

## Error Control Coding

## Linear block codes

# What is error control coding?

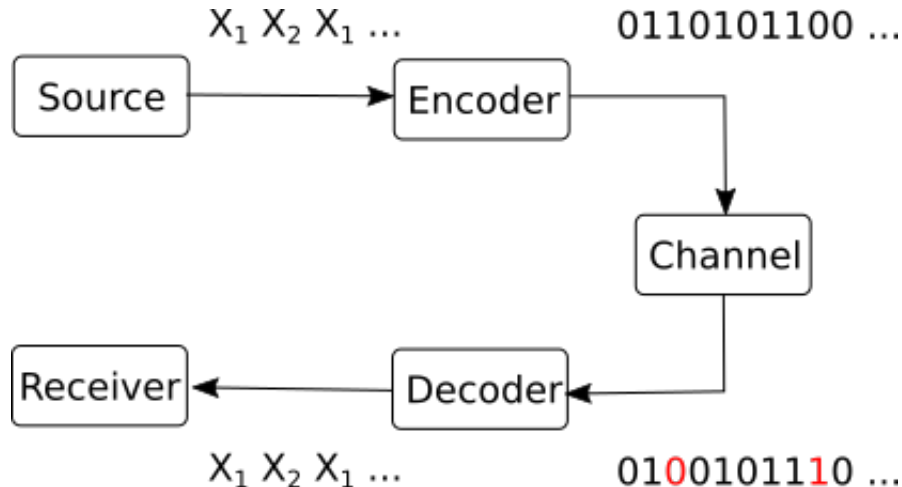


Figure 1: Communication system

- 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 ( $p = 0.01$ ,  $p(x_1) = 0.5$ ,  $p(x_2) = 0.5$ ):

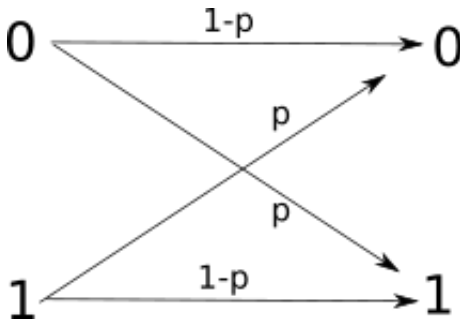


Figure 2: Binary symmetric channel (BSC)

# 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 viceversa
- ▶ Changing the value of a bit = modulo-2 sum with 1
- ▶ Value of a bit remains the same = modulo-2 sum with 0

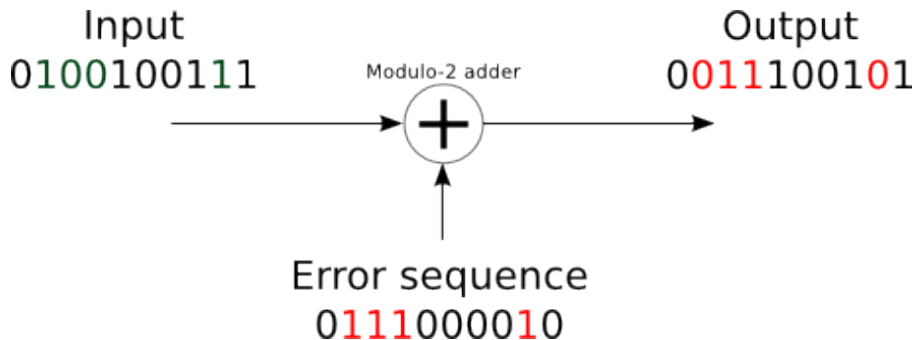


Figure 3: Channel error model

# Error detection and error correction

Binary error correction:

- ▶ For binary channels, know the location of error  $\Rightarrow$  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**

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

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

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

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:

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

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  $\mathbf{i}$  has a certain codeword  $\mathbf{c}$
- ▶ 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:

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

# Definitions

- ▶ 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

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

- ▶ 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:

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

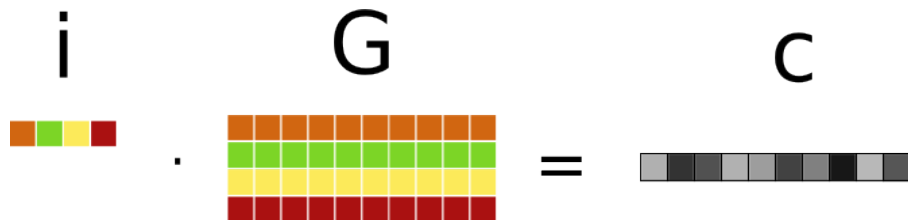


Figure 4: Codeword construction with generator matrix

- ▶  $[\mathbf{G}]$  = **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

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

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

$$[H]_{(n-k) \times n} = [I_{(n-k) \times (n-k)} \quad Q_{(n-k) \times k}^T]$$

- ▶ 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:

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

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  $\mathbf{r}$ :

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

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

# Syndrome-based error correction

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

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

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

## 7. Create a **syndrome lookup table**:

- ▶ for every possible error word  $\mathbf{e}$ , compute the syndrome  $\mathbf{z} = [\mathbf{H}] \cdot \mathbf{e}^T$
- ▶ 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 $\hat{\mathbf{e}}$

## 9. Find the correct word:

- ▶ adding the error word again will invert the errored bits back to the originals

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

# 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  $\Rightarrow$  can detect 2 errors
  - ▶ Sum of two columns equal to a third  $\Rightarrow$  cannot correct 3

**OR**

- ▶ Can correct one error
  - ▶ All columns are different  $\Rightarrow$  can correct 1 error
  - ▶ Sum of two columns equal to a third  $\Rightarrow$  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])
```

## Encoding & decoding example for Hamming(7,4)

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**.



# Circuits for encoding and decoding Hamming(7,4)

At whiteboard, using **shift registers**.

# SECDED Hamming codes

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 → 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:

$$c_0 c_1 c_2 i_3 c_4 i_5 i_6 i_7$$

- ▶ The parity 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 parity 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

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

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

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

- ▶ Syndrome:

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

- ▶ Syndrome-based error detection: syndrome non-zero
- ▶ Syndrome-based error correction: lookup table
- ▶ Hamming codes:  $[\mathbf{H}]$  contains all numbers  $1 \dots 2^r - 1$
- ▶ 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



# Binary polynomials

Every binary sequence  $\mathbf{a}$  corresponds to a polynomial  $\mathbf{a}(\mathbf{x})$  with binary coefficients

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

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.

# Generator polynomial

## 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.**

# Finding a generator polynomial

## 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

$$i(x) = i_0 \oplus i_1x \oplus \dots \oplus i_{k-1}x^{k-1}$$

**Non-systematic** codeword generation:

► Codeword =  $i(x) \cdot g(x)$

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

**Systematic** codeword generation:

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

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

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

- ▶ Cyclic shift to the right by 1 position

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

- ▶ Note that

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

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

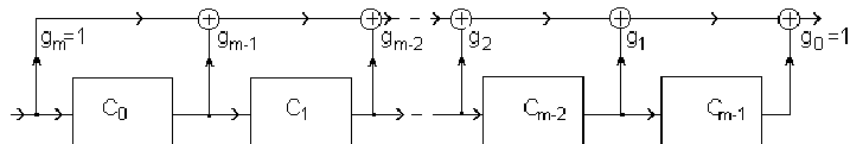


Figure 2

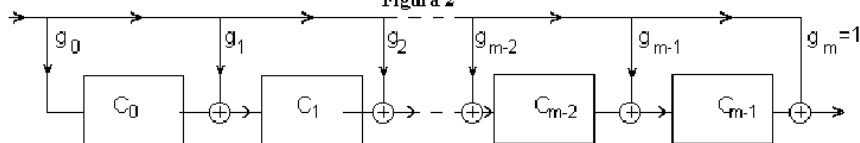


Figure 3

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  $\mathbf{X}(z)$
  - ▶ 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

# Linear analysis of multiplication circuits

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

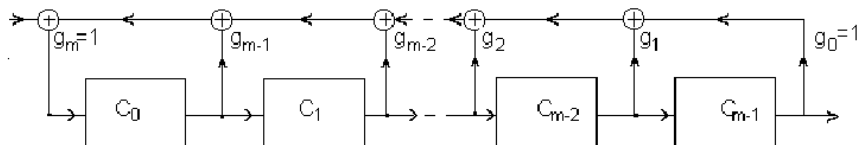


Figura 4

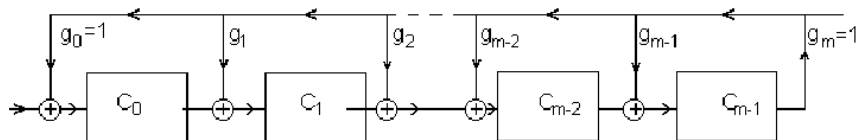


Figura 5

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

## Linear analysis of division circuits

We get:

$$Y(z) = \frac{X(z)}{G(z)}$$

meaning that the **output polynomial = input polynomial / g(x) polynomial**.

# Cyclic encoder circuit

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

# 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** (LSFR) = 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 LSFR

- ▶ **State** of the LSFR = 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  $g_i$ :

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

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

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

# Period of LSFR

- ▶ The number of states is finite  $\rightarrow$  they must repeat at some moment
- ▶ The state equal to 0 must not be encountered (LSFR will remain 0 forever)
- ▶ The **period** of the LSFR = 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**

# LSFR with inputs

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

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

- ▶ In general

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

where  $[U]$  is:

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

# Systematic cyclic encoder circuit

- ▶ Draw on whiteboard only (sorry!)
- ▶ Initially the LSFR 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
  - ▶ LSFR with feedback and input, input = information bits
- ▶ Switch in position II:
  - ▶ LSFR 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  $g(x) \implies$  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 to the input of the division circuit
  - ▶ after division, the cells end up in 0  $\iff$  no remainder  $\iff$  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 LSFR
  - ▶ it is a LSFR with feedback and input
  - ▶ the input is the codeword  $c(x)$
  - ▶ do computations at whiteboard ...
  - ▶ ... arrive at expression for matrix  $[H]$

# Error detection with cyclic codes

- ▶ Like usual for linear codes: check if received word is codeword or not
- ▶ Every codeword is multiple of  $g(x)$
- ▶ Check if received word is actually dividing with  $g(x)$ 
  - ▶ Use a circuit for division of polynomials
- ▶ If remainder is 0  $\Rightarrow$  it is a codeword, no error
- ▶ If remainder is non-0  $\Rightarrow$  error detected!
- ▶ Cyclic codes have very good error detection capabilities

# Error correction capability

## Theorem:

Any  $(n,k)$  cyclic codes is capable of detecting any error **burst** of length  $n - k$  or less.

- ▶ A large fraction of longer bursts can also be detected (but not all)
- ▶ For non-burst errors (random): more difficult to analyze



# Error correction with cyclic codes

- ▶ Like usual for linear codes: lookup table based on remainder
- ▶ Remainder of division = the effect of the error polynomial
- ▶ Create lookup table: for every error word, compute remainder
- ▶ Search the table for the remainder of the received word  $\Rightarrow$  find error word

# 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  $g(x)$
- ▶ Syndrome = remainder of division  $r(x)$  to  $g(x)$
- ▶ Error detection: remainder (syndrome) non-zero
- ▶ Error correction: lookup table