Today’s Lecture

- Codes for storage and communication
  - Cyclic codes
  - Reed-Solomon codes
- Arithmetic codes
- Self-checking logic
Cyclic codes are parity check codes with additional property that cyclic shift of codeword is also a codeword:
- If \((C_{n-1}, C_{n-2}, ..., C_1, C_0)\) is a codeword, \((C_{n-2}, C_{n-3}, ..., C_0, C_{n-1})\) is also a codeword.

Cyclic codes are used in:
- Sequential storage devices, e.g., tapes, disks, and data links
- Communication applications

An \((n,k)\) cyclic code can detect single bit errors, multiple adjacent bit errors affecting fewer than \((n-k)\) bits, and burst transient errors.

Cyclic codes require less hardware, in form of linear feedback shift registers:
- Parity check codes require complex encoding, decoding circuit using arrays of EX-OR gates, AND gates, etc.

Cyclic codes depend on the representation of data by a polynomial:
- If \((C_{n-1}, C_{n-2}, ..., C_1, C_0)\) is a codeword, its polynomial representation is \(C(x) = C_{n-1}x^{n-1} + C_{n-2}x^{n-2} + ... + C_1x + C_0\).

Cyclic codes are characterized by their generator polynomial \(g(x)\):
- \(g(x)\) is a polynomial of degree \((n-k)\) for an \((n,k)\) code, with a unity coefficient in \((n-k)\) term.
- \(g(x)\) is a factor of \(x^n - 1\), i.e., it divides it with zero remainder:
  - If a polynomial with degree \(n-k\) divides \(x^n - 1\), then \(g(x)\) generates a cyclic code.

Example: For \((7,4)\) code, \(g(x) = x^3 + x + 1\).
Cyclic Redundancy Check (CRC)

- Considers dataword and codeword to be polynomials
  - E.g., $i_0, i_1, i_2, \ldots, i_{n-1} \rightarrow i_0 + i_1X + i_2X^2 + \ldots + i_{n-1}X^{n-1}$
- Codeword = Dataword * Generator
  - $C(X) = D(X) \ast g(X)$
  - $g(X)$ is pre-defined CRC polynomial
    - depends on particular code
  - Additions performed during multiplication are mod2
    - $0+0 = 0$, $0+1 = 1+0 = 1$, $1+1 = 0$
- At receiver, divide n-bit codeword by CRC polynomial
  - $D(X) = C(X) / g(X)$
  - If remainder is non-zero, we’ve detected an error

Basic Operations on Polynomials

- Can multiply or divide one polynomial by another, follow modulo 2 arithmetic, coefficients are 1 or 0, and addition and subtraction are same

**Multiplication**

$(x^4 + x^3 + x^2 + 1)(x^3 + x)$

\[
x^7 + x^6 + x^5 + x^4 + x^2 + x
\]

**Division**

$x^4 + x^3 + x^2 + 1$

\[
x^5 + x^4 + x^3 + x^2 + x
\]

\[
x^3 + x
\]

Quotient

Remainder
Cyclic Code - Example

- Consider generator polynomial \( g(x) = x^3 + x + 1 \) for (7,4) code
- Can verify \( g(x) \) divides \( x^7 - 1 \)
- Given data word (1111), generate codeword
  - \( d(x) = x^3 + x^2 + x + 1 \)
  - Then \( c(x) = g(x)d(x) = (x^3 + x^2 + x + 1)(x^3 + x + 1) \)
  - \( = x^6 + x^5 + x^3 + 1 \)
- Hence code word is (1101001)

CRC Properties and Varieties

- An n-bit CRC check can detect all errors of less than n bits and all but 1 in \( 2^n \) multi-bit errors
- Examples:
  - CRC-12: \( g(X) = X^{12} + X^{11} + X^3 + X^2 + X + 1 \)
  - CRC-16: \( g(X) = X^{16} + X^{15} + X^2 + 1 \)
  - Ethernet uses CRC-32
    - More bits → better error detection capability
Circuit to Generate Cyclic Code

- Consider blocks labeled X as multipliers, and addition elements as modulo 2

![Circuit Diagram]

- g(x) = x^3 + x + 1
- d(x) = v(x)

- Another representation is to replace multipliers by storage elements, adders by EX-OR gates

Generation of Code Words

<table>
<thead>
<tr>
<th>Cyclic codes for 4-bit information words.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information</td>
</tr>
<tr>
<td>(d_3, d_2, d_1, d_0)</td>
</tr>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>1001</td>
</tr>
<tr>
<td>1010</td>
</tr>
<tr>
<td>1011</td>
</tr>
<tr>
<td>1100</td>
</tr>
<tr>
<td>1101</td>
</tr>
<tr>
<td>1110</td>
</tr>
<tr>
<td>1111</td>
</tr>
</tbody>
</table>

- Data polynomial = d_0 + d_1x + d_2x^2 + d_3x^3
- Generator polynomial = 1 + x + x^3
- Code polynomial = v_0 + v_1x + v_2x^2 + v_3x^3 + v_4x^4 + v_5x^5 + v_6x^6

![Code Generation Diagram]
Decoding of Cyclic Codes

- Determine if code word \((r_{n-1}, r_{n-2}, \ldots, r_1, r_0)\) is valid
- Code polynomial \(r(x) = r_{n-1}x^{n-1} + r_{n-2}x^{n-2} + \ldots + r_1x + r_0\)
- If \(r(x)\) is a valid code polynomial, it should be a multiple generator polynomial \(g(x)\)
- \(r(x) = d(x)g(x) + s(x)\), where \(s(x)\) the syndrome polynomial should be zero
- Hence, divide \(r(x)\) by \(g(x)\) and check the remainder whether equal to 0

Circuits for Decoding

\[
\begin{align*}
  b(x) &= (x^3 + x) d(x) \\
  v(x) + b(x) &= d(x) \\
  v(x) &= (x^3 + x + 1) d(x)
\end{align*}
\]

Hence, \(d(x) = v(x) / (x^3 + x + 1)\)

Another representation is to replace multipliers by storage elements and adders by EX-OR gates

Note: Once the division is completed, the registers contain the value of the syndrome (remainder)
Systematic Cyclic Codes

- Previous cyclic codes were not systematic, i.e. data not part of code word
- To generate \((n,k)\) systematic cyclic code, do the following:
  - Multiply \(d(x)\) by \(x^{n-k}\), this is accomplished by shifting \(d(x)\) \(n-k\) bits
  - The code polynomial is \(c(x) = r(x) + x^{n-k} d(x)\)
  - Hence \(x^{n-k} d(x) + r(x) = g(x)q(x)\), which is code word \(c(x)\) since it is a multiple of \(g(x)\)
Example of Systematic Cyclic Code

- Generator polynomial \( g(x) = x^4 + x^3 + x^2 + 1 \) of \((7,3)\) code
- Data is 3 bits, \( n-k = 4 \) bits

<table>
<thead>
<tr>
<th>Message Bits</th>
<th>( x^3M(x) )</th>
<th>Code Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>001</td>
<td>( x^3 )</td>
<td>00111011</td>
</tr>
<tr>
<td>010</td>
<td>( x^2 )</td>
<td>01001111</td>
</tr>
<tr>
<td>011</td>
<td>( x^3 + x^2 )</td>
<td>01110100</td>
</tr>
<tr>
<td>100</td>
<td>( x^5 )</td>
<td>10011100</td>
</tr>
<tr>
<td>101</td>
<td>( x^4 + x^2 )</td>
<td>10100111</td>
</tr>
<tr>
<td>110</td>
<td>( x^4 + x^5 )</td>
<td>11010011</td>
</tr>
<tr>
<td>111</td>
<td>( x^2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>

髹 | \( d(x) \) | \( n^4 \) |

Reed-Solomon Codes

- Popular ECC for CDs, DVDs, wireless communications, etc.
- \( k \) data **symbols**, each of which is \( s \) bits
- \( r \) parity symbols, each of which is also \( s \) bits
- Can correct up to \( r/2 \) symbols that contain errors
  - Or can correct up to \( r \) symbol **erasures**
  - Erasure = error in a known symbol
- Denoted by \( RS(n,k) \)
- Common example: \( RS(255, 223) \) with \( s=8 \)
  - \( n = 255 \rightarrow 255 \) codeword bytes
  - \( k = 223 \rightarrow 223 \) dataword bytes
  - \( r = 32 \rightarrow \) can correct errors in \( \leq 16 \) bytes
Reed-Solomon Codes,

- There exist many flavors of RS codes, each of which is tailored to specific purpose
  - Cross-Interleaved Reed-Solomon Coding (CIRC) used in CDs can correct error burst of up to 4000 bits!
  - 4000 bits is roughly equivalent to 2.5mm on the CD surface
- RS codes are best for bursty error model
  - Just as good at handling 1 error in symbol or $s$ errors in symbol
- Codewords created by multiplying datawords with generator polynomial (like CRC)

Checksum Codes - Basic Concepts

- The checksum is appended to block data when such blocks are transferred

\[ d_0 \\ d_1 \\ d_2 \\ d_3 \]

\[ r_0 \\ r_1 \]

Transfer

received word of data

\[ r_i \]

Compare

Received Version of Checksum

\[ d_i \] = original word of data
### Single Precision Checksums

A single-precision checksum is formed by adding the data words and ignoring any overflow. The single-precision checksum is unable to detect certain types of errors. The received checksum and the checksum of the received data are equal, so no error is detected.

### Double Precision Checksums

- Compute 2n-bit checksum for a block of n-bit words
- Overflow is still a concern, but it is now overflow from a 2n-bits
Honeywell Checksums

- Concatenate consecutive words to form double words to create \( \frac{k}{2} \) words of \( 2n \) bits; checksum formed over newly structured data.

Residue Checksums

- The same concept as the single-precision checksum except that the carry bit is not ignored and is added to checksum in an end-around carry fashion.
**Arithmetic Codes**

- Useful to check arithmetic operations
- Parity codes are not preserved under addition, subtraction
- Arithmetic codes can be
  - Separate: check symbols disjoint from data symbols
  - Non-separate: combined check and data
- Several Arithmetic codes
  - AN codes, Residue codes, Bi-residue codes
- Arithmetic codes have been used in STAR fault tolerant computer for space applications

**AN codes**

- Data X is multiplied by check base A to form A.X
- Addition of code words performed modulo M where A divides M
- A(X + _M Y) = AX + _M AY
- Check operation by dividing the result by A
- If result = 0, no error, else error

![AN Codes Diagram](image-url)
Example of 3N Code

<table>
<thead>
<tr>
<th>Original Information</th>
<th>3N Code word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000000</td>
</tr>
<tr>
<td>0001</td>
<td>0000111</td>
</tr>
<tr>
<td>0010</td>
<td>0001010</td>
</tr>
<tr>
<td>0011</td>
<td>0001001</td>
</tr>
<tr>
<td>0100</td>
<td>0010100</td>
</tr>
<tr>
<td>0101</td>
<td>0011110</td>
</tr>
<tr>
<td>0110</td>
<td>0100100</td>
</tr>
<tr>
<td>0111</td>
<td>0101010</td>
</tr>
<tr>
<td>1000</td>
<td>0110000</td>
</tr>
<tr>
<td>1001</td>
<td>0110111</td>
</tr>
<tr>
<td>1010</td>
<td>1001110</td>
</tr>
<tr>
<td>1011</td>
<td>1000001</td>
</tr>
<tr>
<td>1100</td>
<td>1010000</td>
</tr>
<tr>
<td>1101</td>
<td>1011110</td>
</tr>
<tr>
<td>1110</td>
<td>1001010</td>
</tr>
<tr>
<td>1111</td>
<td>1011011</td>
</tr>
</tbody>
</table>

Illustration of the error detection capabilities of the 3N arithmetic code. The presence of the fault results in the sum being an invalid 3N code.

Residue Codes

- Separate code \((X, X \text{ Mod } A)\)
- Created by appending the residue of a number to that number
Berger Codes

- Used in Control units as systematic codes
- The k check bits are the binary encoding of the number of zeros in the d-bit dataword
  - Berger codes are formed by appending $k = \lceil \log_2 (d+1) \rceil$ check bits and $n = d + k$
- Example:
  - $X=1\ 0\ 0\ 1\ 0\ 0\ 0\ 1$ => $k = \lceil \log_2 (8+1) \rceil = 4$
  - the number of 1s in this data is 3 (0011)
  - the complement of (0011) is (1100)
  - the resulting code word is: 1001 0001 1100

Berger Codes

- Can detect all single-bit errors and all unidirectional multi-bit errors
  - Unidirectional: all bit errors are either from 0→1 or from 1→0
- Good for detecting coupling faults
  - Change in one bit erroneously causes change(s) in other bit(s)
  - Models short circuits (including bridging faults)
Self-Checking Circuits

- What properties/invariants can we build into circuits such that codeword inputs do not lead to codeword outputs in the presence of faults?

- Self-testing circuit
  - for every fault from a prescribed set there exists at least one valid input code word that will produce an invalid output code word when a single fault is present in the circuit

- Fault secure circuit
  - any single fault from a prescribed set results in the circuit either producing the correct code word or producing a non-code word, for any valid input code word

- Totally self-checking circuit (TSC)
  - the circuit is both fault secure and self-testing
  - all single faults are detectable by at least one valid code word input, and when a given input combination does not detect the fault, the output is the correct code word output

Circuit of Basic TSC Comparison Element

1. Dual-rail signal coded so two bits are complementary.
2. Comparison element checks for the equality of the two dual-rail signals at its inputs.
3. Outputs a dual-rail signal 01 or 10 if both inputs are equal and properly coded; otherwise, outputs 00 or 11.
Implementing EDC/ECC in Hardware

- Where does EDC/ECC get used?
  - Disk, CD-ROM
  - Memory (DRAM, SRAM)
  - Buses
  - Network
- Tradeoff between EDC and ECC
  - ECC: Forward error recovery
    - Often on critical path, so can slow down even fault-free system
  - EDC: Backward error recovery
    - Detecting error leads to recovery (can be slow)
- So would you use ECC or EDC in your L1 cache?
  - How about in DRAM?