Error detection and correction for memory bit failures

Appendix B of CSF text gives basic idea
Hamming code: Simple and elegant
Note: Problems on last page

Slides taken from same Stallings text used in the Chap 7 memory slides
Suppose that a single data bit must be stored in memory. For example, 1 = "yes" and 0 = "no"

But assume that a single bit error in the memory is possible and must be detected.

How many bits (in addition to the data bit) must actually be stored in the memory to provide single bit error detection?

**Strategy**: Append a 2\textsuperscript{nd} bit to the data bit called a parity bit. For example, the parity bit can be such that the resulting number of "1" bits in the two stored bits (that is, the data bit and parity bit) is even.

Note: zero "1" bits will be considered an even number of "1" bits.

Example: 0 is stored as 00 and 1 is stored as 11. If the 2 stored bits are read, and a single bit error occurs such that a 01 or 10 is observed, then an error is detected.

An odd parity bit can also be used.
Note that a single bit error cannot be corrected with a parity bit.

That is, if a 01 is read, it could result from either a single bit error in 00 or 11.

Note that a double bit error will not be detected.

Repeat the strategy for storing 2 bits (00, 01, 10, or 11)) with a parity bit to achieve single bit error detection. Repeat for 4 bits. Repeat for $2^n$ bits.

The overhead of the above strategy for $2^n$ bits any value of $n$ would be always be only 1 bit.

But for large $n$, the assumption that there is at most only a single bit error become weaker.
Suppose again that a single bit of information must be stored.

But a single bit error is possible in the memory and must be detected **and corrected**?

How many bits can be stored in the memory to provide single bit error detection and correction?

**Strategy:** For each bit to be stored, store the bit in **triplicate**. This is called **triple redundancy**. 0 is stored as 000 and 1 is stored as 111.

Correction example: Assuming only a single bit error, 010 is correctable to 000.

Repeat for storing 2 bits in memory and single bit error detection and correction by using triple redundancy individually for each bit.

00 would be stored as 000 000
11 would be stored as 111 111

Repeating for $2^n$ bits, the overhead/cost is significant.
A 1-bit parity scheme is an error-detecting code; there are also error-correcting codes (ECC) that will detect and allow correction of an error. For large main memories, many systems use a code that allows the detection of up to 2 bits of error and the correction of a single bit of error. These codes work by using more bits to encode the data; for example, the typical codes used for main memories require 7 or 8 bits for every 128 bits of data.

**Elaboration:** A 1-bit parity code is a *distance-2 code*, which means that if we look at the data plus the parity bit, no 1-bit change is sufficient to generate another legal combination of the data plus parity. For example, if we change a bit in the data, the parity will be wrong, and vice versa. Of course, if we change 2 bits (any 2 data bits or 1 data bit and the parity bit), the parity will match the data and the error cannot be detected. Hence, there is a distance of two between legal combinations of parity and data.

To detect more than one error or correct an error, we need a *distance-3 code*, which has the property that any legal combination of the bits in the error correction code and the data have at least 3 bits differing from any other combination. Suppose we have such a code and we have one error in the data. In that case the code plus data will be 1 bit away from a legal combination and we can correct the data to that legal combination. If we have two errors, we can recognize that there is an error, but we cannot correct the errors. Let’s look at an example. Here are the data words and a distance-3 error correction code for a 4-bit data item.

<table>
<thead>
<tr>
<th>Data</th>
<th>Code bits</th>
<th>Data</th>
<th>Code bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000</td>
<td>1000</td>
<td>111</td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
<td>1001</td>
<td>100</td>
</tr>
<tr>
<td>0010</td>
<td>101</td>
<td>1010</td>
<td>010</td>
</tr>
<tr>
<td>0011</td>
<td>110</td>
<td>1011</td>
<td>001</td>
</tr>
<tr>
<td>0100</td>
<td>110</td>
<td>1100</td>
<td>001</td>
</tr>
<tr>
<td>0101</td>
<td>101</td>
<td>1101</td>
<td>010</td>
</tr>
<tr>
<td>0110</td>
<td>011</td>
<td>1110</td>
<td>100</td>
</tr>
<tr>
<td>0111</td>
<td>000</td>
<td>1111</td>
<td>111</td>
</tr>
</tbody>
</table>
KEY POINTS

- The two basic forms of semiconductor random-access memory are dynamic RAM (DRAM) and static RAM (SRAM). SRAM is faster, more expensive, and less dense than DRAM, and is used for cache memory. DRAM is used for main memory.
- Error correction techniques are commonly used in memory systems. These involve adding redundant bits that are a function of the data bits to form an error-correcting code. If a bit error occurs, the code will detect and, usually, correct the error.
- To compensate for the relatively slow speed of DRAM, a number of advanced DRAM organizations have been introduced. The two most common are synchronous DRAM and RamBus DRAM. Both of these involve using the system clock to provide for the transfer of blocks of data.

A memory cache is a small-size (relative to number of bytes/words that can be stored in comparison with the main memory) that has fast access times (read and write times) so as to provide instructions and operands to the microprocessor (CPU) at rates that optimize instruction execution. (Covered in Chapter 7)
A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A **hard failure** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data, but become stuck at 0 or 1 or switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A **soft error** is a random, non-destructive event that alters the contents of one or more memory cells, without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials.

**Key idea for bit-flip error detection and correction.**

Given an $M$ bit data word, generate and append $K$ additional bits such that the $M+K$ bits together support error detection/correction under the condition that there is an upper bound on the total number of bits that can be in error.

For example, for an $M$ bit data word, append an **even parity bit** such that the total number of “1” bits in the resulting $M+1$ bit word is even. Similarly, an odd parity bit can be appended. With an even parity bit, if at most 1 bit is in error, the error is easily detected because the total number of 1 bits will not be even. However, the bit error cannot be identified (and, hence, corrected).

Example: 8 bit data word = 00110010

9 bit data word with even parity bit = 001100101

Even parity bit
Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 5.7 illustrates in general terms how the process is carried out. When data are to be read into memory, a calculation, depicted as a function $f$, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an $M$-bit word of data is to be stored, and the code is of length $K$ bits, then the actual size of the stored word is $M + K$ bits.
Figure 5.8  Hamming Error-Correcting Code

By checking the parity bits, discrepancies are found in circle A and circle C but not in circle B. Only one of the seven compartments is in A and C but not B. The error can therefore be corrected by changing that bit.
To clarify the concepts involved, we will develop a code that can detect and correct single-bit errors in 8-bit words.

To start, let us determine how long the code must be. Referring to Figure 5.7, the comparison logic receives as input two $K$-bit values. A bit-by-bit comparison is done by taking the exclusive-or of the two inputs. The result is called the syndrome word. Thus, each bit of the syndrome is 0 or 1 according to if there is or is not a match in that bit position for the two inputs.

The syndrome word is therefore $K$ bits wide and has a range between 0 and $2^K - 1$. The value 0 indicates that no error was detected, leaving $2^K - 1$ values to indicate, if there is an error, which bit was in error. Now, because an error could occur on any of the $M$ data bits or $K$ check bits, we must have

$$2^K - 1 \geq M + K$$

This inequality gives the number of bits needed to correct a single bit error in a word containing $M$ data bits. For example, for a word of 8 data bits ($M = 8$), we have

- $K = 3$: $2^3 - 1 < 8 + 3$
- $K = 4$: $2^4 - 1 > 8 + 4$

Thus, eight data bits require four check bits. The first three columns of Table 5.2 lists the number of check bits required for various data word lengths.
C1 = D1 ⊕ D2 ⊕ D4 ⊕ D5 ⊕ D7
C2 = D1 ⊕ D3 ⊕ D4 ⊕ D6 ⊕ D7
C4 = D2 ⊕ D3 ⊕ D4 ⊕ D8
C8 = D5 ⊕ D6 ⊕ D7 ⊕ D8

<table>
<thead>
<tr>
<th>Bit position</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>1100</td>
<td>1011</td>
<td>1010</td>
<td>1001</td>
<td>1000</td>
<td>0111</td>
<td>0110</td>
<td>0101</td>
<td>0100</td>
<td>0011</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>Data bit</td>
<td>D8</td>
<td>D7</td>
<td>D6</td>
<td>D5</td>
<td>D4</td>
<td>D3</td>
<td>D2</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check bit</td>
<td>C8</td>
<td>C4</td>
<td>C2</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.9 Layout of Data Bits and Check Bits

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.
- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in Figure 5.9. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check bits. The check bits are calculated as follows, where the symbol ⊕ designates the exclusive-or operation:
Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit. Thus, data bit positions 3, 5, 7, 9, and 11 (D1, D2, D4, D5, D7) all contain a 1 in the least significant bit of their position number as does C1; bit positions 3, 6, 7, 10, and 11 all contain a 1 in the second bit position, as does C2; and so on. Looked at another way, bit position $n$ is checked by those bits $C_i$ such that $\Sigma i = n$. For example, position 7 is checked by bits in position 4, 2, and 1; and $7 = 4 + 2 + 1$.

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

- $C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$
- $C2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$
- $C4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$
- $C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

- $C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$
- $C2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$
- $C4 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$
- $C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$

When the new check bits are compared with the old check bits, the syndrome word is formed:

<table>
<thead>
<tr>
<th></th>
<th>C8</th>
<th>C4</th>
<th>C2</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.
<table>
<thead>
<tr>
<th>Bit position</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position number</td>
<td>1100</td>
<td>1011</td>
<td>1010</td>
<td>1001</td>
<td>1000</td>
<td>0111</td>
<td>0110</td>
<td>0101</td>
<td>0100</td>
<td>0011</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>Data bit</td>
<td>D8</td>
<td>D7</td>
<td>D6</td>
<td>D5</td>
<td>D4</td>
<td>D3</td>
<td>D2</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check bit</td>
<td>C8</td>
<td>C4</td>
<td>C2</td>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word stored as</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Word fetched as</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Position number</td>
<td>1100</td>
<td>1011</td>
<td>1010</td>
<td>1001</td>
<td>1000</td>
<td>0111</td>
<td>0110</td>
<td>0101</td>
<td>0100</td>
<td>0011</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>Check bit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.10**  Check Bit Calculation
Table 5.2  Increase in Word Length with Error Correction

<table>
<thead>
<tr>
<th>Data Bits</th>
<th>Check Bits</th>
<th>% Increase</th>
<th>Check Bits</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>50</td>
<td>5</td>
<td>62.5</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>31.25</td>
<td>6</td>
<td>37.5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>18.75</td>
<td>7</td>
<td>21.875</td>
</tr>
<tr>
<td>64</td>
<td>7</td>
<td>10.94</td>
<td>8</td>
<td>12.5</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>6.25</td>
<td>9</td>
<td>7.03</td>
</tr>
<tr>
<td>256</td>
<td>9</td>
<td>3.52</td>
<td>10</td>
<td>3.91</td>
</tr>
</tbody>
</table>
Figure 5.11  Hamming SEC-DEC Code

Suppose this bit is flipped as the 1st Bit error
By itself, it is detectable and correctable

2nd Bit error fools the check bits
Using an additional check bit
which provides even parity over all 8 bits, the double bit error is
detected, but cannot be corrected
Figure 5.11 illustrates how such a code works, again with a 4-bit data word. The sequence shows that if two errors occur (Figure 5.11c), the checking procedure goes astray (d) and worsens the problem by creating a third error (e). To overcome the problem, an eighth bit is added that is set so that the total number of 1s in the diagram is even. The extra parity bit catches the error (f).
Homework #9 Problems. Due Wed. 6/8

(1) Given a 32 bit MIPS operand, show a table such as that in Figure 5.9 for Hamming coding that specifies the necessary check bits that must be added and their positions so as to provide single bit error correction and double bit error detection.

(2) Suppose that a 32 bit word consists of 16 “1” bits followed by 16 “0” bits:

\[ 11111111111111110000000000000000 \]

If this bit is flipped, illustrate how this bit error is detected/corrected using Hamming code.

\[ 11111111111111110000000000000000 \]

(3) Suppose the first and last bits are both flipped. Illustrate how this double bit error is detected using Hamming code. Given this double bit error, if it was mistakenly assumed that there was only a single bit error, what correction would be attempted.

(4) For a 256 bit string of bits, how many check bits would have to be added to provide single error detection/correction with Hamming encoding?