Error Control Coding

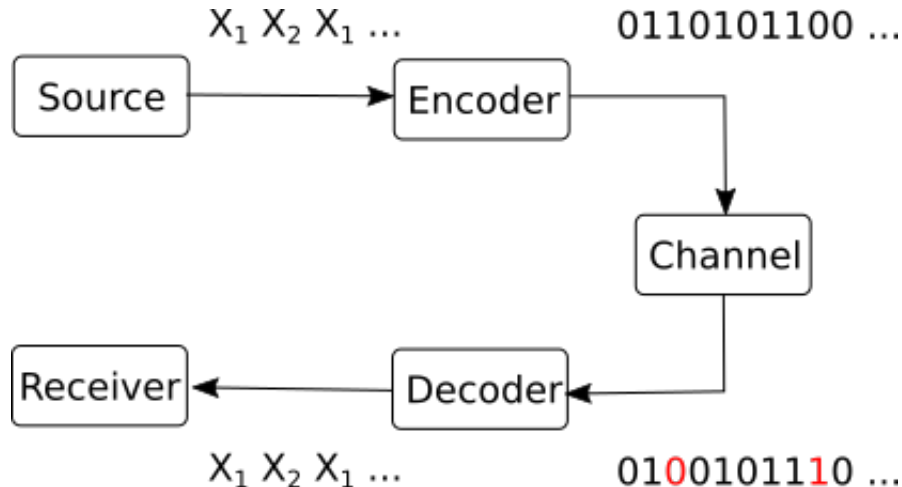# Linear block codes

What is error control coding?

Figure 1: Communication system

- The second main task of coding: **error control**

Mutual information and error control

- Mutual information $I(X, Y)$ = the information transmitted on the channel
- Even though there is information transmitted on the channel, when the channel is noisy the information is **not enough**

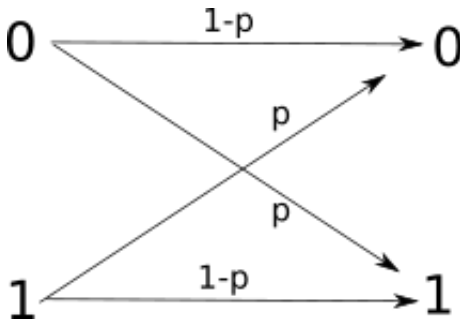Example: consider the following BSC channel ($p = 0.01$, $p(x_1) = 0.5$, $p(x_2) = 0.5$):

Figure 2: Binary symmetric channel (BSC)

Why is error control needed?

- In most communications it is required that *all* bits are received correctly
  - Not 1% errors, not 0.1%, not 0.0001%. **None!**
- But that is not possible unless the channel is ideal.
- So what do to? **Error control coding**

Modelling the errors on the channel

- We consider only binary channels (symbols = $\{0, 1\}$)
- An error = a bit is changed from 0 to 1 or vice-versa
- Changing the value of a bit = modulo-2 sum with 1
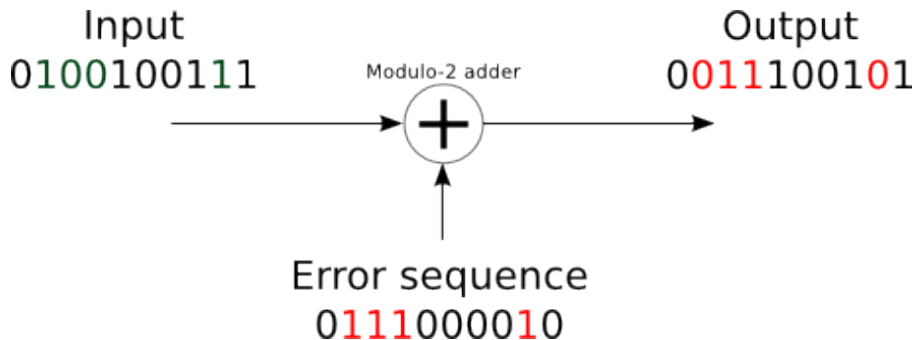- Value of a bit remains the same = modulo-2 sum with 0

Figure 3: Channel error model

Error detection and error correction

Binary error correction:

- ► For binary channels, know the location of error $=>$ fix error by inverting bit
- ► Locating error $=$ correcting error

Two possibilities in practice:

- ► **Error detection**: find out if there is any error in the received sequence
    - ► don't know exactly where, so cannot correct the bits, but can discard whole sequence
    - ► perhaps ask the sender to retransmit (examples: TCP/IP, internet communication etc)
    - ► easier to do
- ► **Error correction**: find out exactly which bits have errors, if any
    - ► can correct all errored bits by inverting them
    - ► useful when can't retransmit (data is stored: on HDD, AudioCD etc.)
    - ► harder to do than mere detection

What is error control coding?

The process of error control:

1. Want to send a sequence of k bits = **information word**

$$\mathbf{i} = i_1 i_2 ... i_k$$

2. For each possible information word, the coder assigns a **codeword** of length $n > k$:

$$\mathbf{c} = c_1 c_2 ... c_n$$

3. The codeword is sent on the channel instead of the original information word

4. The receiver receives a sequence $\mathbf{r} = \mathbf{c} + \mathbf{e}$, with possible errors:

$$\mathbf{r} = r_1 r_2 ... r_n$$

5. The decoding algorithm detects/corrects the errors in $\mathbf{r}$

Definitions

- An **error correcting code** is an association between the set of all possible information words to a set of codewords
  - Each possible information word **i** has a certain codeword **c**
- The association can be done:
  - randomly: codewords are selected and associated randomly to the information words
  - based on a certain rule: the codeword is computed with some algorithm from the information word
- A code is a **block code** if it operates with words of *fixed size*
  - Size of information word $\mathbf{i} = k$, size of codeword $\mathbf{c} = n$, $n > k$
  - Otherwise it is a *non-block code*
- A code is **linear** if any linear combination of codewords is also a codeword
- The **coding rate** of a code is:

$$R = \frac{k}{n}$$

- A code C is an t-**error-detecting** code if it is able to *detect t* errors
- A code C is an t-**error-correcting** code if it is able to *correct t* errors

Hamming distance

- The **Hamming distance** of two binary sequences a, b of length $n =$ the total number of bit differences between them

$$d_H(a, b) = \sum_{i=1}^{N} a_i \bigoplus b_i$$

- We need at least $d_H(a, b)$ bit changes to convert one sequence into another
- It satisfies the 3 properties of a metric function:
    1. $d(a, b) \geq 0 \forall a, b$, with $d(a, b) = 0 \Leftrightarrow a = b$
    2. $d(a, b) = d(b, a), \forall a, b$
    3. $d(a, c) \leq d(a, b) + d(b, c), \forall a, b, c$
- The **minimum Hamming distance of a code**, $d_{Hmin} =$ the minimum Hamming distance between any two codewords $\mathbf{c_1}$ and $\mathbf{c_2}$
- Example at blackboard

Linear block codes

- A code is a **block code** if it operates with words of *fixed size*
  - Size of information word $\mathbf{i} = k$, size of codeword $\mathbf{c} = n$, $n > k$
  - Otherwise it is a *non-block code*
- A code is **linear** if any linear combination of codewords is also a codeword
- A code is called **systematic** if the codeword contains all the information bits explicitly, unaltered
  - coding merely adds supplementary bits besides the information bits
  - codeword has two parts: the information bits and the parity bits
  - example: parity bit added after the information bits
- Otherwise the code is called **non-systematic**
  - the information bits are not explicitly visible in the codeword
- Example: at blackboard

Generator matrix

- All codewords for a linear block code can be generated via a matrix multiplication:

$$\mathbf{i} \cdot [G] = \mathbf{c}$$

Figure 4: Codeword construction with generator matrix

- $[G]$ = **generator matrix** of size $k \times n$
- All operations are done in modulo-2 arithmetic:
  - $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$
  - multiplications as usual

Parity check matrix

- How to check if a binary word is a codeword or not
- Every $k \times n$ generator matrix $[G]$ has complementary matrix $[H]$ such that

$$0 = [H] \cdot [G]^T$$

- For every codeword $\mathbf{c}$ generated with $[G]$:

$$\boxed{0 = [H] \cdot \mathbf{c}^T}$$

- because:

$$\mathbf{i} \cdot [G] = \mathbf{c}$$
$$[G]^T \cdot \mathbf{i}^T = \mathbf{c}^T$$
$$[H] \cdot \mathbf{c}^T = [H] \cdot [G]^T \cdot \mathbf{i}^T = 0$$

Parity check matrix

- $[H]$ is the **parity-check matrix**, size $= (n - k) \times n$
- $[G]$ and $[H]$ are related, one can be deduced from the other
- The resulting vector $z = [H] \cdot [c]^T$ is the **syndrome**
- All codewords generated with $[G]$ will produce 0 when multiplied with $[H]$
- All binary sequences that are not codewords will produce $\neq 0$ when multiplied with $[H]$
- Column-wise interpretation of multiplication:

Z

H

r

[G] and [H] for systematic codes

- For systematic codes, [G] and [H] have special forms
- Generator matrix
    - first part = some matrix Q
    - second part = identity matrix

$$[G]_{k \times n} = [Q_{k \times (n-k)} \;\; I_{k \times k}]$$

- Parity-check matrix
    - first part = identity matrix
    - second part = same Q, transposed

$$[H]_{(n-k) \times n} = [I_{(n-k) \times (n-k)} \;\; Q^T_{(n-k) \times k}]$$

- Can easily compute one from the other
- Example at blackboard

Syndrome-based error detection

Syndrome-based error *detection* for linear block codes:

1. generate codewords with generator matrix:

$$\mathbf{i} \cdot [G] = \mathbf{c}$$

2. send codeword $\mathbf{c}$ on the channel
3. random error word $\mathbf{e}$ is applied on the channel
4. receive word $\mathbf{r} = \mathbf{c} \oplus \mathbf{e}$
5. compute **syndrome** of $\mathbf{r}$:

$$\mathbf{z} = [H] \cdot \mathbf{r}^T$$

6. Decide:
   - If $\mathbf{z} = 0 \Longrightarrow \mathbf{r}$ has no errors
   - If $\mathbf{z} \neq 0 \Longrightarrow \mathbf{r}$ has errors

Syndrome-based error correction

Syndrome-based error *correction* for linear block codes:

- $\mathbf{z} \neq 0 \implies \mathbf{r}$ has errors, we need to locate them
- The syndrome is the effect only of the error word:

$$\mathbf{z} = [H] \cdot \mathbf{r}^T = [H] \cdot (\mathbf{c}^T \oplus \mathbf{e}^T) = [H] \cdot \mathbf{e}^T$$

7. Create a **syndrome lookup table**:
   - for every possible error word $\mathbf{e}$, compute the syndrome $\mathbf{z} = [H] \cdot \mathbf{e}^T$
   - start with error words with 1 error (most likely), then with 2 errors (less likely), and so on

8. Locate the syndrome $\mathbf{z}$ in the table, read the corresponding error word $\widehat{\mathbf{e}}$

9. Find the correct word:
   - adding the error word again will invert the errored bits back to the originals

$$\widehat{\mathbf{c}} = \mathbf{r} \oplus \widehat{\mathbf{e}}$$

Summing up

Summing up until now:

- Linear block codes use a generator matrix G for encoding, and a parity-check matrix H for checking the received word.

Hamming codes

- A particular class of linear error-correcting codes
- Definition: a **Hamming code** is a linear block code where the columns of $[H]$ are *the binary representation of all numbers from 1 to $2^r - 1$, $\forall r \geq 2$*
- Example (blackboard): (7,4) Hamming code
- Systematic: arrange the bits in the codeword, such that the control bits correspond to the columns having a single 1
  - no big difference from the usual systematic case, just a rearrangement of bits
  - makes implementation easier
- Example codeword for Hamming(7,4):

$$c_1 \, c_2 \, i_3 \, c_4 \, i_5 \, i_6 \, i_7$$

Properties of Hamming codes

- From definition of $[H]$ it follows:
  1. Codeword has length $n = 2^r - 1$
  2. r bits are parity bits (also known as **control bits**)
  3. $k = 2^r - r - 1$ bits are information bits

- Notation: **(n,k) Hamming code**
  - n = codeword length = $2^r - 1$,
  - k = number of information bits = $2^r - r - 1$
  - Example: (7,4) Hamming code, (15,11) Hamming code, (127, 120) Hamming code

Properties of Hamming codes

- Can detect two errors
    - All columns are different $=>$ can detect 2 errors
    - Sum of two columns equal to a third $=>$ cannot correct 3

OR

- Can correct one error
    - All columns are different $=>$ can correct 1 error
    - Sum of two columns equal to a third $=>$ cannot correct 2
    - Non-systematic: syndrome $=$ error position

BUT * Not simultaneously! * same non-zero syndrome can be obtained with 1 or 2 errors, can't distinguish

Coding rate of Hamming codes

Coding rate of a Hamming code:

$$R = \frac{k}{n} = \frac{2^r - r - 1}{2^r - 1}$$

The Hamming codes can correct 1 OR detect 2 errors in a codeword of size n * (7,4) Hamming code: $n = 7$ * (15,11) Hamming code: $n = 15$ * (31,26) Hamming code: $n = 31$

Longer Hamming codes are progressively weaker: * weaker error correction capability * better efficiency (higher coding rate) * more appropriate for smaller error probabilities

```
%matplotlib inline
import numpy as np, matplotlib.pyplot as plt
r = np.array([3., 4., 5., 6., 7., 8., 9., 10.])
k = 2**r - r - 1
n = 2**r-1
R = k/n
plt.plot(r, R)
plt.xticks(r, [str(int(i)) for i in n])
```

See whiteboard.

In this example, encoding is done without the generator matrix G, directly with the matrix H, by finding the values of the parity bits c_1, c_2, c_4 such that

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = [H] \begin{bmatrix} c_1 \\ c_2 \\ i_3 \\ c_4 \\ i_5 \\ i_6 \\ i_7 \end{bmatrix}$$

For a single error, the syndrome **is the binary representation of the location of the error**.

At whiteboard, using **shift registers**.

Hamming codes can correct 1 error OR can detect 2 errors, but we cannot differentiate the two cases:

Example: * the syndrome $\mathbf{z} = [H] \cdot \mathbf{r}^T = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ can be caused by: * a single error in location 3 (bit i_3) * two errors in location 1 and 2 (bits c_1, bits c_2)

- ▶ if we know it is a single error, we can go ahead and correct it, then use the corrected data
- ▶ if we know there are two errors, we should NOT attempt to correct them, because we cannot locate the errors correctly

Unfortunately, it is **not possible to differentiate** between the two cases. **Solution?** Add additional parity bit $\rightarrow$ SECDED Hamming codes

SECDED Hamming codes

- Add an additional parity bit to differentiate the two cases
  - c_0 = sum of all n bits of the codeword
- For (7,4) Hamming codes:

$$\mathbf{c_0}\, c_1\, c_2\, i_3\, c_4\, i_5\, i_6\, i_7$$

- The pairty check matrix is extended by 1 row and 1 column

$$\tilde{H} = \begin{bmatrix} 1 & 1 \\ 0 & \mathbf{H} \end{bmatrix}$$

- Known as SECDED Hamming codes
  - **S**ingle **E**rror **C**orrection - **D**ouble **E**rror **D**etection

Encoding and decoding of SECDED Hamming codes

Encoding

- compute codeword using $\tilde{H}$

Decoding

- Compute syndrome of the received word using $\tilde{H}$

$$\tilde{\mathbf{z}} = \begin{bmatrix} z_0 \\ \mathbf{z} \end{bmatrix} = [\tilde{H}] \cdot \mathbf{r}^T$$

- z_0 is an additional bit in the syndrome corresponding to c_0
- z_0 tells us whether the received c_0 matches the paroty of the received word
    - $z_0 = 0$: the additional parity bit matches the parity of the received word
    - $z_0 = 1$: the additional parity bit does not match the parity of the received word

Decide which of the following cases happened

Circuits for encoding and decoding SECDED Hamming codes

At whiteboard. Slight modification of the previous schematic to accommodate the extra parity bit.

Summary until now

- Systematic codes: information bits + parity bits
- Generator matrix: use to generate codeword

$$\mathbf{i} \cdot [G] = \mathbf{c}$$

- Parity-check matrix: use to check if a codeword

$$0 = [H] \cdot \mathbf{c}^T$$

- Syndrome:

$$\mathbf{z} = [H] \cdot \mathbf{r}^T$$

- Syndrome-based error detection: syndrome non-zero
- Syndrome-based error correction: lookup table
- Hamming codes: $[H]$ contains all numbers $1...2^r - 1$
- SECDED Hamming codes: add an extra parity bit

Cyclic codes

Definition of cyclic codes

Cyclic codes are a particular class of linear block codes for which *every cyclic shift of a codeword is also a codeword*

- Cyclic shift: cyclic rotation of a sequence of bits (any direction)
- Are a particular class of linear block codes, so all the theory up to now still applies
    - they have a generator matrix, parity check matrix etc.
- But they can be implemented more efficient than general linear block codes (e.g. Hamming)
- Used **everywhere** under the common name **CRC** (**C**yclic **R**edundancy **C**heck)
    - Network communications (Ethernet), data storage in Flash memory

Every binary sequence **a** corresponds to a polynomial **a(x)** with binary coefficients

$$a_0 a_1 ... a_{n-1} \rightarrow \mathbf{a(x)} = a_0 \oplus a_1 x \oplus ... \oplus a_{n-1} x^{n-1}$$

Example:
$$10010111 \rightarrow 1 \oplus x^3 \oplus x^5 \oplus x^6 \oplus x^7$$

From now on, by "codeword" we also mean the corresponding polynomial. Can perform all mathematical operations with these polynomials: * addition, multiplication, division etc. (examples)
There are efficient circuits for performing multiplications and divisions.

Theorem:

All the codewords of a cyclic code are multiples of a certain polynomial $g(x)$, known as **generator polynomial**.

Properties of generator polynomial $g(x)$:

- The generator polynomial has first and last coefficient equal to 1.
- The generator polynomial is a factor of $X^n \oplus 1$
- The *degree* of $g(x)$ is $n - k$, where:
    - The codeword = polynomial of degree $n - 1$ (n coefficients)
    - The information polynomial = polynomial of degree $k - 1$ (k coefficients)
    $$(k - 1) + (n - k) = n - 1$$
- **The degree of $g(x)$ is the number of parity bits of the code.**

Theorem:
If $g(x)$ is a polynomial of degree $(n - k)$ and is a factor of $X^n \oplus 1$, then $g(x)$ generates a (n, k) cyclic code.
Example:

$$1 \oplus x^7 = (1 \oplus x)(1 \oplus x + \oplus x^3)(1 \oplus x^2 \oplus x^3)$$

Each factor generates a code:

- $1 \oplus x$ generates a (7,6) cyclic code
- $1 \oplus x \oplus x^3$ generates a (7,4) cyclic code
- $1 \oplus x^2 \oplus x^3$ generates a (7,4) cyclic code

Computing the codewords

Start from **information polynomial** with k bits

$$i(x) = i_0 \oplus i_1 x \oplus ... \oplus i_{k-1} x^{k-1}$$

Non-systematic codeword generation:

- Codeword $= i(x) \cdot g(x)$

$$\boxed{c(x) = i(x) \cdot g(x)}$$

Systematic codeword generation:

$$\boxed{c(x) = b(x) \oplus x^{n-k} i(x)}$$

where $b(x)$ is the remainder of dividing $x^{n-k} i(x)$ to $g(x)$:

$$x^{n-k} i(x) = a(x) g(x) \oplus b(x)$$

- (Proof: at blackboard)

Proving the cyclic property

We prove that any cyclic shift of a codeword is also a codeword.
Proof: at whiteboard

- Original codeword

$$c_0 c_1 c_2 ... c_{n-1} \rightarrow \mathbf{c(x)} = c_0 \oplus c_1 x \oplus ... \oplus c_{n-1} x^{n-1}$$

- Cyclic shift to the right by 1 position

$$c_{n-1} c_0 c_1 ... c_{n-2} \rightarrow \mathbf{c'(x)} = c_{n-1} \oplus c_0 x \oplus ... \oplus c_{n-2} x^{n-1}$$

- Note that

$$\mathbf{c'(x)} = x \cdot \mathbf{c(x)} \oplus c_{n-1} x^n \oplus c_{n-1}$$
$$= x \cdot \mathbf{c(x)} \oplus c_{n-1}(x^n \oplus 1)$$

Since $\mathbf{c(x)}$ is a multiple of $g(x)$, so is $x \cdot \mathbf{c(x)}$. Also $(x^n \oplus 1)$ is always a multiple of $g(x)$. It follows that their sum $\mathbf{c'(x)}$ is a also a multiple of $g(x)$, which means it is a codeword.

Cyclic code encoder circuits

- Coding = based on polynomial multiplications and divisions
- Efficient circuits for multiplication / division exist, that can be used for systematic or non-systematic codeword generation (draw on blackboard)

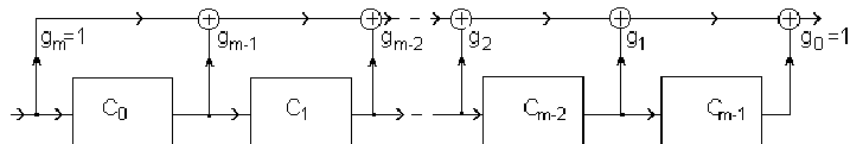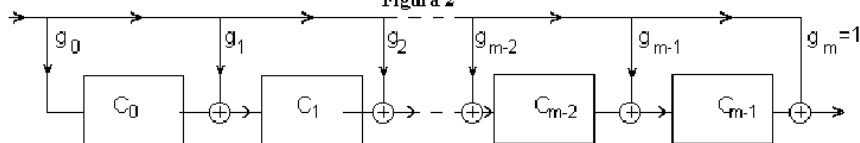# Circuits for multiplication of binary polynomials

Figura 2

Figura 3

Figure 7: Circuits for polynomial multiplication

Operation of multiplication circuits

- The input polynomial is applied at the input, 1 bit at a time, starting from highest degree
- The output polynomial is obtained at the output, 1 bit at a time, starting from highest degree
- Because output polynomial has larger degree, the circuit needs to operate a few more samples until the final result is obtained. During this time the input is 0.
- Examples: at the whiteboard

Linear analysis of multiplication circuits

- These circuits are **linear time-invariant systems** (remember Digital Signal Processing class?), because they are composed only of summations, multiplication by scalars, and delay blocks.
- Therefore, using the Z transform approach (to come soon in Digital Signal Processing class), the output can be computed based on the graph of the system:
  - Draw the graph of the system: cells become z^{-1} blocks, everything else is the same
  - Every z^{-1} block means a delay of one, which is what a cell does
  - Call the input polynomial is $\mathbf{X}(\mathbf{z})$
  - Call the output polynomial is $\mathbf{Y}(\mathbf{z})$
  - Every z^{-1} block means multiplying with z^{-1}
  - Compute the output $\mathbf{Y}(\mathbf{z})$ based on $\mathbf{X}(\mathbf{z})$, from the graph

We get:

$$Y(z) = X(z) \cdot G(z) \cdot z^{-m},$$

meaning that the **output polynomial = input polynomial * g(x) polynomial, with a delay of m bits (time samples)**.

The delay of m time samples is caused by the fact that the input polynomial has degree $(k-1)$, but the resulting polynomial has larger degree $(k-1) + m$, therefore we need to wait m more time samples until we get the full result.
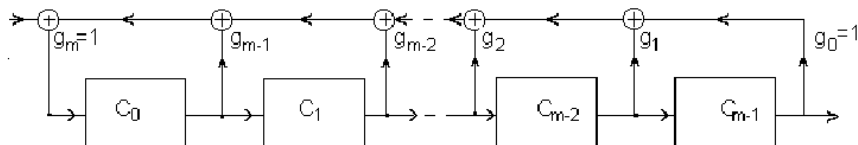
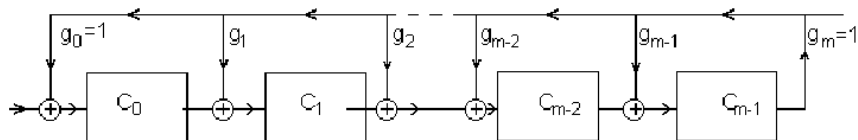# Circuits for division binary polynomials

Figura 4

Figura 5

Figure 8: Circuits for polynomial division

Operation of division circuits

- The input polynomial is applied at the input, 1 bit at a time, starting from highest degree
- The output polynomial is obtained at the output, 1 bit at a time, starting from highest degree
- Because output polynomial has smaller degree, the circuit first outputs some zero values, until starting to output the result.
- Examples: at the whiteboard

Linear analysis of division circuits

- These circuits are also **linear time-invariant systems**, because they are composed only of summations, multiplication by scalars, and delay blocks.
- Therefore, using the Z transform approach, the output can be computed based on the graph of the system:
  - Draw the graph of the system: cells become z^{-1} blocks, everything else is the same
  - Every z^{-1} block means a delay of one, which is what a cell does
  - Call the input polynomial is $\mathbf{X}(z)$
  - Call the output polynomial is $\mathbf{Y}(z)$
  - Every z^{-1} block means multiplying with z^{-1}
  - Compute the output $\mathbf{Y}(z)$ based on $\mathbf{X}(z)$, from the graph

We get:
$$Y(z) = \frac{X(z)}{G(z)}$$
meaning that the **output polynomial = input polynomial / g(x) polynomial**.

- Non-systematic cyclic encoder circuit:
  - simply a polynomial multiplication circuit
- A systematic cyclic encoder circuit:
  - more complicated
  - must analyze first Linear Feedback Shift Registers (LFSR)

Linear-Feedback Shift Registers (LFSR)

- A **flip-flop** = a cell holding a bit value (0 or 1)
  - called *"bistabil"* in Romanian
  - operates on the edges of a clock signal

- A **register** = a group of flip-flops, holding multiple bits
  - example: an 8-bit register

- A **shift register** = a register where the output of a flip-flop is connected to the input of the next one
  - the bit sequence is shifted to the right
  - has an input (for the first cell)

- A **linear feedback shift register** (LFSR) = a shift register for which the input is a computed as a linear combination of the flip-flops values
  - input = usually a XOR of some cells from the register
  - like a division circuit without any input
  - feedback = all flip-flops, with coefficients g_i in general
  - example at whiteboard

States and transitions of LFSR

- **State** of the LFSR = the sequence of bit values it holds at a certain moment
- The state at the next moment, $S(k + 1)$, can be computed by multiplication of the current state $S(k)$ with the **companion matrix** (or **transition matrix**) $[T]$:

$$S(k + 1) = [T] * S(k)$$

- The companion matrix is defined based on the feedback coefficients g_i:

$$T = \begin{bmatrix} 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ . & . & . & ... & . \\ 0 & 0 & 0 & ... & 1 \\ g_0 & g_1 & g_2 & \cdots & g_{m-1} \end{bmatrix}$$

- Note: reversing the order of bits in the state $->$ transposed matrix
- Starting at time 0, then the state at time k is:

$$S(k) = [T]^k S(0)$$

Period of LFSR

- The number of states is finite $->$ they must repeat at some moment
- The state equal to 0 must not be encountered (LFSR will remain 0 forever)
- The **period** of the LFSR = number of time moments until the state repeats
- If period is N, then state at time N is same as state at time 0:

$$S(N) = [T]^N S(0) = S(0),$$

  which means:

$$[T]^N = I_m$$

- Maximum period is $N_{max} = 2^m - 1$ (excluding state 0), in this case the polynomial $g(x)$ is called **primitive polynomial**

LFSR with inputs

- ▶ What if the LFSR has an input added to the feedback (XOR)?
  - ▶ example at whiteboard
  - ▶ assume the input is a sequence $a_{N-1}, ... a_0$
- ▶ Since a LFSR is a **linear circuit**, the effect is added:

$$S(1) = [T] \cdot S(0) + \begin{bmatrix} 0 \\ 0 \\ ... \\ a_{N-1} \end{bmatrix}$$

- ▶ In general

$$S(k_1) = [T] \cdot S(k) + a_{N-k} \cdot [U],$$

where $[U]$ is:

$$[U] = \begin{bmatrix} 0 \\ 0 \\ ... \\ 1 \end{bmatrix}$$

Systematic cyclic encoder circuit

- ▶ Draw on whiteboard only (sorry!)
- ▶ Initially the LFSR state is 0 (all cells are 0)
- ▶ Switch in position I:
  - ▶ information bits applied to the output and to the division circuit
  - ▶ first bits = information bits, systematic, OK
  - ▶ LFSR with feedback and input, input = information bits
- ▶ Switch in position II:
  - ▶ LFSR with feedback and input, input = feedback
  - ▶ output bits are also applied to the input of the division circuit
- ▶ In the end all cells end up in 0, so ready for next encoding
  - ▶ because the input and feedback cancel each other (are identical)

Systematic cyclic encoder circuit

- Why is the result the desired codeword?
- The output polynomial $c(x)$:
    1. has the information bits in the first part (systematic)
    2. is a multiple of $g(x)$ ==> therefore it is the systematic codeword for the information bits
- the output $c(x)$ is a multiple of $g(x)$ because:
    - the output is always applied also t the input of the division circuit
    - after division, the cells end up in 0 <=> no remainder <=> so $c(x)$ is a multiple $g(x)$
- Side note: we haven't really explained *why* the constructed output $c(x)$ is a codeword, but we *proved* that it is so, and this is enough

The parity-check matrix for systematic cyclic codes

- Cyclic codes are linear block codes, so they have a parity-check and a generator matrix
  - but it is more efficient to implement them with polynomial multiplication / division circuits
- The parity-check matrix $[H]$ can be deduced by analyzing the states of the LFSR
  - it is a LFSR with feedback and input
  - the input is the codeword $c(x)$
  - do computations at whiteboard ...
  - ... arrive at expression for matrix $[H]$

The parity-check matrix for systematic cyclic codes

- The parity check matrix $[H]$ has the form

$$[H] = [U, TU, T^2U, ... T^{n-1}U]$$

- The cyclic codeword satisfies the usual relation

$$S(n) = 0 = [H]\mathbf{c^T}$$

- In case of error, the state at time n will be the syndrome (non-zero):

$$S(n) = [H]\mathbf{r^T} \neq 0$$

- Implement a 1-error-correcting cyclic decoder using LFSRs
- Draw schematic at whiteboard only (sorry!)
- Contents of schematic:
    - main shift register MSR
    - main switch SW
    - 2 LFSRs (divider circuits) after $g(x)$
    - 2 error locator blocks, one for each divider
    - 2 validation gates V1, V2, for each divider
    - output XOR gate for correcting errors

Cyclic decoder implemented with LFSR

- Operation phases:

1. Input phase: SW on position I, validation gate V1 blocked

   - The received codeword $r(x)$ is received one by one, starting with largest power of x^n
   - The received codeword enters the MSR and first LFSR (divider)
   - The first divider computes $r(x) : g(x)$
   - The validation gate V1 is blocked, no output

- Input phase ends after n moments, the switch SW goes into position II
- If the received word has no errors, all LFSR cells are 0 (no remainder), will remain 0, the error locator will always output 0

Cyclic decoder implemented with LFSR

2. Decoding phase: SW on position II, validation gate V1 open
   - LFSR keeps running with no input for n more moments
   - the MSR provides the received bits at the output, one by one
   - **exactly when the erroneous bit is at the main output of MSR, the error locator will output 1, and the output XOR gate will correct the bit (TO BE PROVEN)**
   - during this time the next codeword is loaded into MSR and into second LFSR (input phase for second LFSR)

- After n moments, the received word is fully decoded and corrected
- SW goes back into position I, the second LFSR starts decoding phase, while the first LFSR is loading the new receiver word, and so on
- **To prove:** error locator outputs 1 exactly when the erroneous bit is at the main output

Cyclic decoder implemented with LFSR

Theorem: if the k-th bit r_{n-k} from $r(x)$ has an error, the error locator will output 1 exactly after $k-1$ moments

- The k-th bit will be output from MSR after $k-1$ moments, i.e. exactly when the error locator will output 1 $->$ will correct it
- **Proof:**
    1. assume error on position r_{n-k}
    2. the state of the LFSR at end of phase I = syndrome = column $(n-k)$ from $[H]$
       $$S(n) = [H]\mathbf{r}^T = [H]\mathbf{e}^T = T^{n-k}U$$
    3. after another $k-1$ moments, the state will be
       $$T^{k-1}T^{n-k}U = T^{n-1}U$$
    4. since $T^n = I_n$ $->$ $T^{n-1} = T^{-1}$
    5. $T^{-1}U$ is the state preceding state U, which is state
       $$\begin{bmatrix} 1 \\ 0 \\ ... \\ 0 \end{bmatrix}$$

Cyclic decoder implemented with LFSR

- Step 5 above can be shown in two ways:
  - reasoning on the circuit
  - using the definition of T^{-1}

$$T = \begin{bmatrix} g_1 & g_2 & \cdots g_{m-1} & 1 & \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

- The error locator is designed to detect this state $T^{-1}U$, i.e. it is designed as shown
- Therefore, the error locator will correct an error
- This works only for 1 error, due to proof (1 column from [H])

Thresholding cyclic decoder

- ► A different variant of cyclic decoder
- ► Consider the parity check matrix $[H]$ of the cyclic code
- ► Perform **elementary transformations** on $[H]$ to obtain a **reduced** matrix $[H_R]$ such that:
  - ► last column contains only 1's
  - ► all other columns contain a single 1 somewhere
- ► **Elementary transformation** = summation of two rows
- ► Some rows can be deleted if they cannot be put into required form $->$ the matrix $[H_R]$ will have J rows (the more the better)
- ► Denote with A_j the entries of the resulting vector:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ .. \\ A_J \end{bmatrix} = [H_R]r^T$$

Thresholding cyclic decoder

- Because a codeword c^T produces 0 when multiplied with $[H]$, it will produce 0 when multiplied with $[H_R]$ also
  - because rows of $[H_R]$ = summation of rows of $[H]$, but c^T makes a 0 with all of them
- Then
$$A = [H_R]r^T = [H_R](c + e)^T = [H_R]e^T$$
- e^T is the error word having 1's where errors are
- Consider how many of the entries A_k are equal to 1
  - If there is just one error on last position of e, **all** A_k are 1
  - If there is just one error on some other position (non-last), only a **single** A_k is 1

Thresholding cyclic decoder

- **Theorem:** If there are at most $\left\lfloor \frac{J}{2} \right\rfloor$ errors in e, then
  - if $\sum A_k > \left\lfloor \frac{J}{2} \right\rfloor$, then there is an error on last position
  - if $\sum A_k \leq \left\lfloor \frac{J}{2} \right\rfloor$, then there is no error on last position
- So we can **reliably** detect an error on last position even though there might be errors on other positions
- **Proof:**
  - if no error is on last position, at most $\left\lfloor \frac{J}{2} \right\rfloor$ sums A_k are equal to 1
  - if there is error on last position, then there are less than half errors on other position, so less then half A_k's are 0
- Because the code is cyclic, we can rotate the codeword so that next bit is last one $->$ compute again and decide for second bit, and so on for all

Thresholding cyclic decoder

- Draw schematic on whiteboard only (sorry!)
- Contents:
    - a cyclic shift register
    - circuits for computing the sums A_k
    - *adder and comparator* that adds all A_j and compares sum with $\lfloor \frac{J}{2} \rfloor$
    - output XOR gate for correcting the error
- Operation
    - received word is loaded into shift register
    - compute A_j, decide and correct error on first bit (last position)
    - word rotates cyclically, do the same on next bit
    - and so on until all bits have been on last position and corrected

Packets of errors

- Until now, we considered a single error
- If errors appear **independently** in a long data sequence, they will be typically rare −> only one error in a codeword is likely
- So a single error may be good enough for random errors

But:

- In real life, many times the errors appear in packets
- A **packet of errors** (*an error burst*) is a sequence of two or more **consecutive errors**
  - examples: fading in wireless channels
- The **length** of the packet = the number of consecutive errors

Condition on columns of [H]

- Consider *e* errors in a codeword

Conditions on the parity-check matrix $[H]$:

- Error **detection** of *e* independent errors
  - sum of **any** *e* or fewer columns is **non-zero**
- Error **detection** of a packet of *e* errors
  - sum of any **consecutive** *e* or fewer columns is **non-zero**
- Error **correction** of *e* independent errors
  - sum of **any** *e* or fewer columns is **unique**
- Error **correction** of a packet of *e* errors
  - sum of any **consecutive** *e* or fewer columns is **unique**

Theorem:
Any (n,k) cyclic codes is capable of detecting any error packet of length $n - k$ or less

- In other words: remainder after division with $g(x)$ is always non-zero
- A large fraction of longer bursts can also be detected (but not all)
- No proof (too complicated)

- More difficult to analyze in general, will consider **only the case of packets two errors**
- Cyclic encoder: identical! (might need a longer $g(x)$ though)
- Cyclic decoder with LFSR: similar, but **error locator must be changed**

Cyclic decoder for packets of 2 errors or less

- ▶ Similar schematic, but **error locator** is changed
- ▶ Operation is identical
- ▶ Error locator:
  - ▶ Assume the error word has errors on positions $(n - k)$ and $(n - k - 1)$
  - ▶ After phase I, the state of the LFSR = column $(n - k)$ + column $(n - k - 1)$
  $$S(n) = T^{n-k}U \oplus T^{n-k-1}U$$
  - ▶ After $k - 1$ samples, the first erroneous bit is at the output, and the state is
  $$S(n + k - 1) = T^{-1}U \oplus T^{-2}U = \begin{bmatrix} 0 \\ 1 \\ ... \\ 0 \end{bmatrix}$$
  - ▶ At the next sample, the state will be
  $$S(n + k) = T^{-1}U = \begin{bmatrix} 1 \\ 0 \\ ... \\ 0 \end{bmatrix}$$

- The error locator must detect these two states $->$ draw on whiteboard
- If only a single error appears $->$ also works

Summary of cyclic codes

- Generated using a generator polynomial $g(x)$
- Non-systematic:

$$c(x) = i(x) \cdot g(x)$$

- Systematic:

$$c(x) = b(x) \oplus X^{n-k} i(x)$$

  - $b(x)$ is the remainder of dividing $X^{n-k} i(x)$ to $g(x)$

- A codeword is always a multiple of $g(x)$
- Error detection: divide by $g(x)$, look at remainder
- Schematics:
  - Cyclic encoder
  - Cyclic decoder with LFSR
  - Thresholding cyclic decoder
  - Encoder/decoder for packets of up to 2 errors