EECS150 - Digital Design
Lecture 26 – Error Correction Codes, Linear Feedback Shift Registers (LFSRs)

Nov 21, 2002
John Wawrzynek

Outline

• Error detection using parity
• Hamming code for error detection/correction
• Linear Feedback Shift Registers
  – Theory and practice
Error Correction Codes (ECC)

- Memory systems generate errors (accidentally flipped-bits)
  - DRAMs store very little charge per bit
  - "Soft" errors occur occasionally when cells are struck by alpha particles or other environmental upsets.
  - Less frequently, "hard" errors can occur when chips permanently fail.
- Where "perfect" memory is required
  - servers, spacecraft/military computers, ...
- Memories are protected against failures with ECCs
- Extra bits are added to each data-word
  - extra bits are used to detect and/or correct faults in the memory system
  - in general, each possible data word value is mapped to a unique "code word". A fault changes a valid code word to an invalid one - which can be detected.

Simple Error Detection Coding

Parity Bit

- Each data value, before it is written to memory is "tagged" with an extra bit to force the stored word to have even parity:

- Each word, as it is read from memory is "checked" by finding its parity (including the parity bit).

- A non-zero parity indicates an error occurred:
  - two errors (on different bits) is not detected (nor any even number of errors)
  - odd numbers of errors are detected.
Hamming Error Correcting Code

- Use more parity bits to pinpoint bit(s) in error, so they can be corrected.
- Example: Single error correction (SEC) on 4-bit data
  - use 3 parity bits, with 4-data bits results in 7-bit code word
  - 3 parity bits sufficient to identify any one of 7 code word bits
  - overlap the assignment of parity bits so that a single error in the 7-bit work can be corrected
- Group parity bits so they correspond to subsets of the 7 bits:
  - \( p_1 \) protects bits 1,3,5,7
  - \( p_2 \) protects bits 2,3,6,7
  - \( p_3 \) protects bits 4,5,6,7

<table>
<thead>
<tr>
<th>Bit position number</th>
<th>001 = 1_{10}</th>
<th>011 = 3_{10}</th>
<th>101 = 5_{10}</th>
<th>111 = 7_{10}</th>
<th>p_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>010 = 2_{10}</td>
<td></td>
<td></td>
<td></td>
<td>011 = 3_{10}</td>
<td>p_2</td>
</tr>
<tr>
<td>110 = 6_{10}</td>
<td></td>
<td></td>
<td></td>
<td>111 = 7_{10}</td>
<td>p_3</td>
</tr>
<tr>
<td>100 = 4_{10}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 = 5_{10}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 = 6_{10}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111 = 7_{10}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hamming Code Example

1 2 3 4 5 6 7
p_1 p_2 d_1 p_3 d_2 d_3 d_4

- Note: parity bits occupy power-of-two bit positions in code-word.
- On writing to memory:
  - parity bits are assigned to force even parity over their respective groups.
- On reading from memory:
  - check bits \((c_1, c_2, c_3)\) are generated by finding the parity of the group and its parity bit. If an error occurred in a group, the corresponding check bit will be 1, if no error the check bit will be 0.
  - check bits \((c_1, c_2, c_3)\) form the position of the bit in error.

Example: \( c = c_2c_3c_4 = 101 \)
- error in 4,5,6, or 7 (by \( c_3 = 1 \))
- error in 1,3,5, or 7 (by \( c_1 = 1 \))
- no error in 2, 3, 6, or 7 (by \( c_2 = 0 \))
- Therefore error must be in bit 5.
- Note the check bits point to 5
- By our clever positioning and assignment of parity bits, the check bits always address the position of the error!
- \( c = 000 \) indicates no error
Hamming Error Correcting Code

- Overhead involved in single error correction code:
  - Let $p$ be the total number of parity bits and $d$ the number of data bits in a $p + d$ bit word.
  - If $p$ error correction bits are to point to the error bit ($p + d$ cases) plus indicate that no error exists (1 case), we need:
    \[ 2^p \geq p + d + 1, \]
    thus $p \geq \log(p + d + 1)$
    for large $d$, $p$ approaches $\log(d)$

- Adding on extra parity bit covering the entire word can provide double error detection
  \[
  1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8
  
  p_1 \quad p_2 \quad d_1 \quad p_3 \quad d_2 \quad d_3 \quad d_4 \quad p_4
  \]

- On reading the $C$ bits are computed (as usual) plus the parity over the entire word, $P$:
  \[
  C=0 \quad P=0, \text{ no error}
  
  C\neq0 \quad P=1, \text{ correctable single error}
  
  C\neq0 \quad P=0, \text{ a double error occurred}
  
  C=0 \quad P=1, \text{ an error occurred in } p_4 \text{ bit}

Typical modern codes in DRAM memory systems:
- 64-bit data blocks (8 bytes) with 72-bit code words (9 bytes).

Linear Feedback Shift Registers (LFSRs)

- These are $n$-bit counters exhibiting pseudo-random behavior.
- Built from simple shift-registers with a small number of xor gates.
- Used for:
  - pseudo-random number generation
  - counters
  - error checking and correction
- Advantages:
  - very little hardware
  - high speed operation
- Example 4-bit LFSR:
4-bit LFSR

- Circuit counts through $2^4 - 1$ different non-zero bit patterns.
- Leftmost bit decides whether the "10011" xor pattern is used to compute the next value or if the register just shifts left.
- Can build a similar circuit with any number of FFs, may need more xor gates.
- In general, with $n$ flip-flops, $2^n - 1$ different non-zero bit patterns.
- (Intuitively, this is a counter that wraps around many times and in a strange way.)

Applications of LFSRs

- Performance:
  - In general, xors are only ever 2-input and never connect in series.
  - Therefore the minimum clock period for these circuits is:
    \[ T > T_{2\text{-input-xor}} + \text{clock overhead} \]
  - Very little latency, and independent of $n$!
- This can be used as a fast counter, if the particular sequence of count values is not important.
  - Example: micro-code micro-pc
- Can be used as a random number generator.
  - Sequence is a pseudo-random sequence:
    - numbers appear in a random sequence
    - repeats every $2^n - 1$ patterns
  - Random numbers useful in:
    - computer graphics
    - cryptography
    - automatic testing
- Used for error detection and correction
  - CRC (cyclic redundancy codes)
  - ethernet uses them
Galois Fields - The theory behind LFSRs

- LFSR circuits performs multiplication on a field.
- A field is defined as a set with the following:
  - two operations defined on it:
    - “addition” and “multiplication”
  - closed under these operations
  - associative and distributive laws hold
  - additive and multiplicative identity elements
  - additive inverse for every element
  - multiplicative inverse for every non-zero element

- Example fields:
  - set of rational numbers
  - set of real numbers
  - set of integers is not a field
- Finite fields are called Galois fields.
- Example:
  - Binary numbers 0,1 with XOR as “addition” and AND as “multiplication”.
  - Called GF(2).

---

Galois Fields - The theory behind LFSRs

- Consider polynomials whose coefficients come from GF(2).
- Each term of the form $x^n$ is either present or absent.
- Examples: 0, 1, $x, x^2, x^7 + x^6 + 1$
- With addition and multiplication these form a field:
- “Add”: XOR each element individually with no carry:
  $$\begin{array}{c}
x^4 + x^3 + x + 1 \\
+ x^4 + x^2 + x \\
\hline
x^3 + x^2 + 1
\end{array}$$
- “Multiply”: multiplying by $x^n$ is like shifting to the left.
  $$\begin{array}{c}
x^3 + x + 1 \\
\times x + 1 \\
\hline
x^3 + 1
\end{array}$$
Galois Fields - The theory behind LFSRs

- These polynomials form a Galois (finite) field if we take the results of this multiplication modulo a prime polynomial \( p(x) \).
  - A prime polynomial is one that cannot be written as the product of two non-trivial polynomials \( q(x)r(x) \).
  - Perform modulo operation by subtracting a (polynomial) multiple of \( p(x) \) from the result. If the multiple is 1 this corresponds to XOR-ing the result with \( p(x) \).
- For any degree, there exists at least one prime polynomial.
- With it we can form \( GF(2^n) \).

- Additionally, …
- Every Galois field has a primitive element, \( \alpha \), such that all non-zero elements of the field can be expressed as a power of \( \alpha \). By raising \( \alpha \) to powers (modulo \( p(x) \)), all non-zero field elements can be formed.
- Certain choices of \( p(x) \) make the simple polynomial \( x \) the primitive element. These polynomials are called primitive, and one exists for every degree.
- For example, \( x^4 + x + 1 \) is primitive. So \( \alpha = x \) is a primitive element and successive powers of \( \alpha \) will generate all non-zero elements of \( GF(16) \). Example on next slide.

\[
\begin{align*}
\alpha^0 &= 1 \\
\alpha^1 &= x \\
\alpha^2 &= x^2 \\
\alpha^3 &= x^3 \\
\alpha^4 &= x + 1 \\
\alpha^5 &= x^2 + x \\
\alpha^6 &= x^3 + x \\
\alpha^7 &= x^4 + x + 1 \\
\alpha^8 &= x^2 + 1 \\
\alpha^9 &= x^3 + x \\
\alpha^{10} &= x^4 + x + 1 \\
\alpha^{11} &= x^2 + x \\
\alpha^{12} &= x^3 + x^2 + x + 1 \\
\alpha^{13} &= x^4 + x^2 + 1 \\
\alpha^{14} &= x^3 + 1 \\
\alpha^{15} &= 1
\end{align*}
\]

- Note this pattern of coefficients matches the bits from our 4-bit LFSR example.

- In general finding primitive polynomials is difficult. Most people just look them up in a table, such as:
Primitive Polynomials

\begin{align*}
x^2 + x + 1 & \quad x^{12} + x^6 + x^4 + x + 1 \quad x^{22} + x + 1 \\
x^3 + x + 1 & \quad x^{13} + x^4 + x^3 + x + 1 \quad x^{23} + x^5 + 1 \\
x^4 + x + 1 & \quad x^{14} + x^{10} + x^6 + x + 1 \quad x^{24} + x^7 + x^2 + x + 1 \\
x^5 + x^2 + 1 & \quad x^{15} + x + 1 \quad x^{25} + x^3 + 1 \\
x^6 + x + 1 & \quad x^{16} + x^{12} + x^3 + x + 1 \quad x^{26} + x^6 + x^2 + x + 1 \\
x^7 + x^3 + 1 & \quad x^{17} + x^3 + 1 \quad x^{27} + x^5 + x^2 + x + 1 \\
x^8 + x^4 + x^3 + x^2 + 1 & \quad x^{18} + x^7 + 1 \quad x^{28} + x^3 + 1 \\
x^9 + x^4 + 1 & \quad x^{19} + x^5 + x^2 + x + 1 \quad x^{29} + x + 1 \\
x^{10} + x^3 + 1 & \quad x^{20} + x^3 + 1 \quad x^{30} + x^6 + x^4 + x + 1 \\
x^{11} + x^2 + 1 & \quad x^{21} + x^2 + 1 \quad x^{31} + x^3 + 1 \\
\end{align*}

Galois Field

- Multiplication by \( x \) \( \Leftrightarrow \) shift left
- Taking the result mod \( p(x) \) \( \Leftrightarrow \) XOR-ing with the coefficients of \( p(x) \) when the most significant coefficient is 1.
- Obtaining all \( 2^{n-1} \) non-zero \( \Leftrightarrow \) Shifting and XOR-ing \( 2^{n-1} \) times.

Obtaining all \( 2^{n-1} \) non-zero \( \Leftrightarrow \) Shifting and XOR-ing \( 2^{n-1} \) times.

for \( k = 1, \ldots, 2^{n-1} \)

Building an LFSR from a Primitive Polynomial

- For \( k \)-bit LFSR number the flip-flops with FF1 on the right.
- The feedback path comes from the Q output of the leftmost FF.
- Find the primitive polynomial of the form \( x^k + \ldots + 1 \).
- The \( x^0 = 1 \) term corresponds to connecting the feedback directly to the D input of FF1.
- Each term of the form \( x^n \) corresponds to connecting an xor between FF \( n \) and \( n+1 \).
- 4-bit example, uses \( x^4 + x + 1 \)
  \begin{itemize}
  \item \( x^4 \Leftrightarrow \) FF4’s Q output
  \item \( x \Leftrightarrow \) xor between FF1 and FF2
  \item \( 1 \Leftrightarrow \) FF1’s D input
  \end{itemize}
- To build an 8-bit LFSR, use the primitive polynomial \( x^8 + x^4 + x^3 + x^2 + 1 \) and connect xors between FF2 and FF3, FF3 and FF4, and FF4 and FF5.
Error Correction with LFSRs

• XOR Q4 with incoming bit sequence. Now values of shift-register don’t follow a fixed pattern. Dependent on input sequence.
• Look at the value of the register after 15 cycles: “1010”
• Note the length of the input sequence is $2^4-1 = 15$ (same as the number of different nonzero patterns for the original LFSR)
• Binary message occupies only 11 bits, the remaining 4 bits are “0000”.
  – They would be replaced by the final result of our LFSR: “1010”
  – If we run the sequence back through the LFSR with the replaced bits, we would get “0000” for the final result.
  – 4 parity bits, “neutralize” the sequence with respect to the LFSR.
  
  \[
  \begin{array}{c}
  110010001110000 \\
  1100100011110100 \\
  \end{array} \Rightarrow \begin{array}{c}
  1010 \\
  0000 \\
  \end{array}
  \]
• If parity bits not all zero, an error occurred in transmission.
• If number of parity bits = log total number of bits, then single bit errors can be corrected.
• Using more parity bits allows more errors to be detected.
• Ethernet uses 32 parity bits per frame (packet) with 16-bit LFSR.