Data Representation and Error Correction in Computer Systems

Hamza Osman İLHAN

hoilhan@yildiz.edu.tr

YTU-CE / D037

Floating-Point Representation

- The signed magnitude, one's complement, and two's complement representation that we have just presented deal with integer values only.
- Without modification, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.

Floating-Point Representation

- If we are clever programmers, we can perform floating-point calculations using any integer format.
- This is called *floating-point emulation*, because floating point values aren't stored as such, we just create programs that make it seem as if floating-point values are being used.
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.

Floating-Point Representation

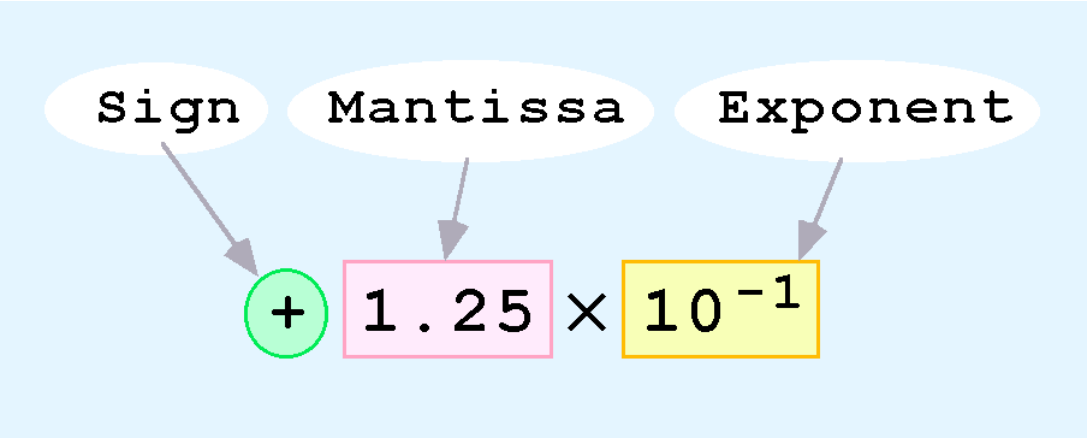
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

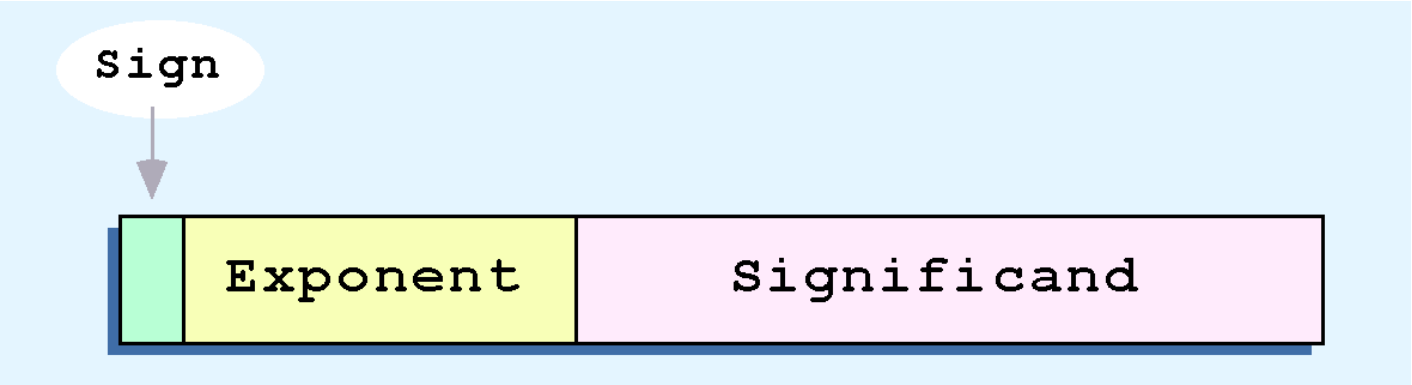
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



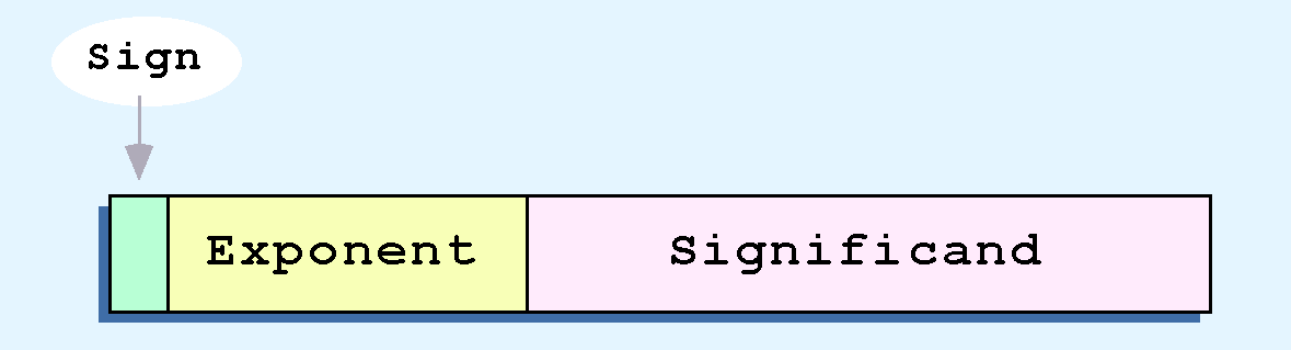
Floating-Point Representation

- Computer representation of a floating-point number consists of three fixed-size fields:



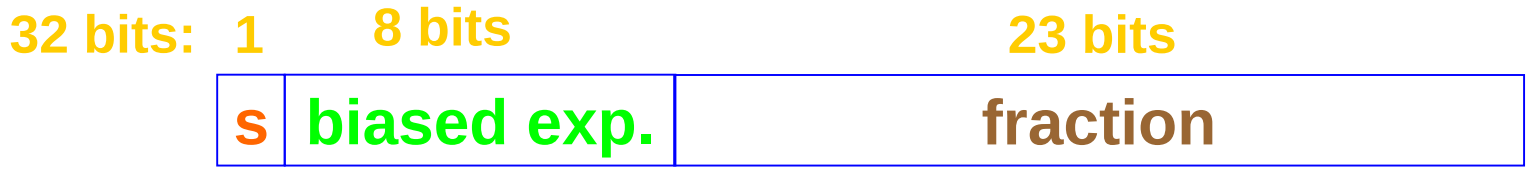
- This is the standard arrangement of these fields.

Floating-Point Representation



- The one-bit sign field is the sign of the stored value.
- The size of the exponent field, determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

IEEE-754 fp numbers - 1



$$N = (-1)^{s} \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 127)}$$

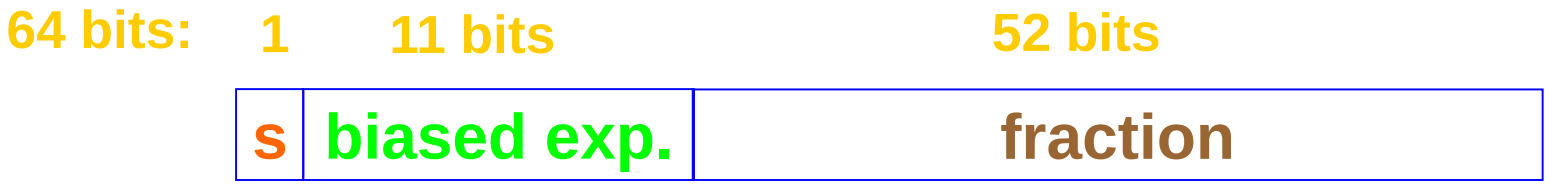
- Sign: 1 bit
- Mantissa: 23 bits
  - We "normalize" the mantissa by dropping the leading 1 and recording only its fractional part
- Exponent: 8 bits
  - In order to handle both +ve and -ve exponents, we add 127 to the actual exponent to create a "biased exponent":
    - 2^{-127} => biased exponent = 0000 0000 (= 0)
    - 2^{0} => biased exponent = 0111 1111 (= 127)
    - 2^{+127} => biased exponent = 1111 1110 (= 254)

IEEE-754 fp numbers - 2

- Example: Find the corresponding fp representation of 25.75
  - 25.75 => 00011001.110 => 1.1001110×2^4
  - sign bit = 0 (+ve)
  - normalized mantissa (fraction) = 100 1110 0000 0000 0000 0000
  - biased exponent = 4 + 127 = 131 => 1000 0011
  - so 25.75 => 0 1000 0011 100 1110 0000 0000 0000 0000 => x41CE0000
- Values represented by convention:
  - Infinity (+ and -): exponent = 255 (1111 1111) and fraction = 0
  - NaN (not a number): exponent = 255 and fraction $\neq$ 0
  - Zero (0): exponent = 0 and fraction = 0
    - note: exponent = 0 => fraction is *de-normalized,* i.e no hidden 1

IEEE-754 fp numbers - 3

- Double precision (64 bit) floating point



$$N = (-1)^{s} \times 1.\mathbf{fraction} \times 2^{(\mathbf{biased\ exp.} - 1023)}$$

- **Range & Precision:**
  - **32 bit:**
    - **mantissa of 23 bits + 1 => approx. 7 digits decimal**
    - **$2^{+/-127}$ => approx. $10^{+/-38}$**
  - **64 bit:**
    - **mantissa of 52 bits + 1 => approx. 15 digits decimal**
    - **$2^{+/-1023}$ => approx. $10^{+/-306}$**

Floating-point addition

- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper.
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.
- If the exponent requires adjustment, we do so at the end of the calculation.

Floating-point multiplication

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
- We multiply the two operands and add their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

- No matter how many bits we use in a floating-point representation, our model must be finite.
- The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.
- At some point, every model breaks down, introducing errors into our calculations.
- By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.

- Floating-point overflow and underflow can cause programs to crash.
- Overflow occurs when there is no room to store the high-order bits resulting from a calculation.
- Underflow occurs when a value is too small to store, possibly resulting in division by zero.

Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.

Character Codes

- Calculations aren't useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.

- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
- While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
  - The Java programming language, and some operating systems now use Unicode as their default character code.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

- The Unicode codes- pace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

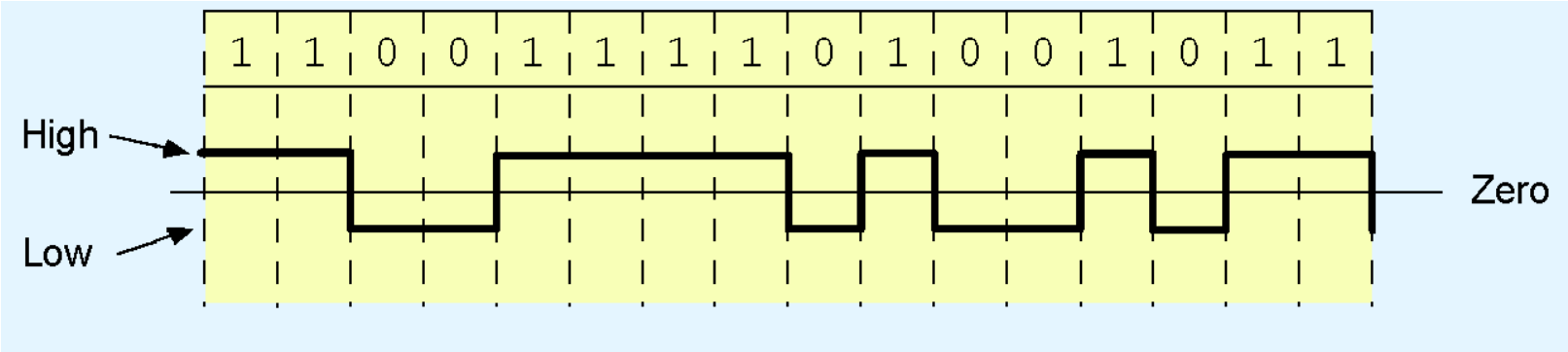
Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Codes for Data Recording and Transmission

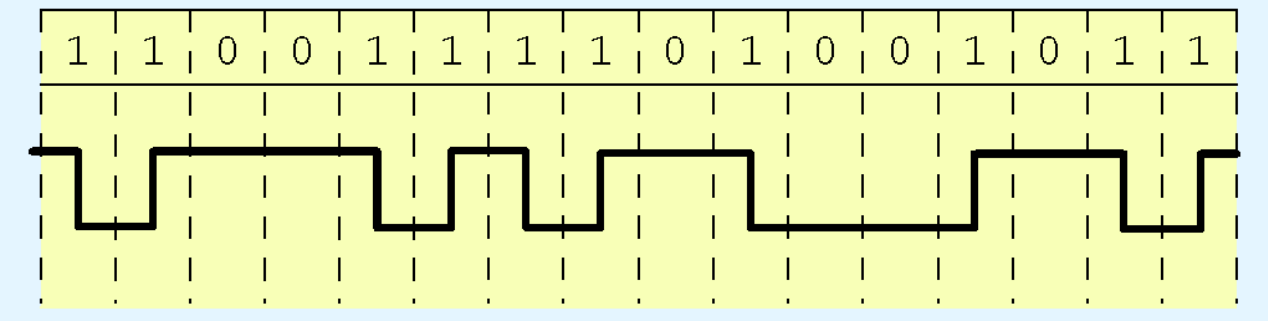
- When character codes or numeric values are stored in computer memory, their values are unambiguous.
- This is not always the case when data is stored on magnetic disk or transmitted over a distance of more than a few feet.
  - Owing to the physical irregularities of data storage and transmission media, bytes can become garbled.
- Data errors are reduced by use of suitable coding methods as well as through the use of various error-detection techniques.

- To transmit data, pulses of “high” and “low” voltage are sent across communications media.
- To store data, changes are induced in the magnetic polarity of the recording medium.
  - These polarity changes are called *flux reversals.*
- The period of time during which a bit is transmitted, or the area of magnetic storage within which a bit is stored is called a *bit cell*.

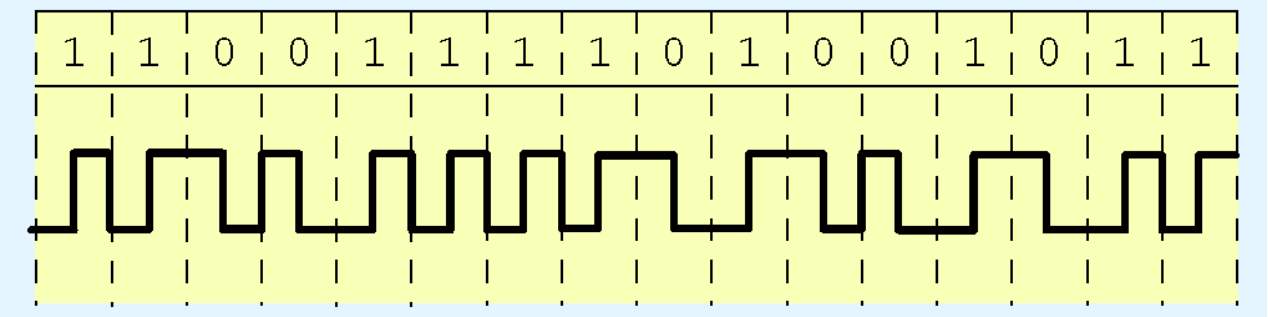
- The simplest data recording and transmission code is the non-return-to-zero (NRZ) code.
- NRZ encodes 1 as “high” and 0 as “low.”
- The coding of *OK* (in ASCII) is shown below.



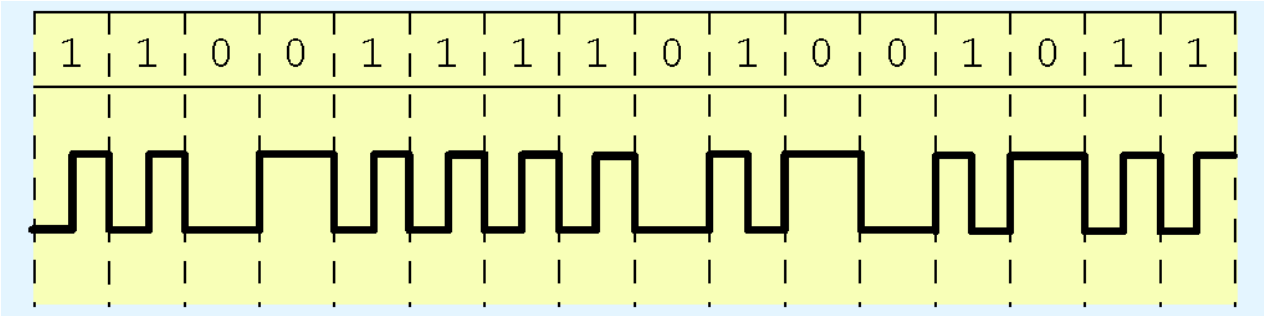
- The problem with NRZ code is that long strings of zeros and ones cause synchronization loss.
- Non-return-to-zero-invert (NRZI) reduces this synchronization loss by providing a transition (either low-to-high or high-to-low) for each binary 1.



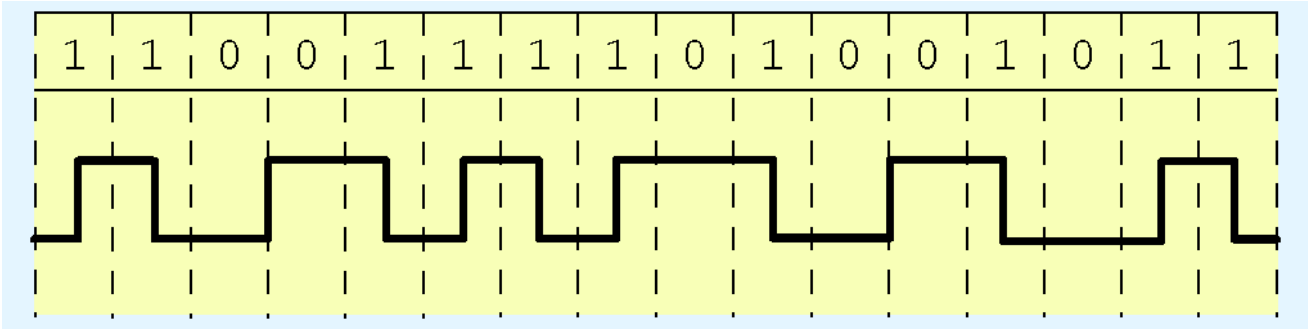
- Although it prevents loss of synchronization over long strings of binary ones, NRZI coding does nothing to prevent synchronization loss within long strings of zeros.
- Manchester coding (also known as phase modulation) prevents this problem by encoding a binary one with an “up” transition and a binary zero with a “down” transition.



- For many years, Manchester code was the dominant transmission code for local area networks.
- It is, however, wasteful of communications capacity because there is a transition on every bit cell.
- A more efficient coding method is based upon the frequency modulation (FM) code. In FM, a transition is provided at each cell boundary. Cells containing binary ones have a mid-cell transition.



- At first glance, FM is worse than Manchester code, because it requires a transition at each cell boundary.
- If we can eliminate some of these transitions, we would have a more economical code.
- Modified FM does just this. It provides a cell boundary transition only when adjacent cells contain zeros.
- An MFM cell containing a binary one has a transition in the middle as in regular FM.



Error Detection and Correction

- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
- As more bits are packed onto a square centimeter of disk storage, as communications transmission speeds increase, the likelihood of error increases-- sometimes geometrically.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.

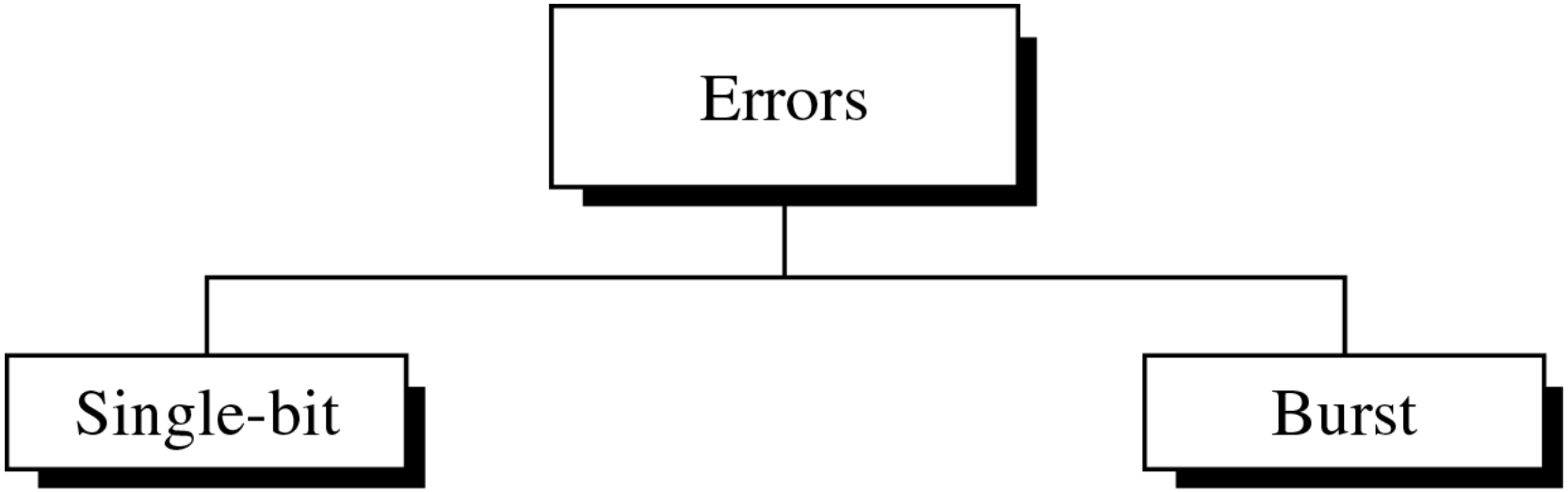
- Check digits, appended to the end of a long number can provide some protection against data input errors.
  - The last character of UPC barcodes and ISBNs are check digits.
  - Example UPC = 0-36000-24145-7
  - Example ISBN(10) = 0-201-53082-1
- Longer data streams require more economical and sophisticated error detection mechanisms.

Error Detection and Correction

- Types of Errors
- Detection
- Error Correction

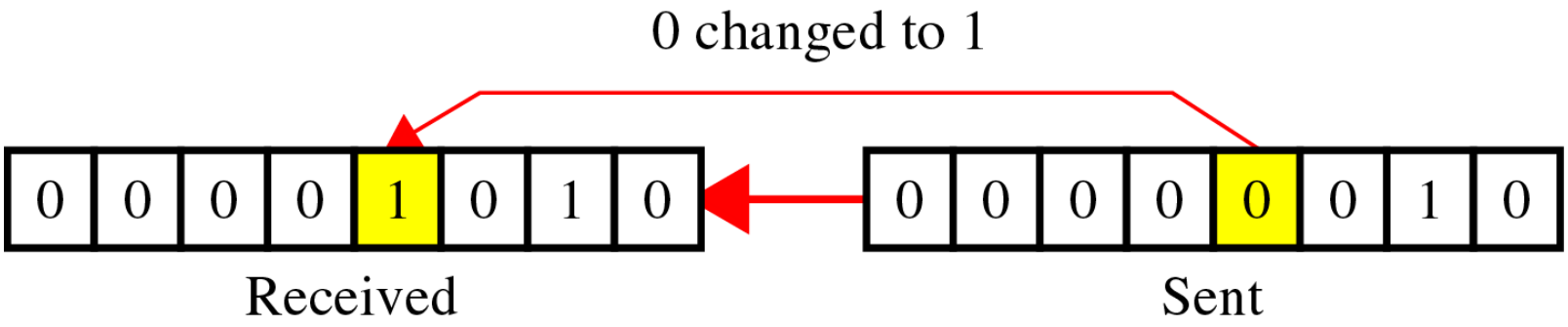
- Data can be corrupted during transmission. For reliable communication, error must be detected and corrected

Type of Errors



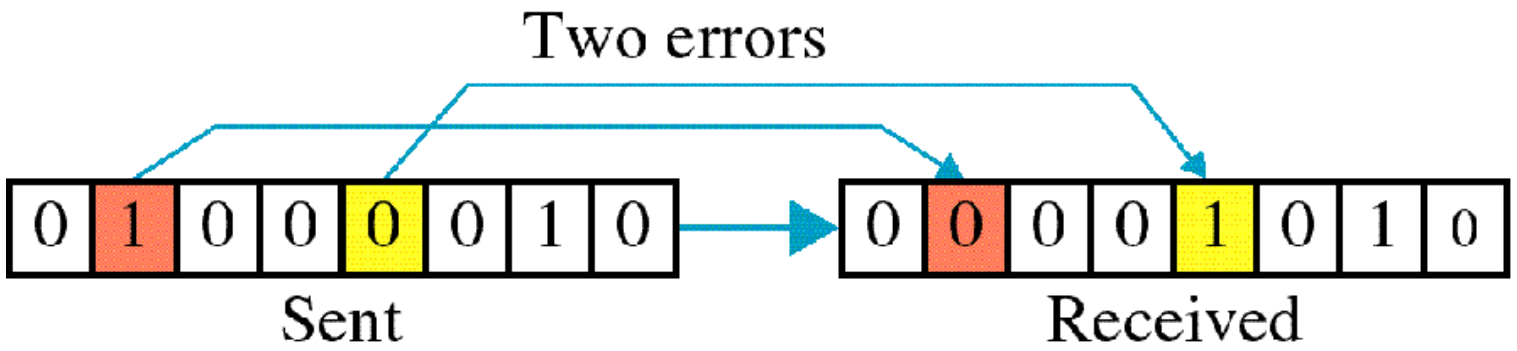
Type of Errors(cont'd)

- Single-Bit Error is when only one bit in the data unit has changed



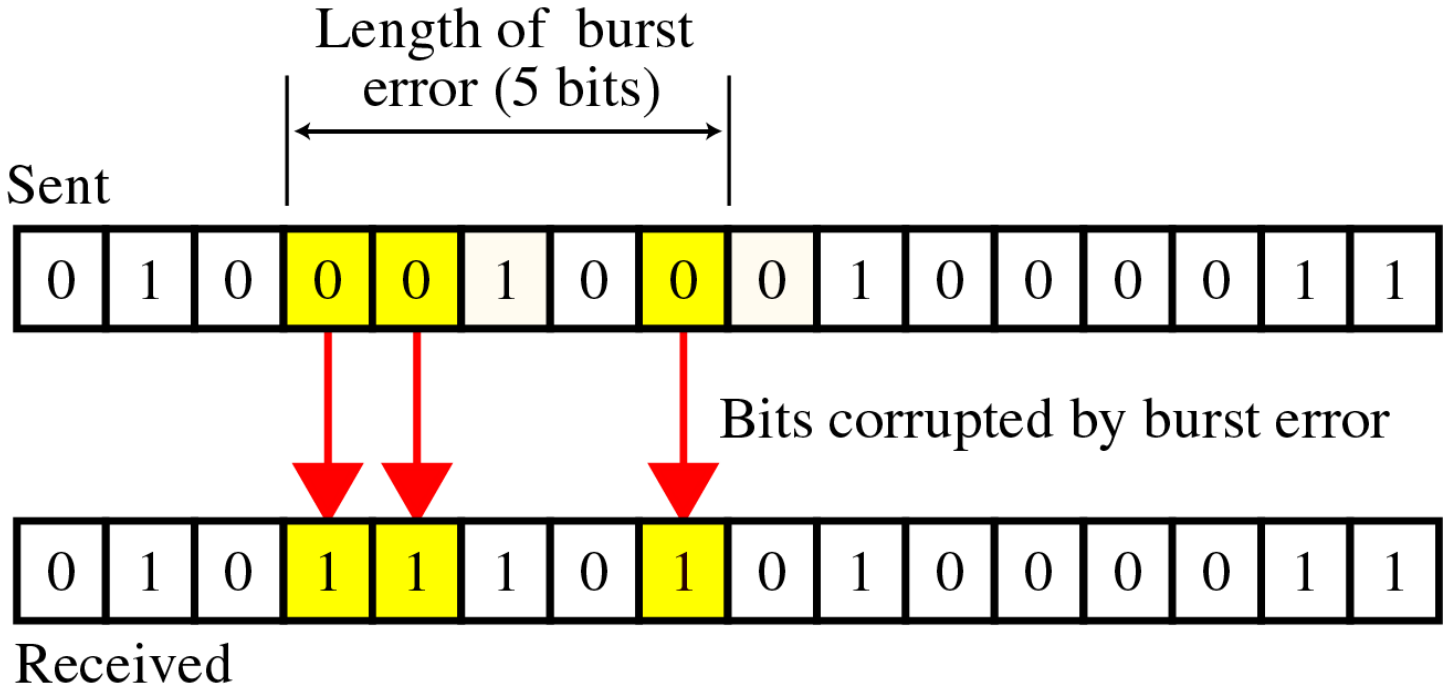
Type of Errors(cont'd)

- Multiple-Bit Error is when two or more nonconsecutive bits in the data unit have changed



Type of Errors(cont'd)

- Burst Error means that two or more consecutive bits in the data unit have changed

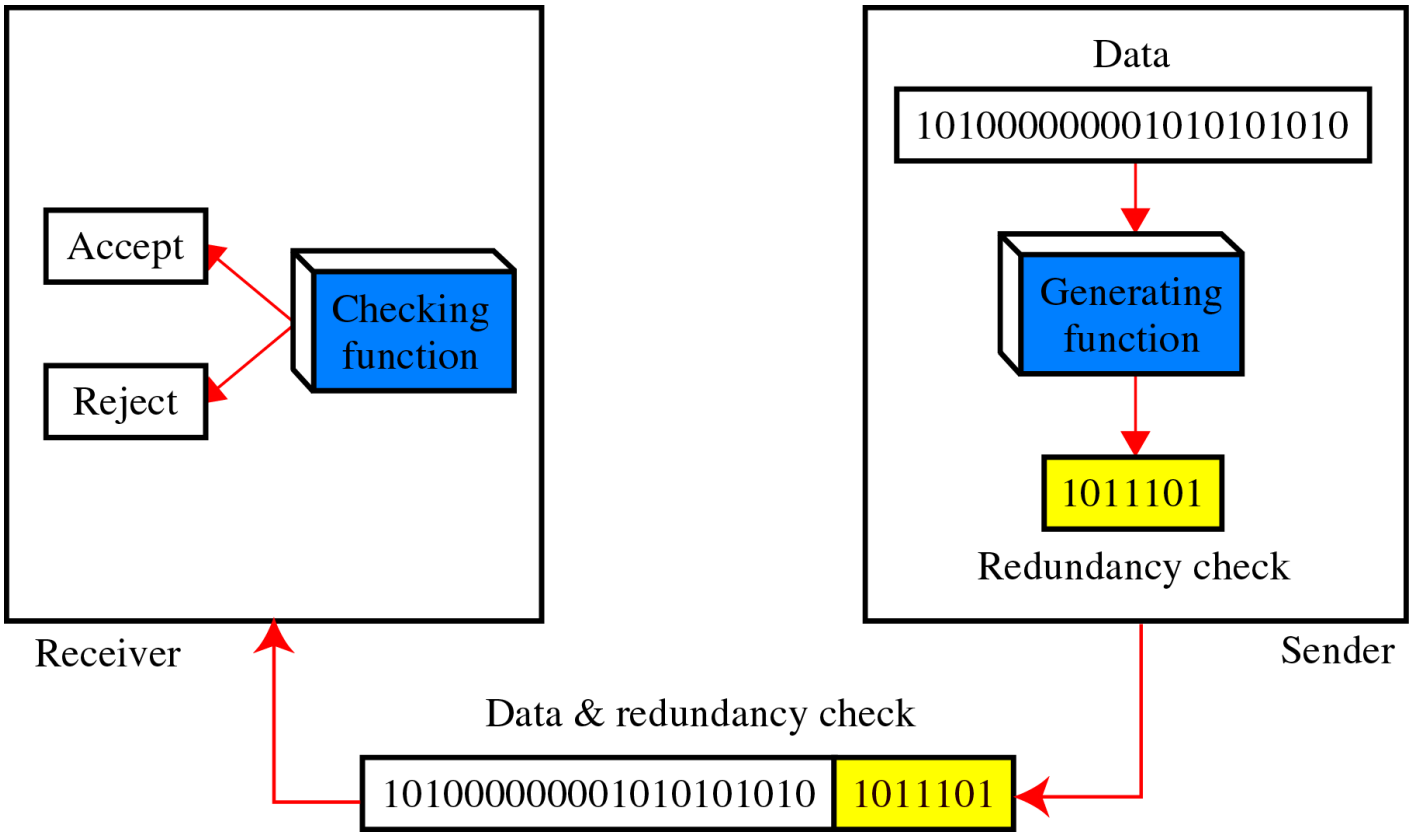


Detection

- Error detection uses the concept of redundancy, which means adding extra bits for detecting errors at the destination

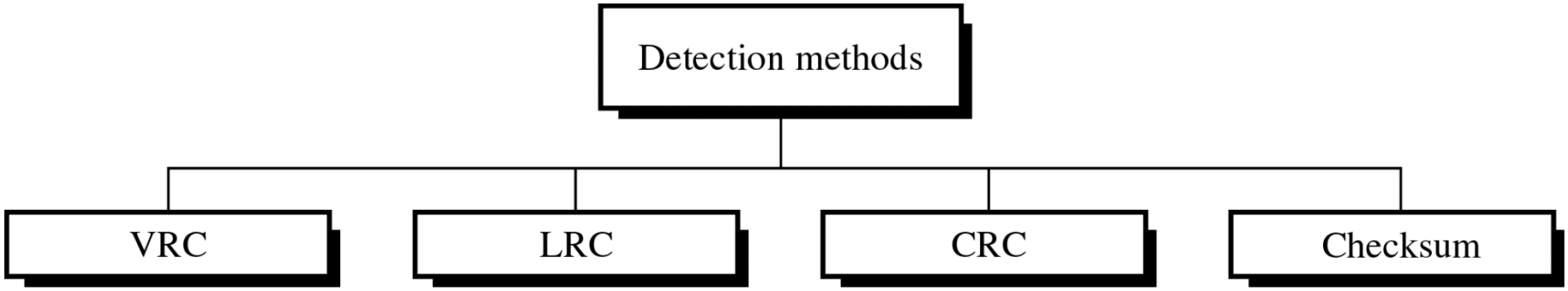
Detection(cont'd)

- Redundancy



Detection(cont'd)

- Detection methods
  - VRC(Vertical Redundancy Check)
  - LRC(Longitudinal Redundancy)
  - CRC(Cyclic redundancy Check)
  - Checksum

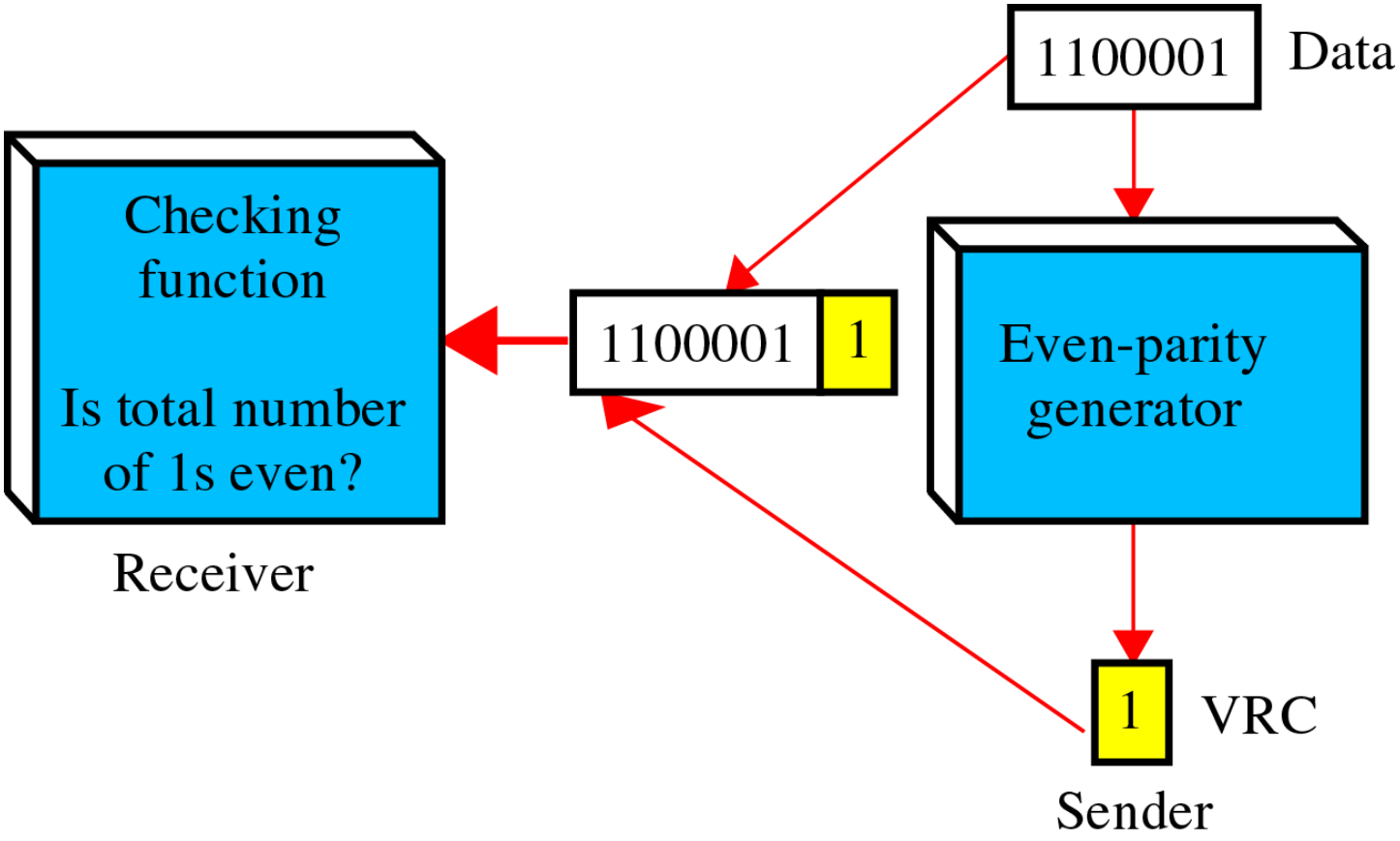


Detection(cont'd)

- VRC(Vertical Redundancy Check)
  - A parity bit is added to every data unit so that the total number of 1s(including the parity bit) becomes even for even-parity check or odd for odd-parity check
  - VRC can detect all single-bit errors. It can detect multiple-bit or burst errors only the total number of errors is odd.

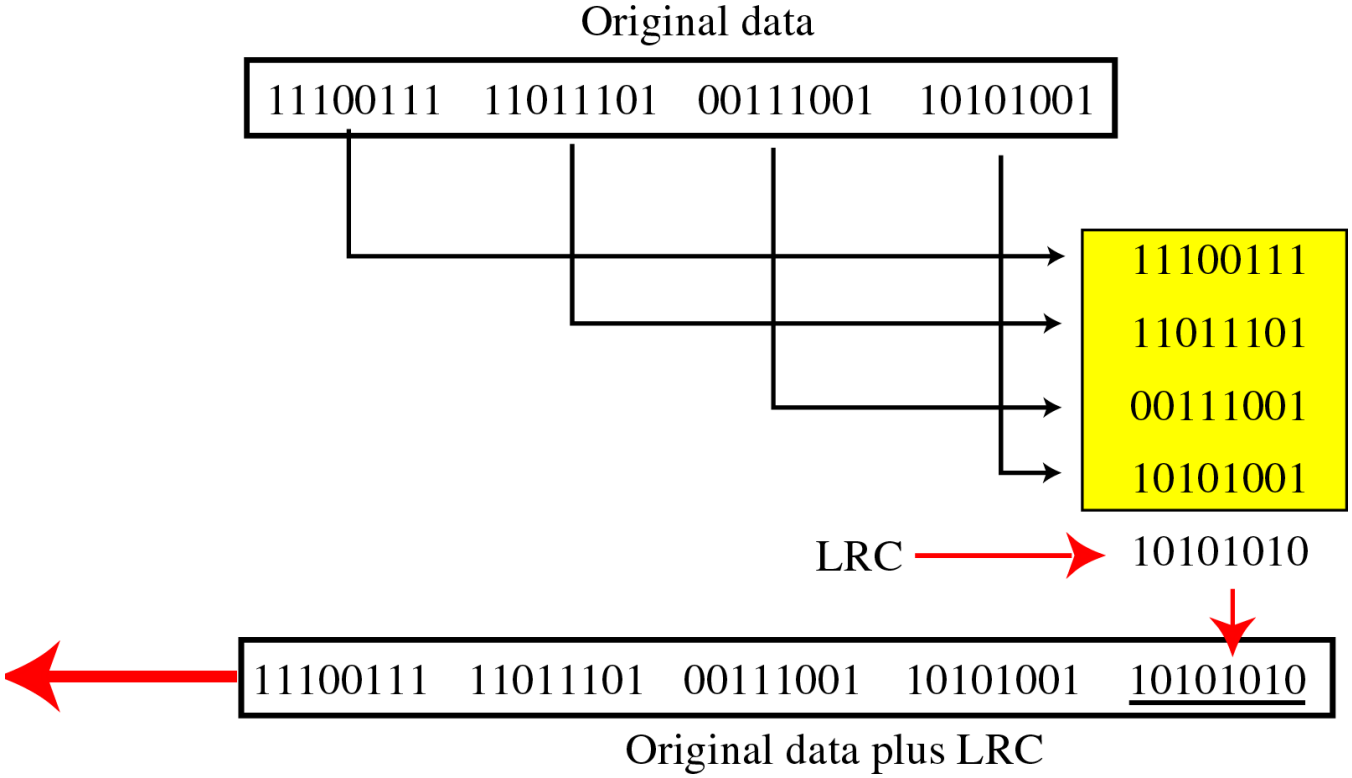
Detection(cont'd)

- Even parity VRC concept



Detection(cont'd)

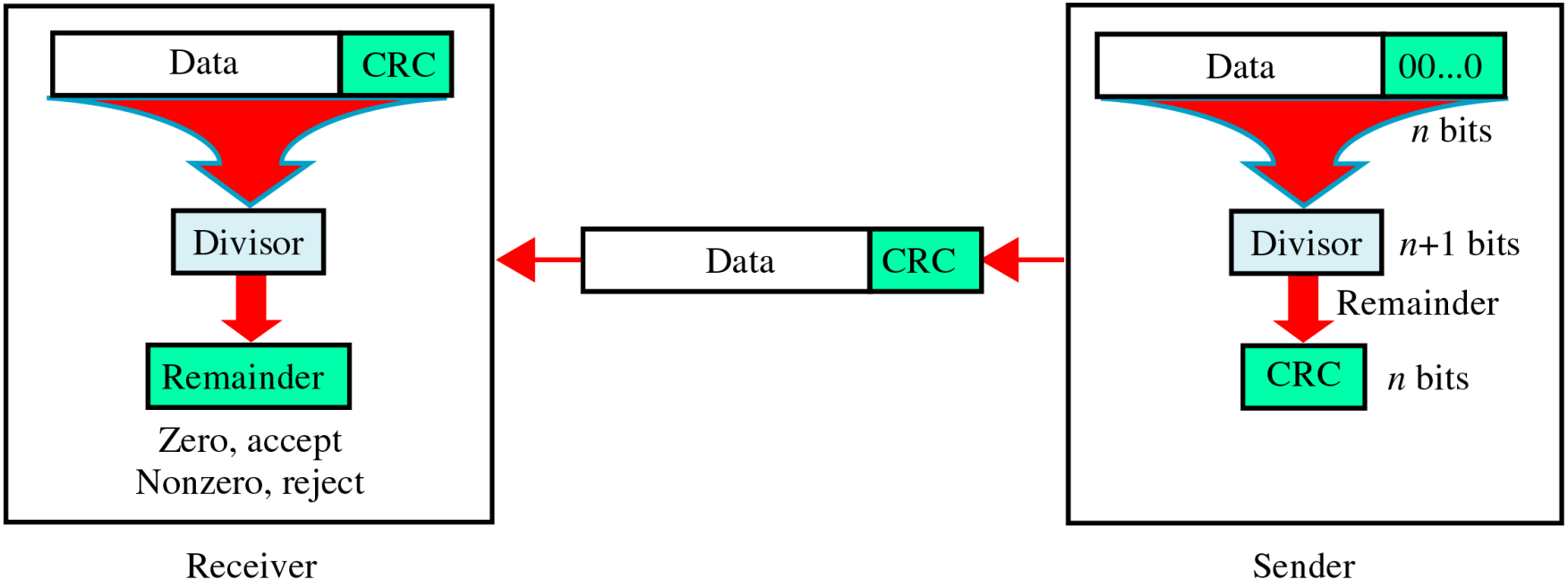
- LRC(Longitudinal Redundancy Check)
  - Parity bits of all the positions are assembled into a new data unit, which is added to the



Detection(cont'd)

- CRC(Cyclic Redundancy Check)

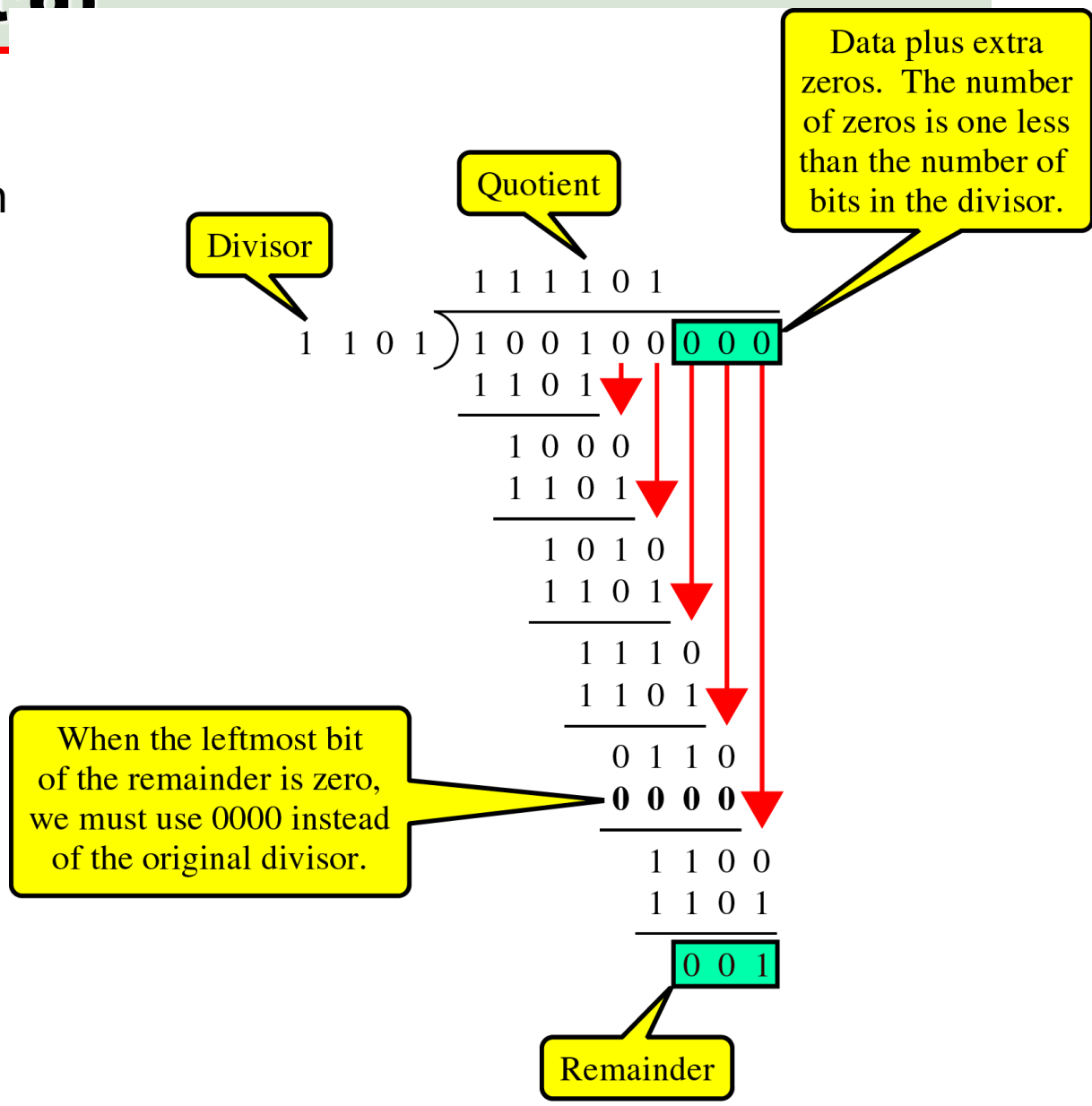
~ is based on binary division.



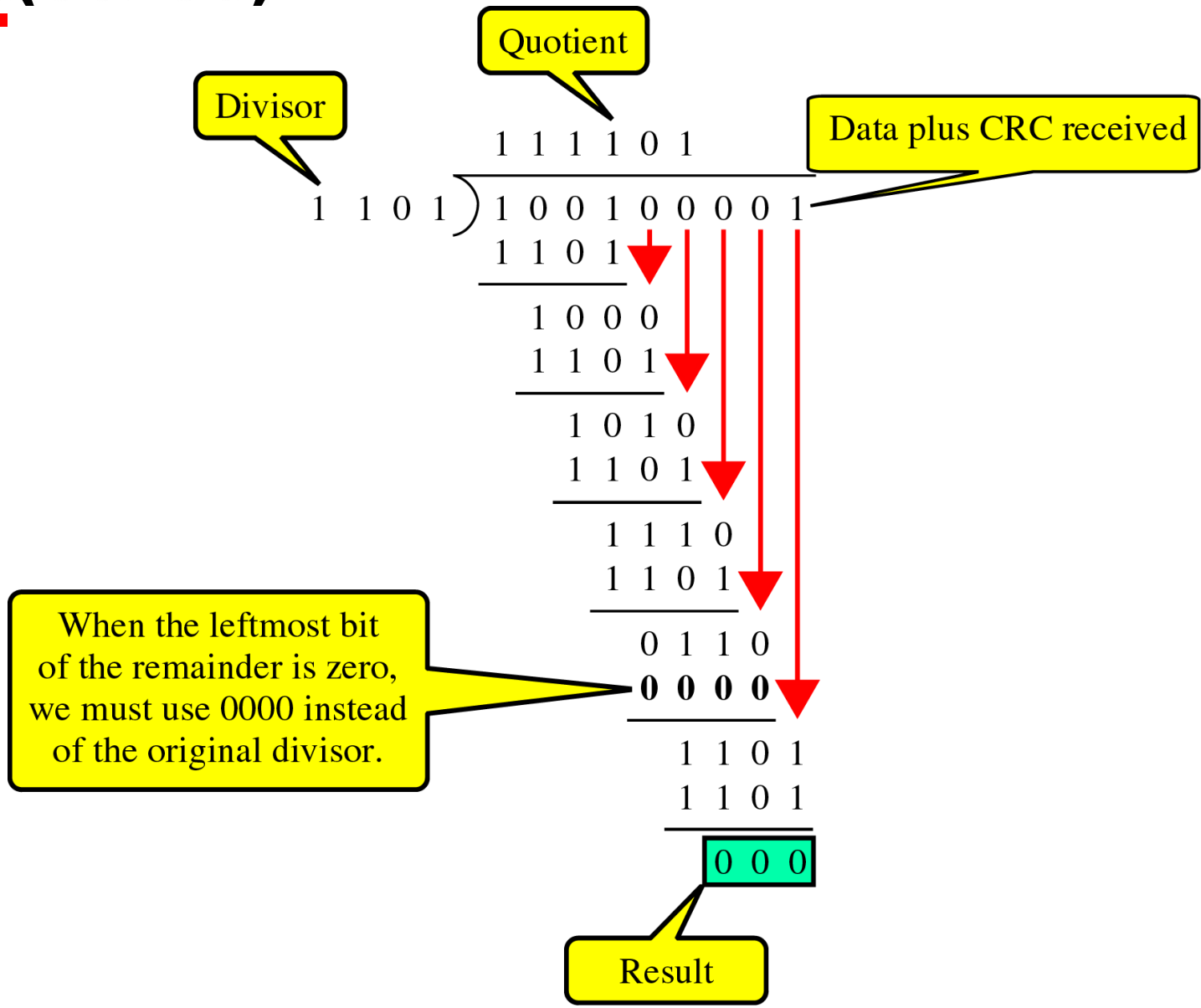
Detection(cont'd)

- CRC generator

~ uses modular-2 division



Detection(cont'd)



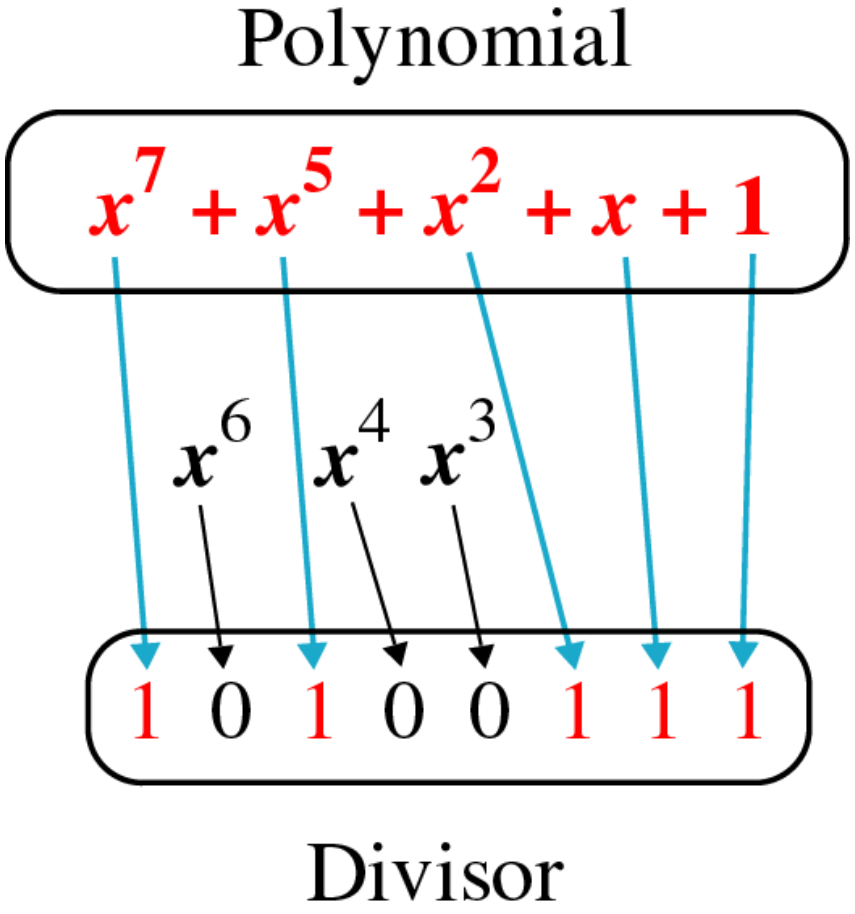
Detection(cont'd)

- Polynomials
  - CRC generator(divisor) is most often represented not as a string of 1s and 0s, but as an algebraic polynomial.

$$x^7 + x^5 + x^2 + x + 1$$

Detection(cont'd)

- A polynomial representing a divisor



Detection(cont'd)

- Standard polynomials

CRC-12

$$x^{12}+x^{11}+x^{3}+x+1$$

CRC-16

$$x^{16}+x^{15}+x^{2}+1$$

CRC-ITU-T

$$x^{16}+x^{12}+x^{5}+1$$

CRC-32

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$$

9.3 Error Correction

It can be handled in two ways

- when an error is discovered, the receiver requests to the sender to retransmit the entire data unit.

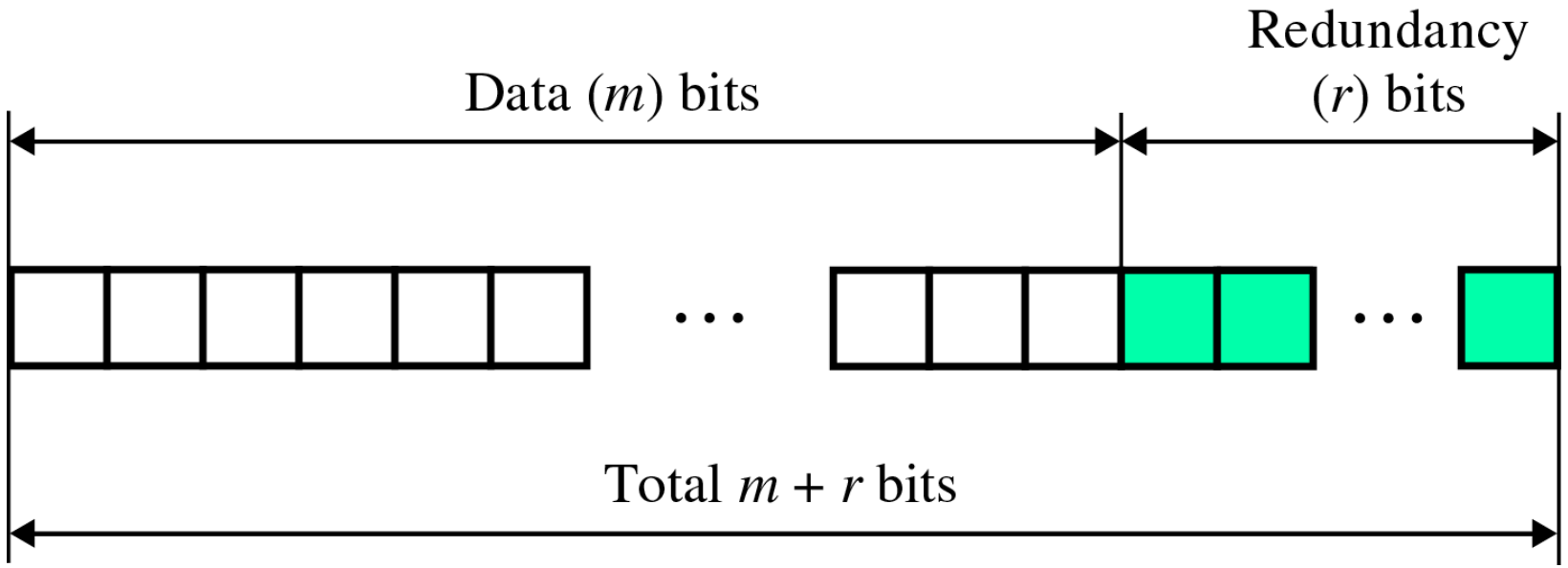
- a receiver can use an error-correcting code, which automatically corrects certain errors.

Error Correction(cont'd)

- Single-Bit Error Correction
  - parity bit
  - The secret of error correction is to locate the invalid bit or bits
  - For ASCII code, it needs a three-bit redundancy code(000-111)

Error Correction(cont'd)

- In order to calculate the number of redundancy bits (R) required to correct a given number of data bit (M)



Error Correction(cont'd)

- If the total number of bits in a transmittable unit is m+r, then r must be able to indicate at least m+r+1 different states

$$2^r \geq m + r + 1$$

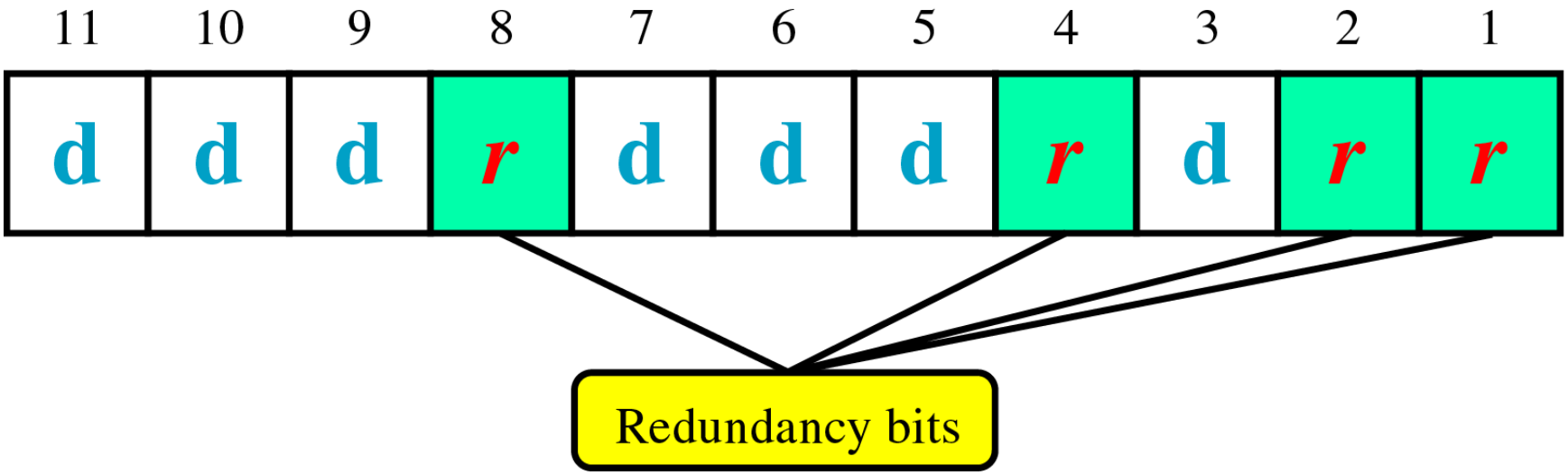
ex) For value of m is 7(ASCII), the smallest r value that can satisfy this equation is 4

$$2^4 \geq 7 + 4 + 1$$

Error Correction(cont'd)

- Hamming Code

  ~ developed by R.W.Hamming
- positions of redundancy bits in Hamming code



Error Correction(cont'd)

- each r bit is the VRC bit for one combination of data bits

  r_1 = bits 1, 3, 5, 7, 9, 11

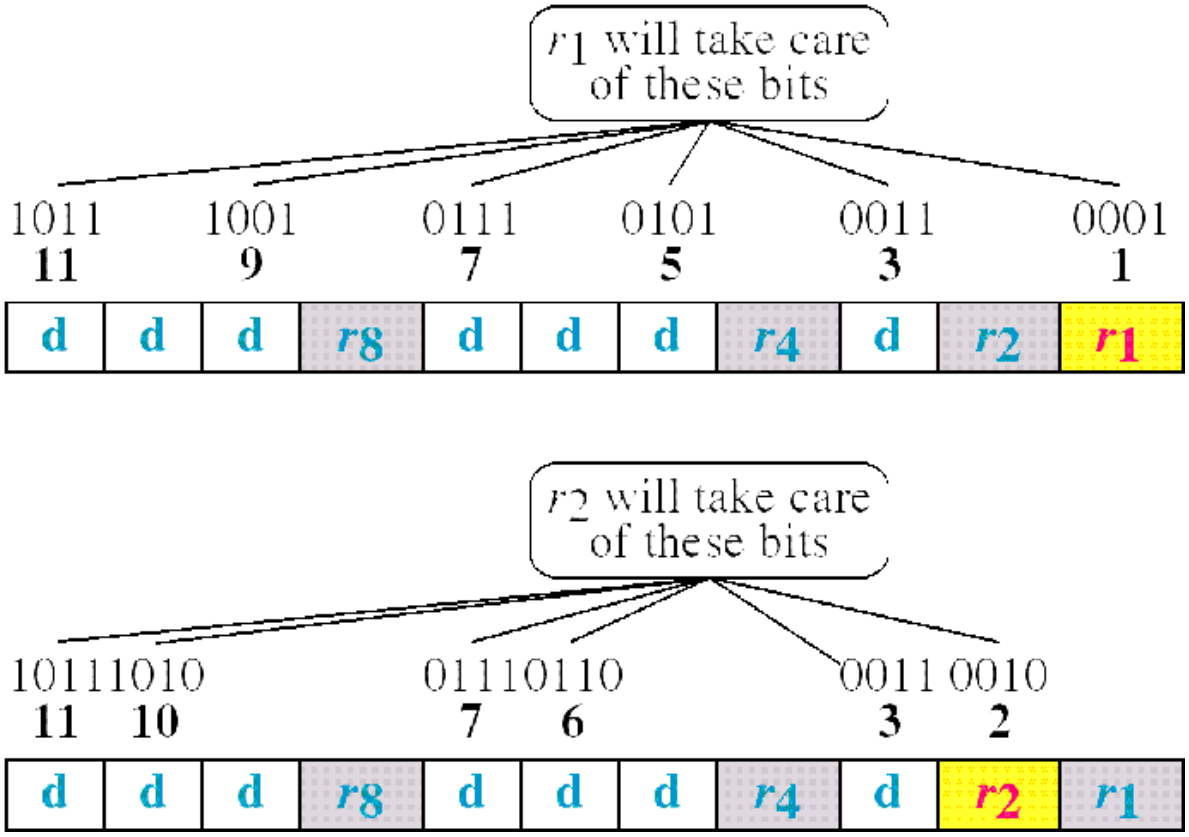
  r_2 = bits 2, 3, 6, 7, 10, 11

  r_4 = bits 4, 5, 6, 7

  r_8 = bits 8, 9, 10, 11

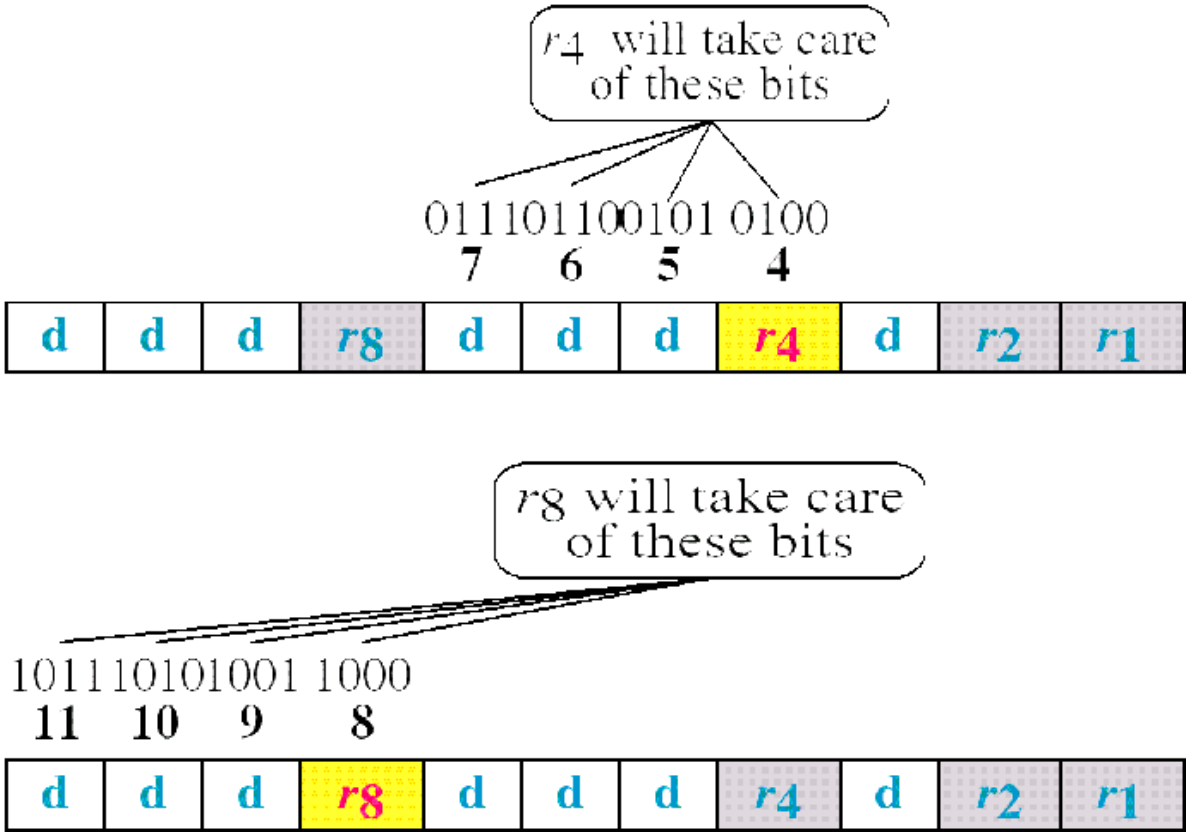
Error Correction(cont'd)

- Redundancy bits calculation(cont'd)



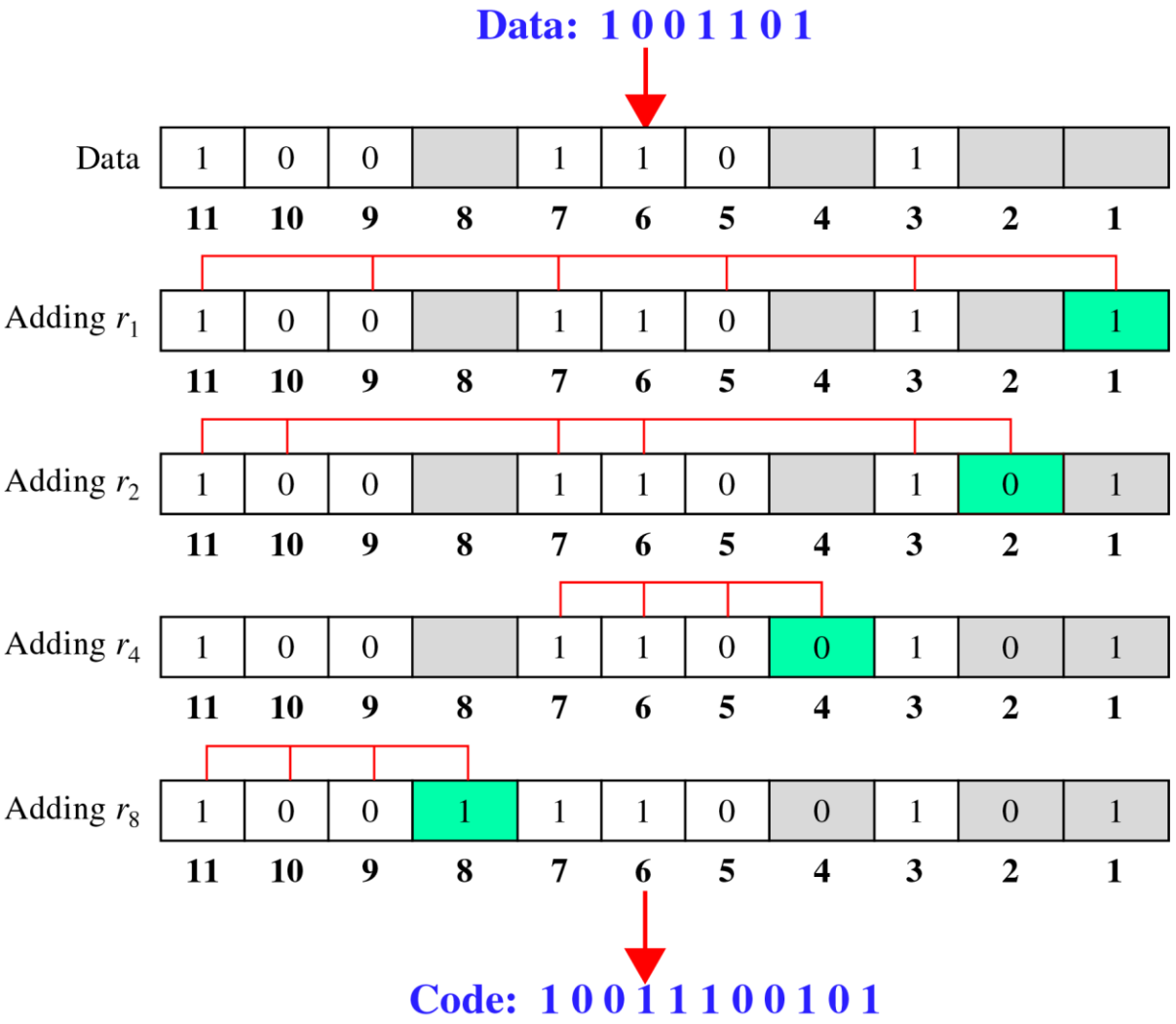
Error Correction(cont'd)

- Redundancy bits calculation



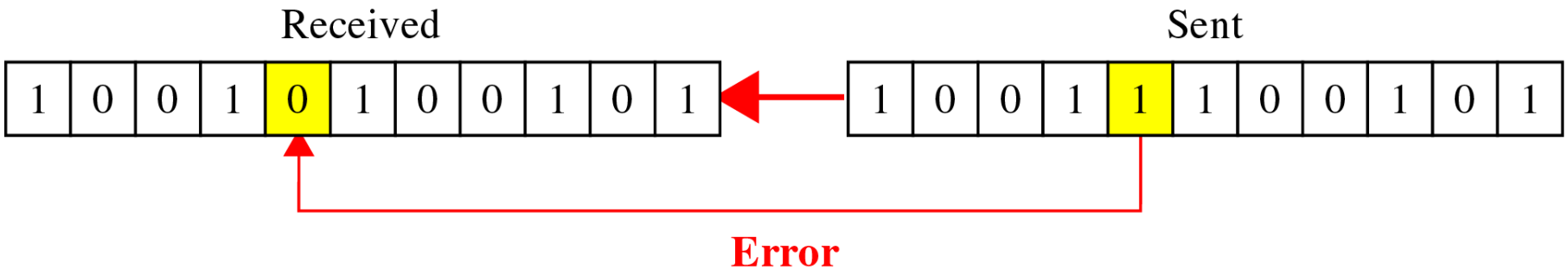
Error Correction(cont'd)

- Calculating the r values



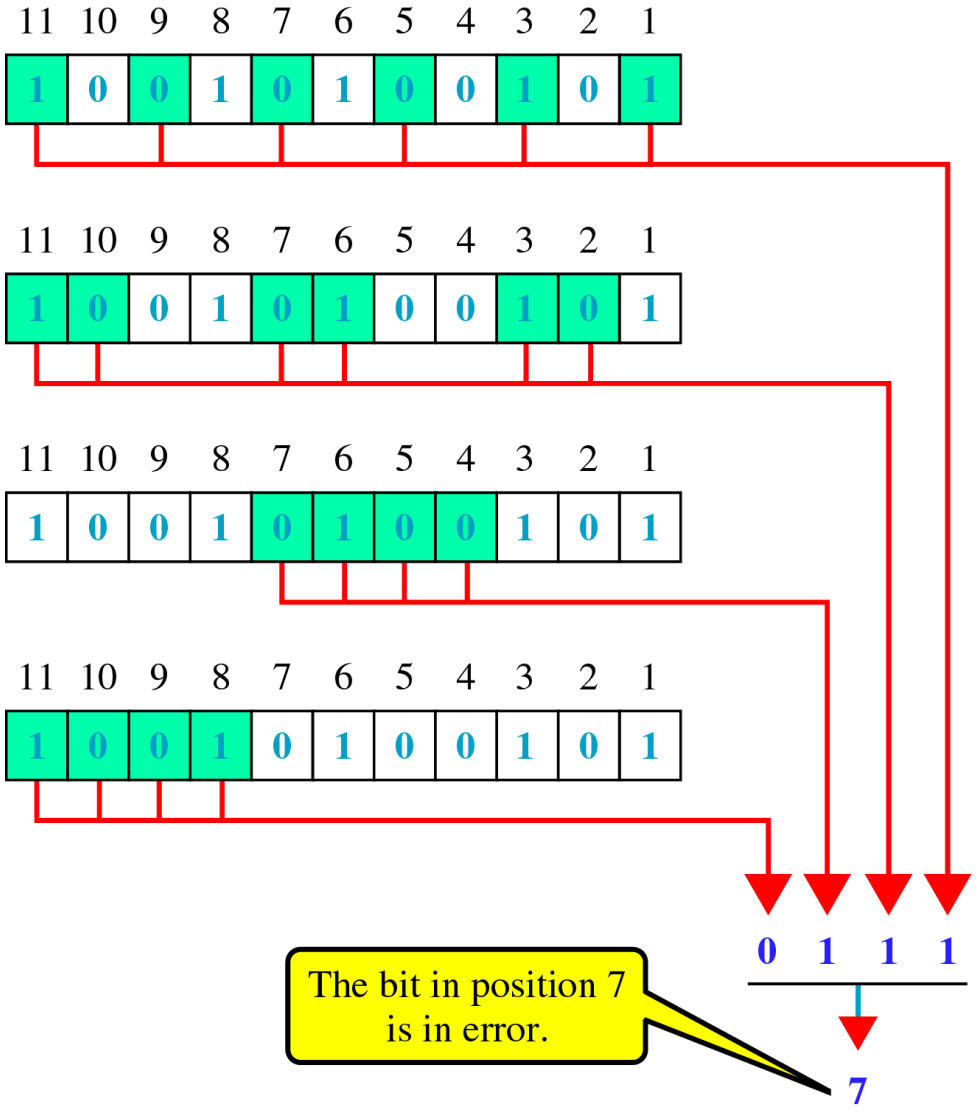
Error Correction(cont'd)

- Error Detection and Correction



Error Correction(cont'd)

- Error detection using Hamming Code



Error Correction(cont'd)

- Multiple-Bit Error Correction
  - redundancy bits calculated on overlapping sets of data units can also be used to correct multiple-bit errors.

  Ex) to correct double-bit errors, we must take into consideration that two bits can be a combination of any two bits in the entire sequence