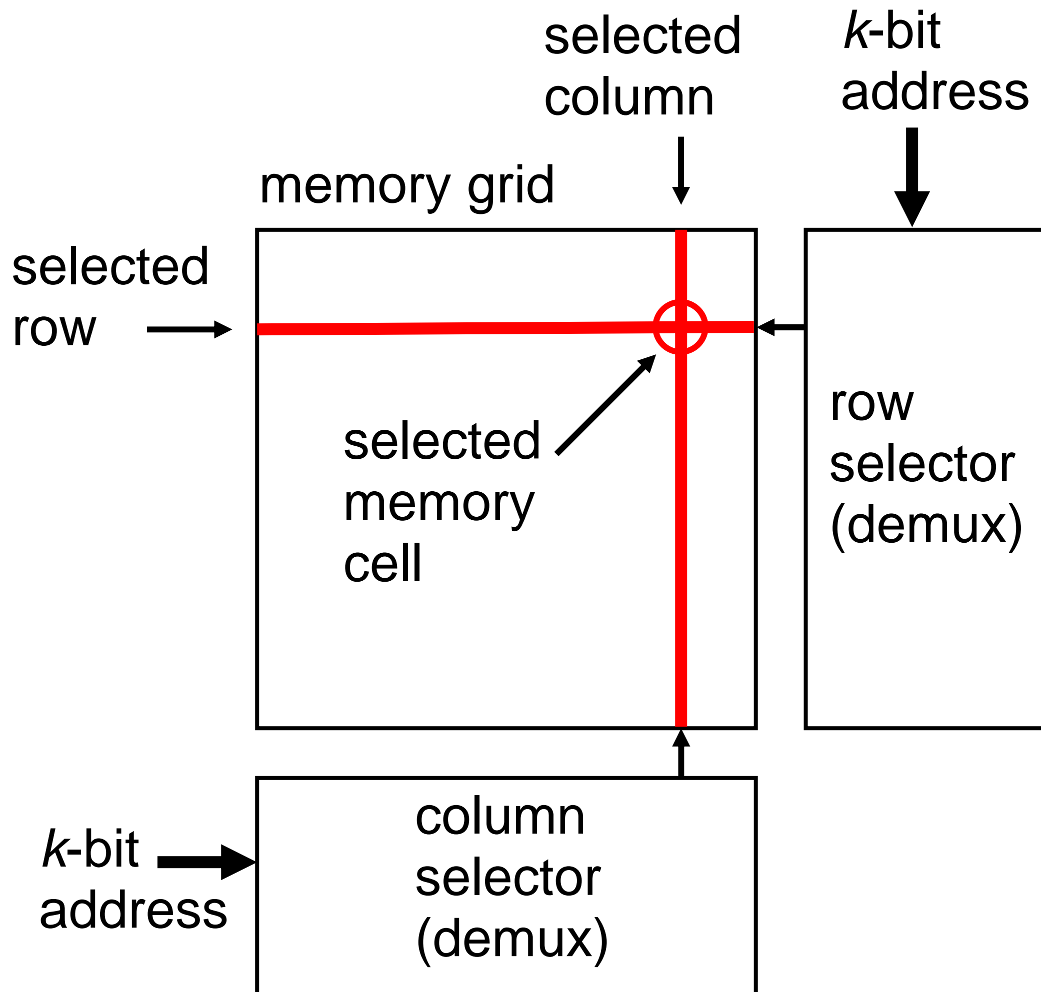

Fig. 3.

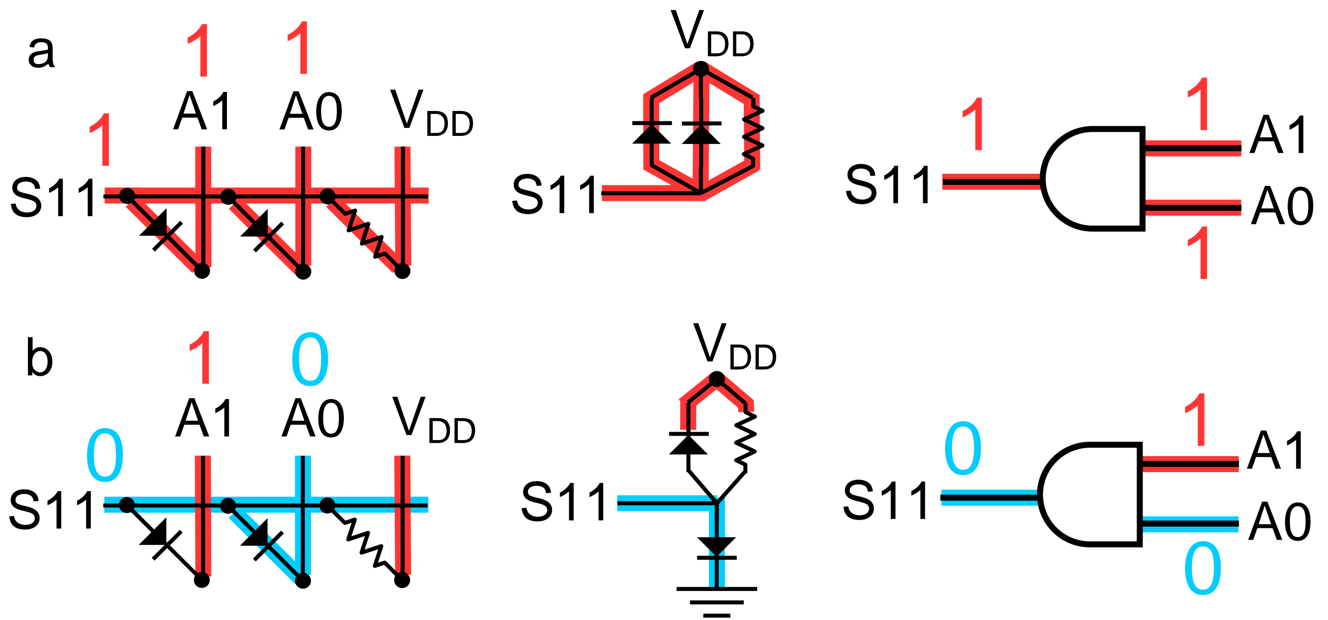

Fig. 4

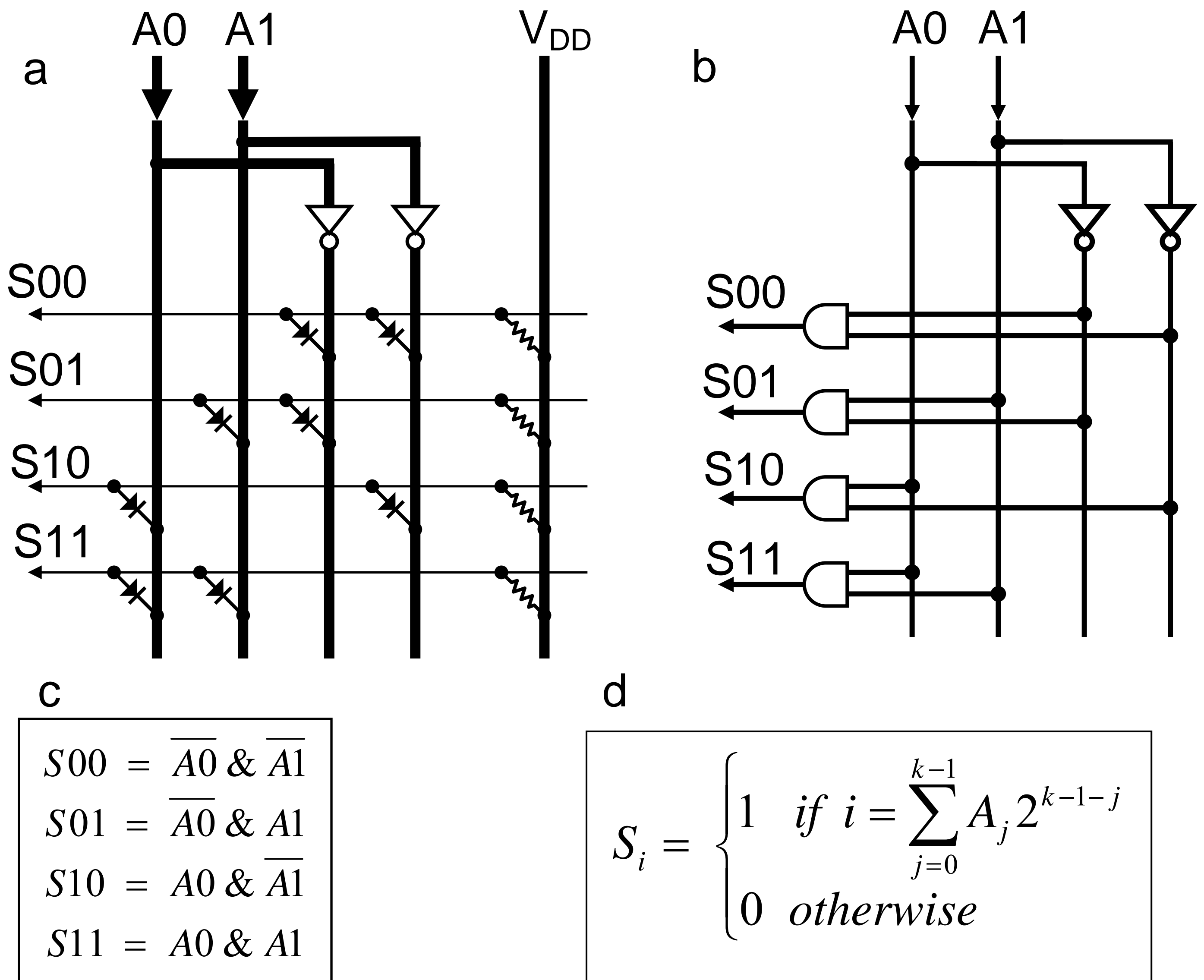

Fig. 5.

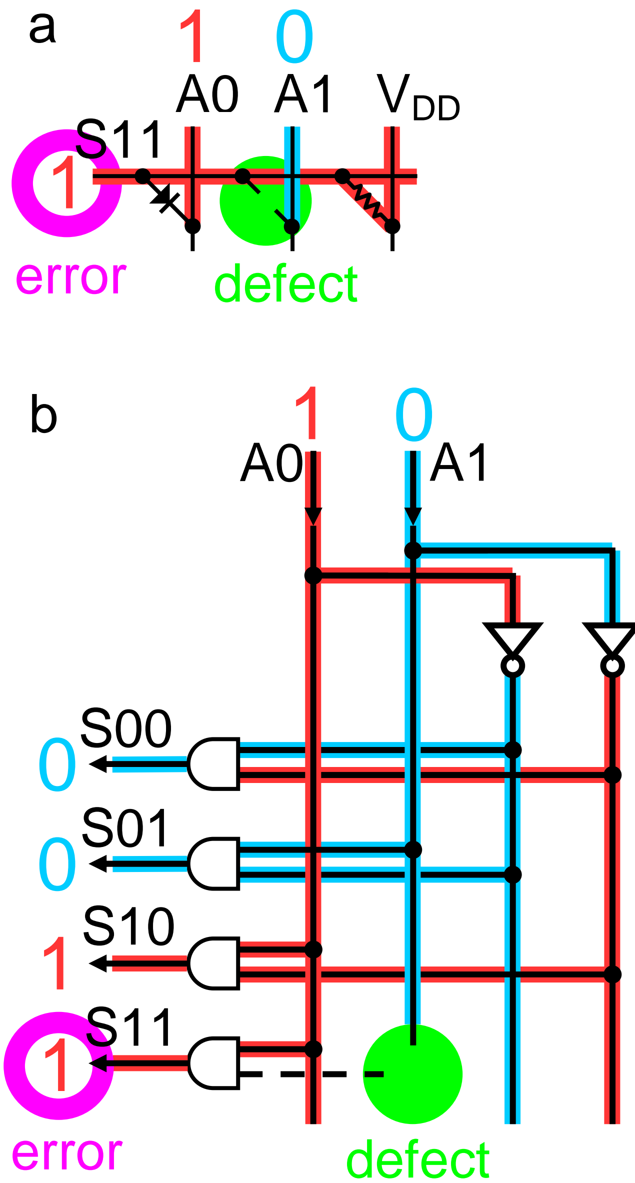

Fig. 6.

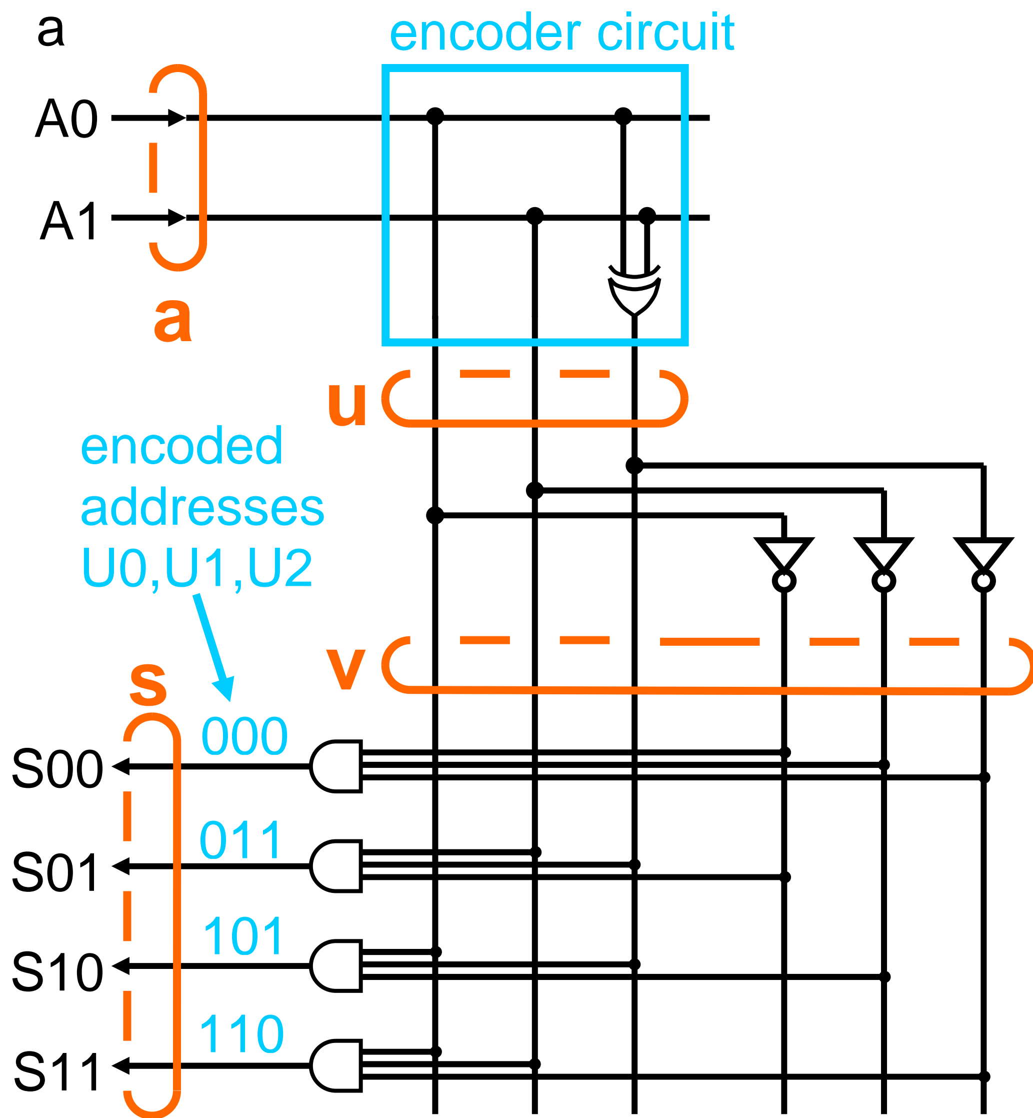

Fig. 6. (continued)

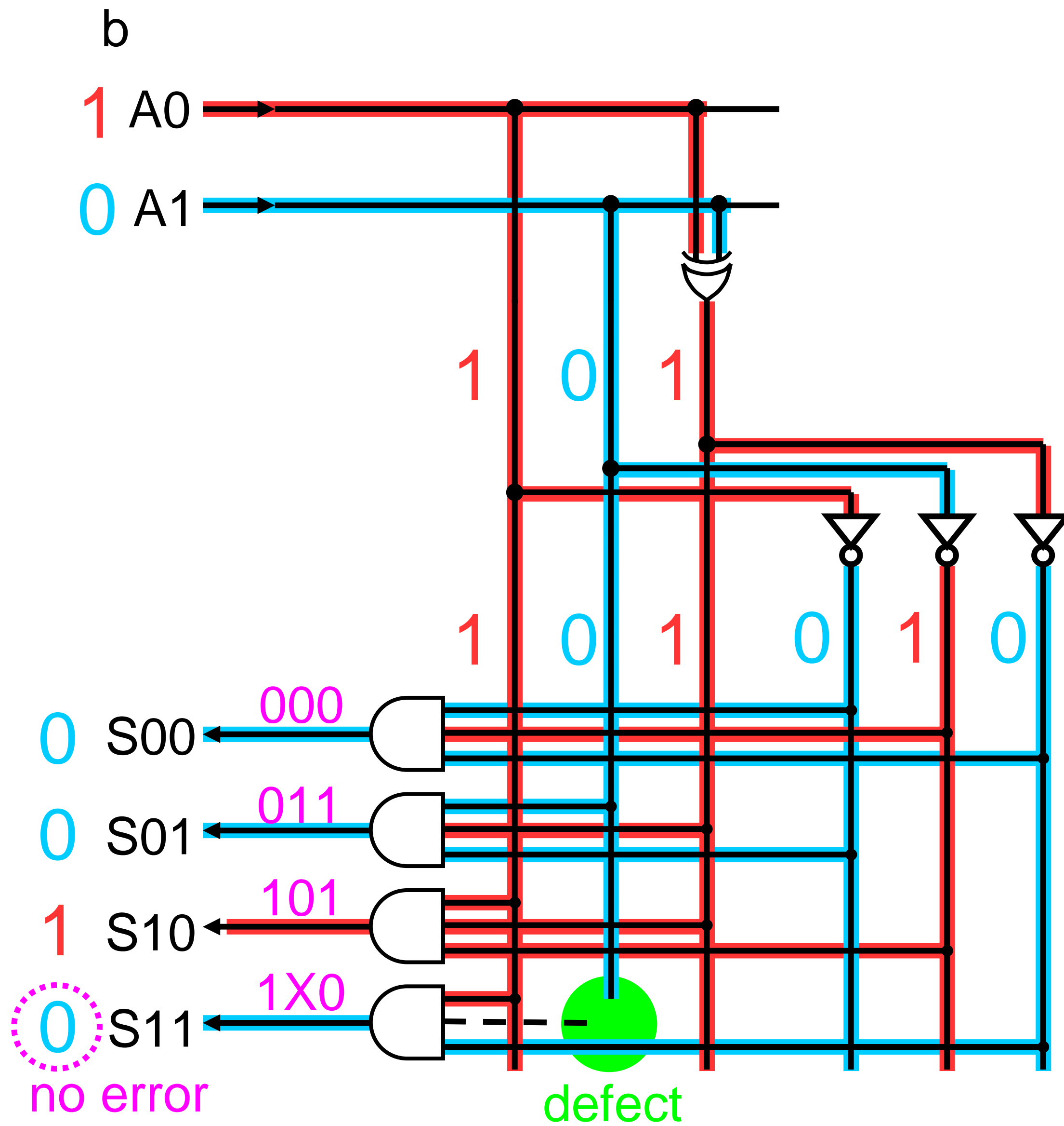

Fig. 6 (continued).

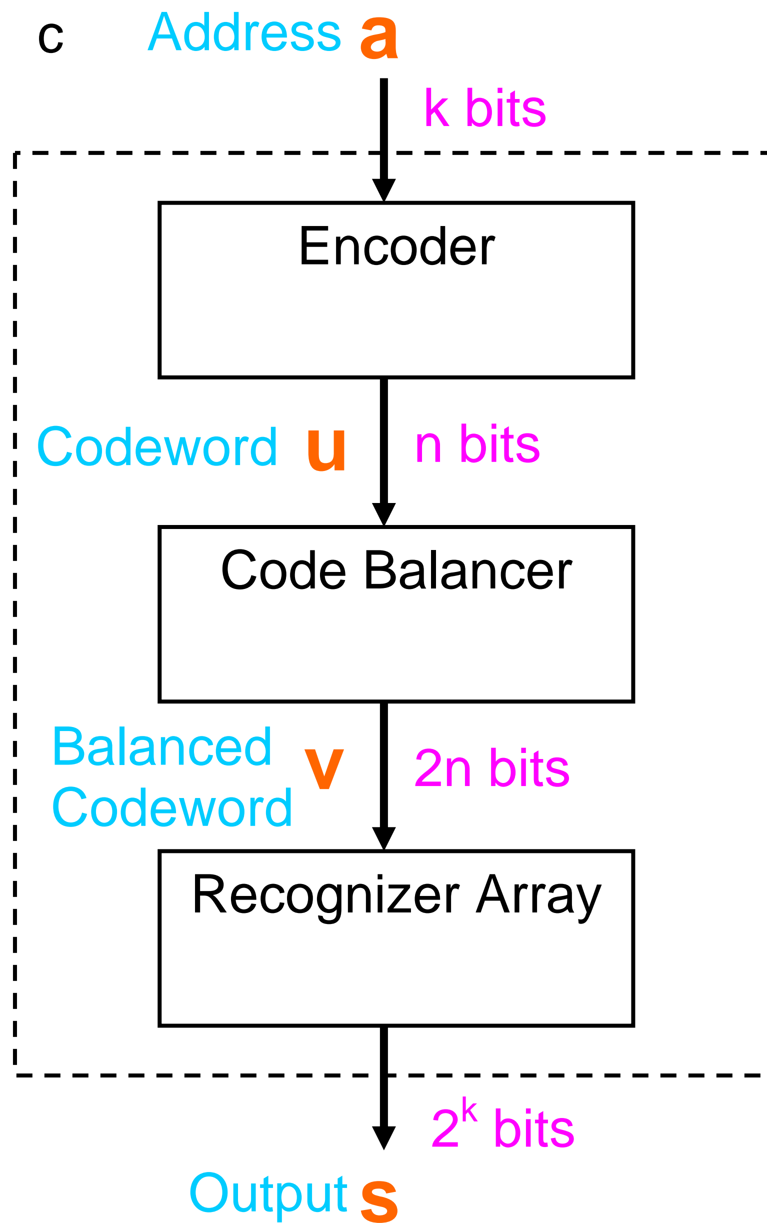

Fig. 7.

a

a		**u**
00	→	000
01	→	011
10	→	101
11	→	110

b

$$\mathbf{u} = \mathbf{a} \cdot G$$

$$where \quad G = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Fig. 8.

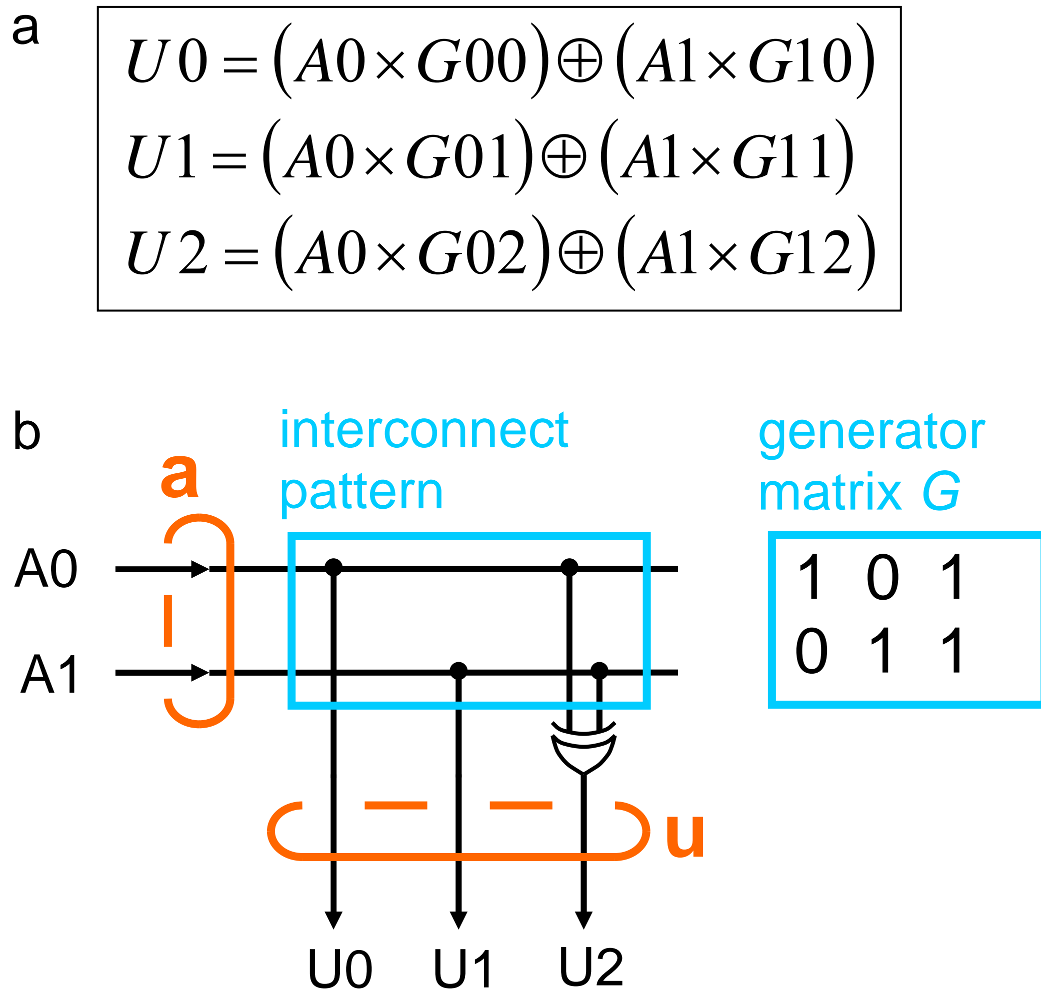

Fig. 9.

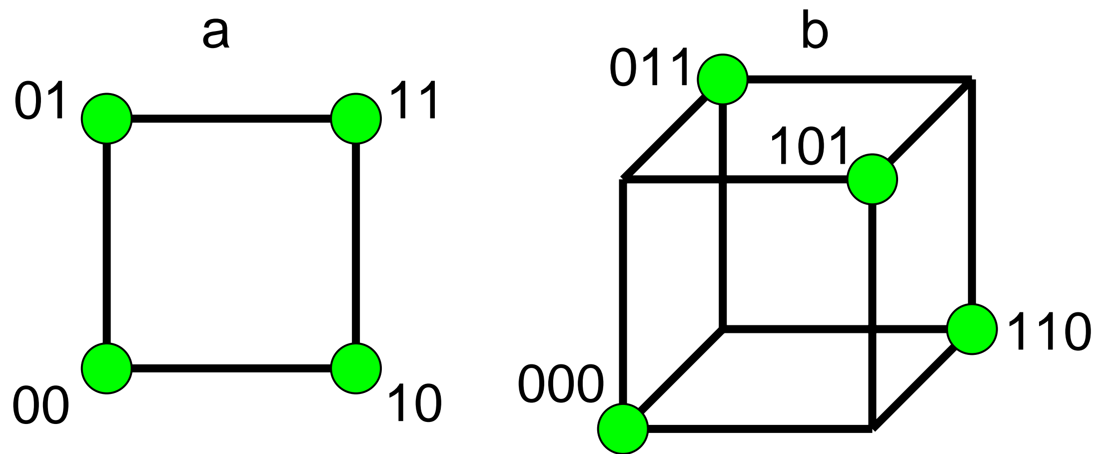

Fig. 10.

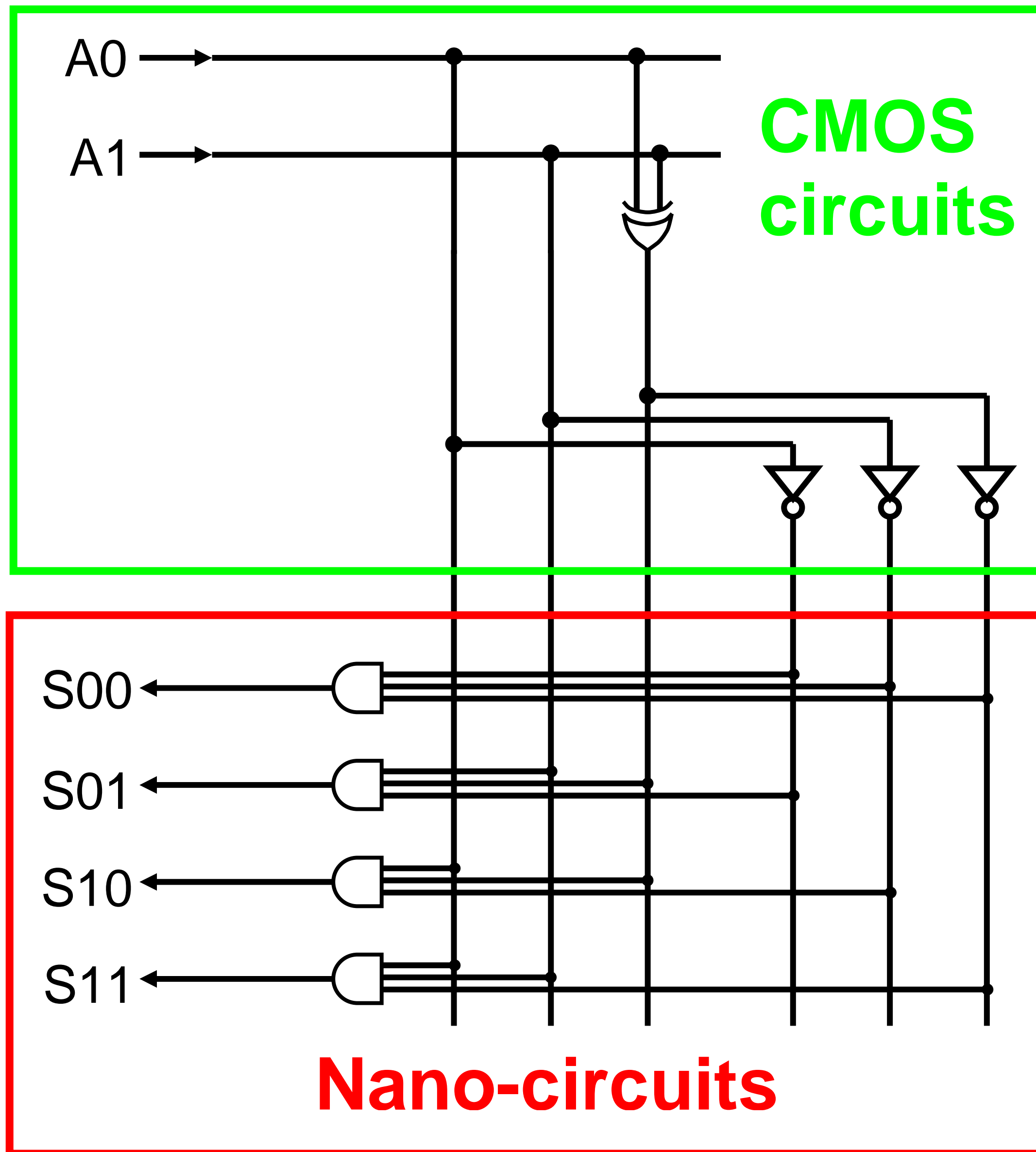

Fig. 11.

[7,7,1]

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

[8,7,2]

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

[11,7,3]

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

Fig. 12.

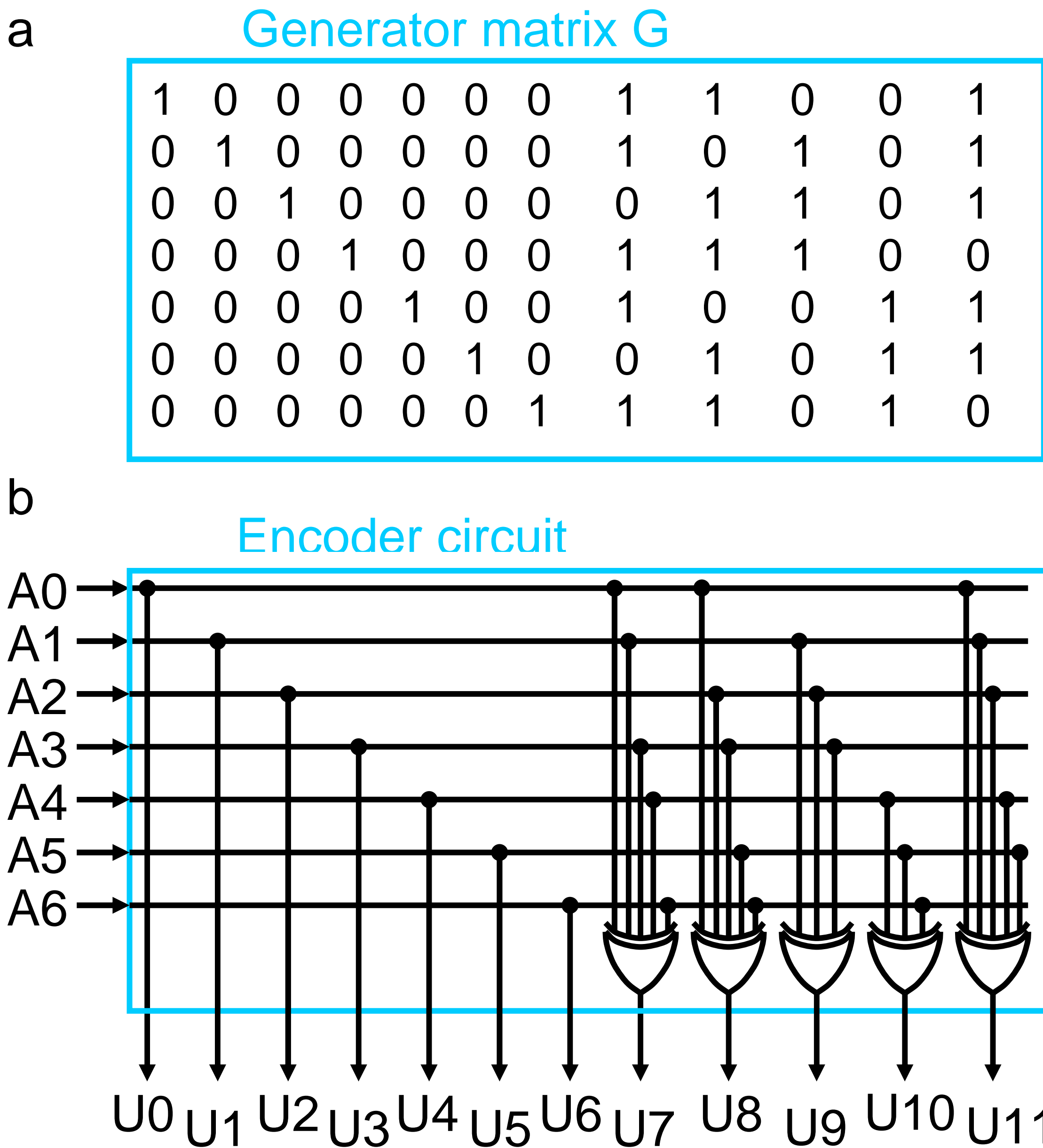

Fig. 13.

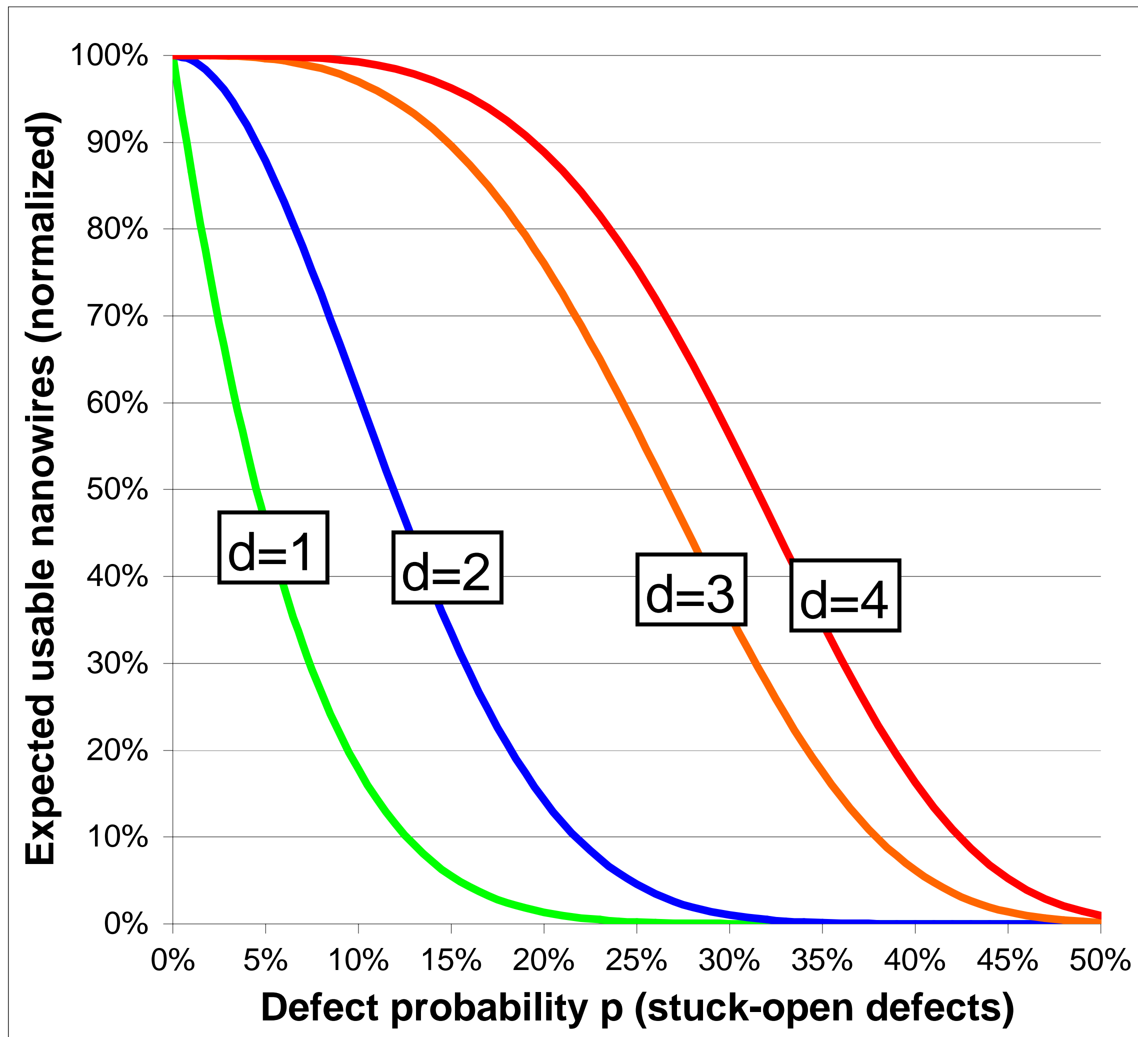

Fig. 14.

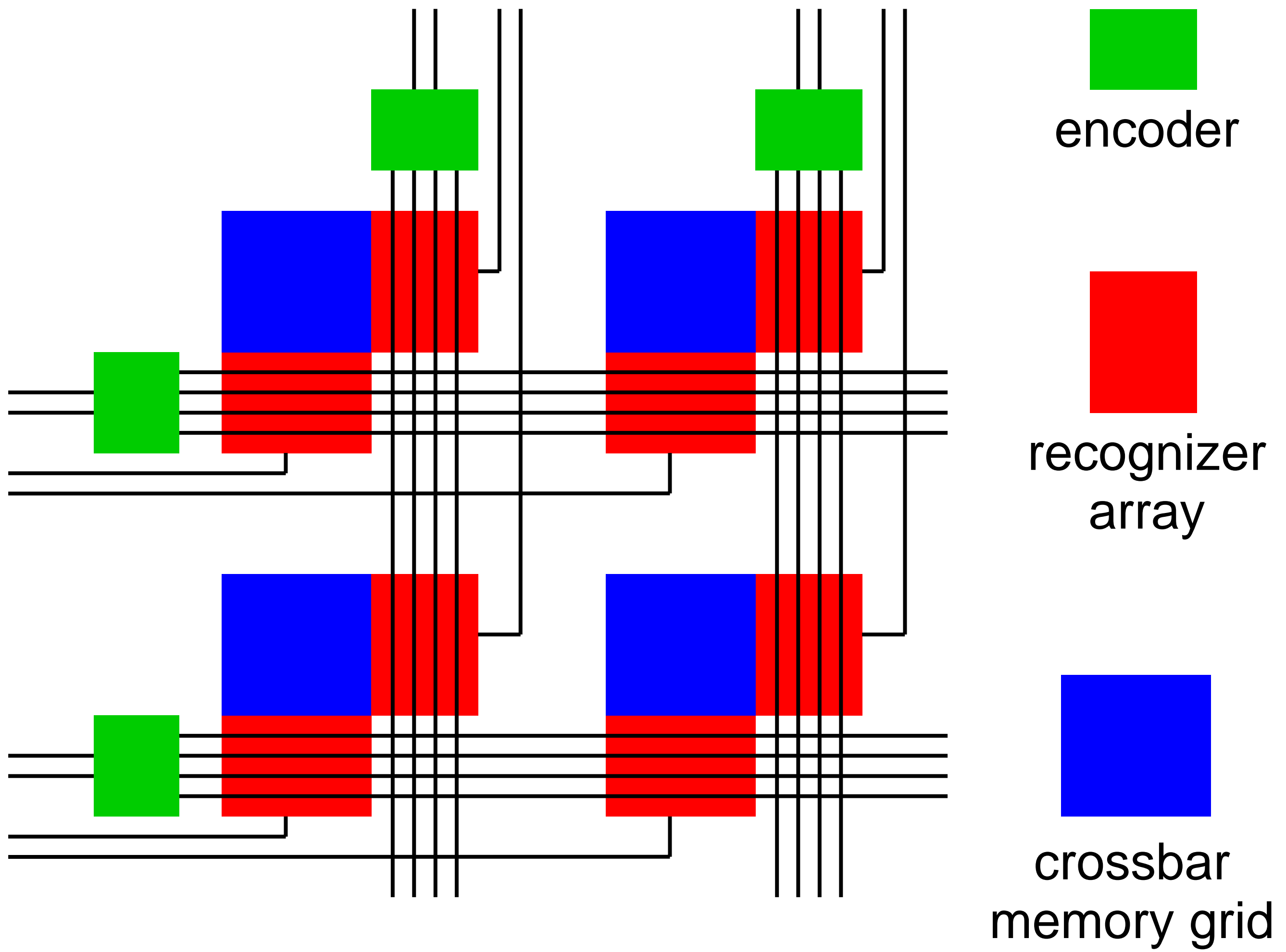

Fig. 15.

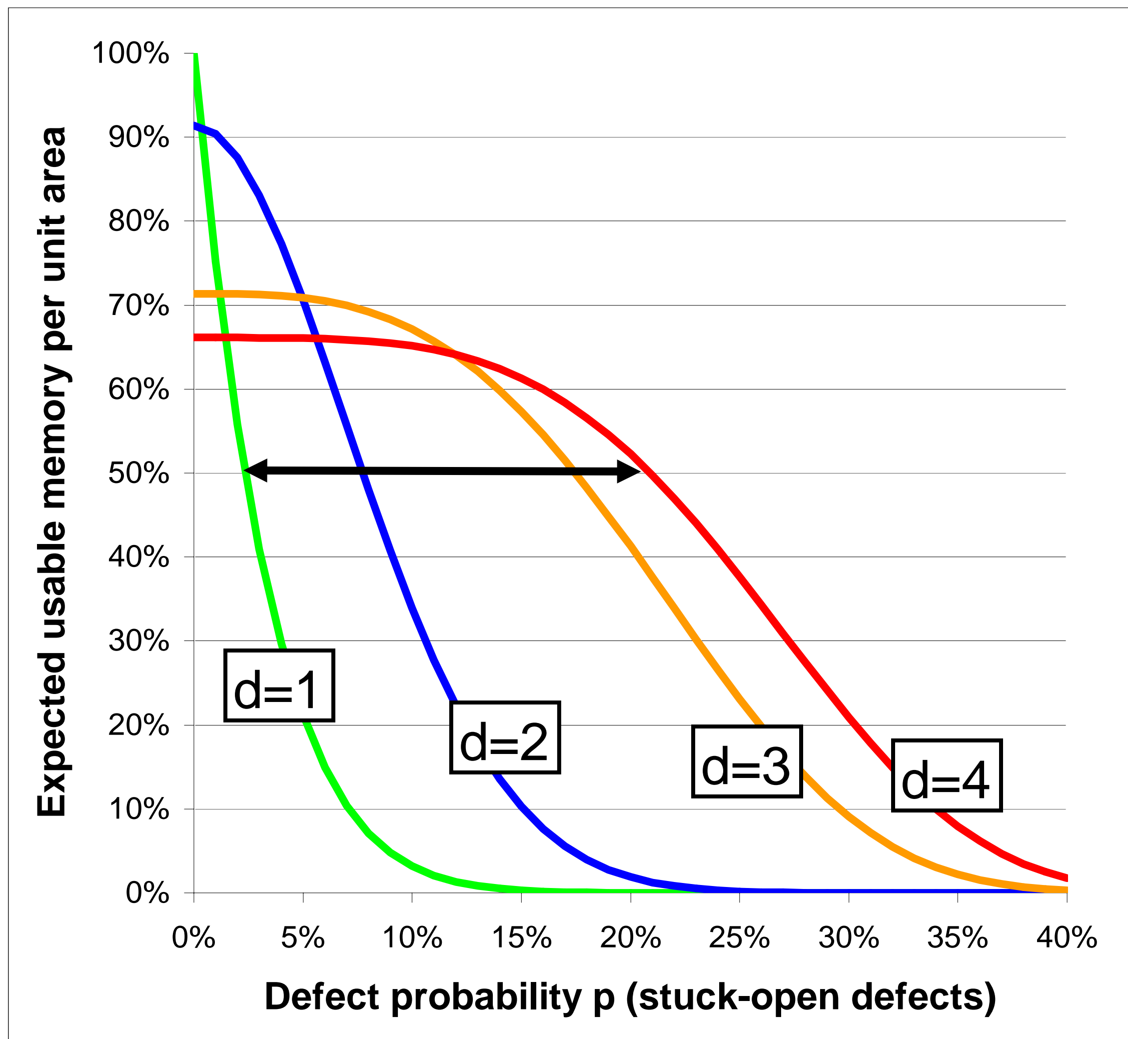

Fig. 16.

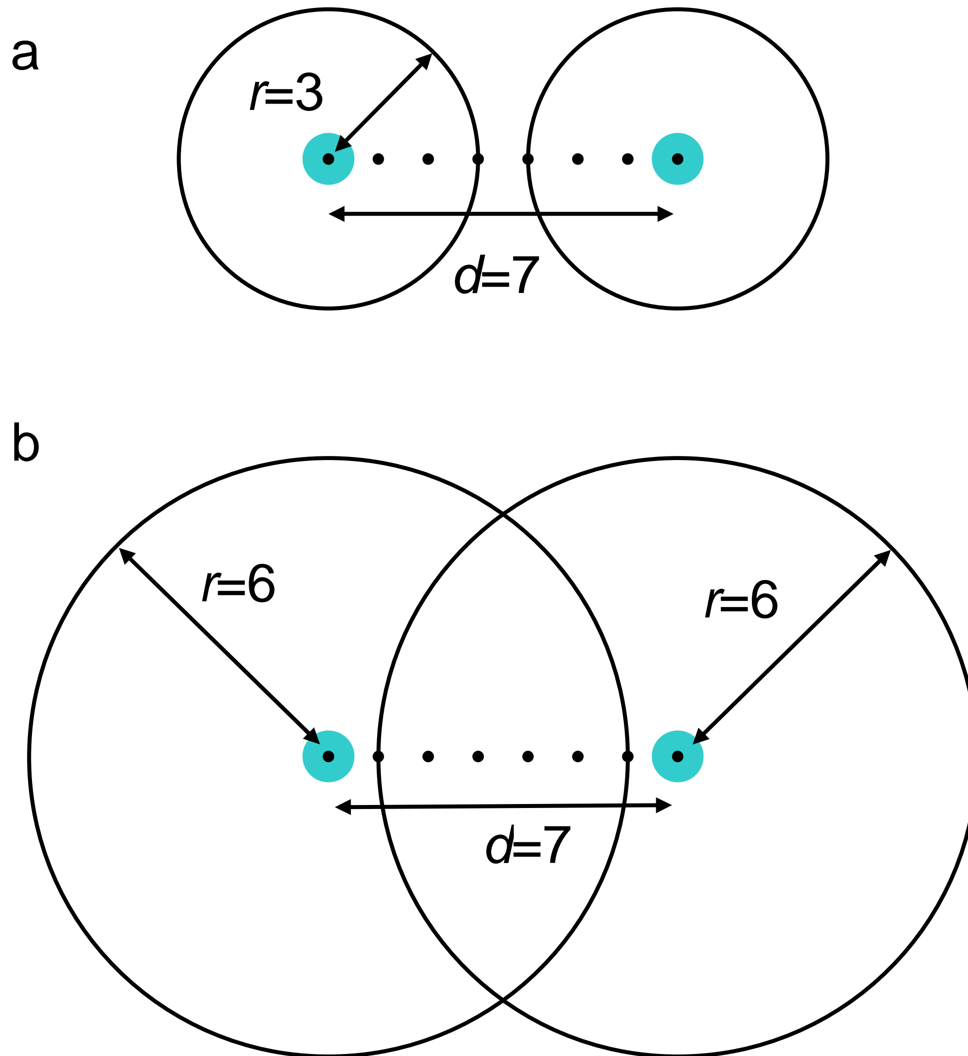

Fig. 17.