DATA REPRESENTATION

Data Types

Complements

Fixed Point Representations

Floating Point Representations

Other Binary Codes

Error Detection Codes

Hamming Codes

1. DATA REPRESENTATION

Information that a Computer is dealing with

* Data
    - Numeric Data
        Numbers(Integer, real)
    - Non-numeric Data
        Letters, Symbols

* Relationship between data elements
    - Data Structures
        Linear Lists, Trees, Rings, etc

* Program (Instruction)

NUMERIC DATA REPRESENTATION

Data
    Numeric data – numbers (integer, real)
    Non-numeric data - symbols, letters

Number System
    Nonpositional number system
        - Roman number system

Positional number system

- Each digit position has a value called a *weight* associated with it
- Decimal, Octal, Hexadecimal, Binary

Base (or radix) R number

- Uses R distinct symbols for each digit
- Example AR = an-1 an-2 ... a1 a0 .a-1…a-m

REPRESENTATION OF NUMBERS - POSITIONAL NUMBERS

Decimal	Binary	Octal	Hexadecimal
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Convert 41.687510 to base 2

Fraction = 0.6875

0.6875
x 2
1.3750
x 2
0.7500
x 2
1.5000
x 2
1.0000

Integer = 41
41
20 1

10 0
5 0
2 1
1 0
0 1

$(41)_{10} = (101001)_2$ $(0.6875)_{10} = (0.1011)_2$

$(41.6875)_{10} = (101001.1011)_2$

2. COMPLEMENT OF NUMBERS

Two types of complements for base R number system:

- R's complement and (R-1)'s complement

The (R-1)'s Complement

Subtract each digit of a number from (R-1)

Example

- 9's complement of 83510 is 16410
- 1's complement of 10102 is 01012(bit by bit complement operation)

The R's Complement

Add 1 to the low-order digit of its (R-1)'s complement

Example

- 10's complement of 83510 is 16410 + 1 = 16510
- 2's complement of 10102 is 01012 + 1 = 01102

3. FIXED POINT NUMBERS

Numbers: Fixed Point Numbers and Floating Point Numbers

Binary Fixed-Point Representation

$X = x_n x_{n-1} x_{n-2} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$

Sign Bit(x_n): 0 for positive - 1 for negative

Remaining Bits($x_{n-1} x_{n-2} \ldots x_1 x_0 . x_{-1} x_{-2} \ldots x_{-m}$)

SIGNED NUMBERS

Need to be able to represent both *positive* and *negative* numbers

- Following 3 representations

Signed magnitude representation
Signed 1's complement representation
Signed 2's complement representation

Example: Represent +9 and -9 in 7 bit-binary number

Only one way to represent +9 ==> 0 001001
Three different ways to represent -9:
In signed-magnitude: 1 001001
In signed-1's complement: 1 110110
In signed-2's complement: 1 110111

In general, in computers, fixed point numbers are represented either integer part only or fractional part only.

CHARACTERISTICS OF 3 DIFFERENT REPRESENTATIONS

Complement
Signed magnitude: Complement *only* the sign bit
Signed 1's complement: Complement *all* the bits including sign bit
Signed 2's complement: Take the 2's complement of the number, *including* its sign bit.

Maximum and Minimum Representable Numbers and Representation of Zero

Signed Magnitude

Max: 2n - 2-m 011 ... 11.11 ... 1
Min: -(2n - 2-m) 111 ... 11.11 ... 1
Zero: +0 000 ... 00.00 ... 0
-0 100 ... 00.00 ... 0

Signed 1’s Complement

Max: 2n - 2-m 011 ... 11.11 ... 1
Min: -(2n - 2-m) 100 ... 00.00 ... 0

Zero: +0 000 ... 00.00 ... 0
-0 111 ... 11.11 ... 1

Signed 2's Complement

Max: 2n - 2-m 011 ... 11.11 ... 1
Min: -2n 100 ... 00.00 ... 0
Zero: 0 000 ... 00.00 ... 0

ARITHMETIC ADDITION: SIGNED MAGNITUDE

1] Compare their signs
[2] If two signs are the *same* ,
ADD the two magnitudes - Look out for an *overflow*
[3] If *not the same*, compare the relative magnitudes of the numbers and
then *SUBTRACT* the smaller from the larger --> need a subtractor to add
[4] Determine the sign of the result

Add the two numbers, including their sign bit, and discard any carry out of leftmost (sign) bit - Look out for an *overflow*

ARITHMETIC SUBTRACTION

Arithmetic Subtraction in 2's complement

Take the complement of the subtrahend (including the sign bit)
and add it to the minuend including the sign bits.

$(\pm A) - (- B) = (\pm A) + B$
$(\pm A) - B = (\pm A) + (- B)$

4. FLOATING POINT NUMBER REPRESENTATION

* The location of the fractional point is not fixed to a certain location
* The range of the representable numbers is wide

F = EM

mn ekek-1 ... e0 mn-1mn-2 … m0 . m-1 … m-m
sign exponent mantissa

- Mantissa

Signed fixed point number, either an integer or a fractional number

- Exponent
  Designates the position of the radix point

Decimal Value

$V(F) = V(M) * RV(E)$
M: Mantissa
E: Exponent
R: Radix

CHARACTERISTICS OF FLOATING POINT NUMBER REPRESENTATIONS

Normal Form

- There are many different floating point number representations of the same number
  → Need for a unified representation in a given computer

- *the most significant position of the mantissa contains a non-zero digit*

Representation of Zero

- Zero
  Mantissa = 0

- Real Zero
  Mantissa = 0
  Exponent
  = smallest representable number
  which is represented as
  00 ... 0
  ← Easily identified by the hardware

5. OTHER DECIMAL CODES

Decimal	BCD (8421)	2421	84-2-1	Excess-3
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110

4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

Note: 8,4,2,-2,1,-1 in this table is the weight associated with each bit position.

d3 d2 d1 d0: symbol in the codes

BCD: d3 x 8 + d2 x 4 + d1 x 2 + d0 x 1
$\Rightarrow$ 8421 code.
2421: d3 x 2 + d2 x 4 + d1 x 2 + d0 x 1
84-2-1: d3 x 8 + d2 x 4 + d1 x (-2) + d0 x (-1)
Excess-3: BCD + 3

GRAY CODE

Characterized by having their representations of the binary integers differ in only one digit between consecutive integers

* Useful in some applications

4-bit Gray codes

Decimal	Gray				Binary			
no	g3	g2	g1	g0	b3	b2	b1	b0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	1	0	0	1	0
3	0	0	1	0	0	0	1	1
4	0	1	1	0	0	1	0	0
5	0	1	1	1	0	1	0	1
6	0	1	0	1	0	1	1	0
7	0	1	0	0	0	1	1	1
8	1	1	0	0	1	0	0	0
9	1	1	0	1	1	0	0	1
10	1	1	1	1	1	0	1	0
11	1	1	1	0	1	0	1	1
12	1	0	1	0	1	1	0	0
13	1	0	1	1	1	1	0	1
14	1	0	0	1	1	1	1	0
15	1	0	0	0	1	1	1	1

6. ERROR DETECTING CODES

Parity System

- Simplest method for error detection
- One *parity* bit attached to the information
- *Even Parity* and *Odd Parity*

Even Parity
- One bit is attached to the information so that the total number of 1 bits is an even number

1011001 0
1010010 1

Odd Parity
- One bit is attached to the information so that the total number of 1 bits is an odd number

1011001 1
1010010 0

Error detection techniques

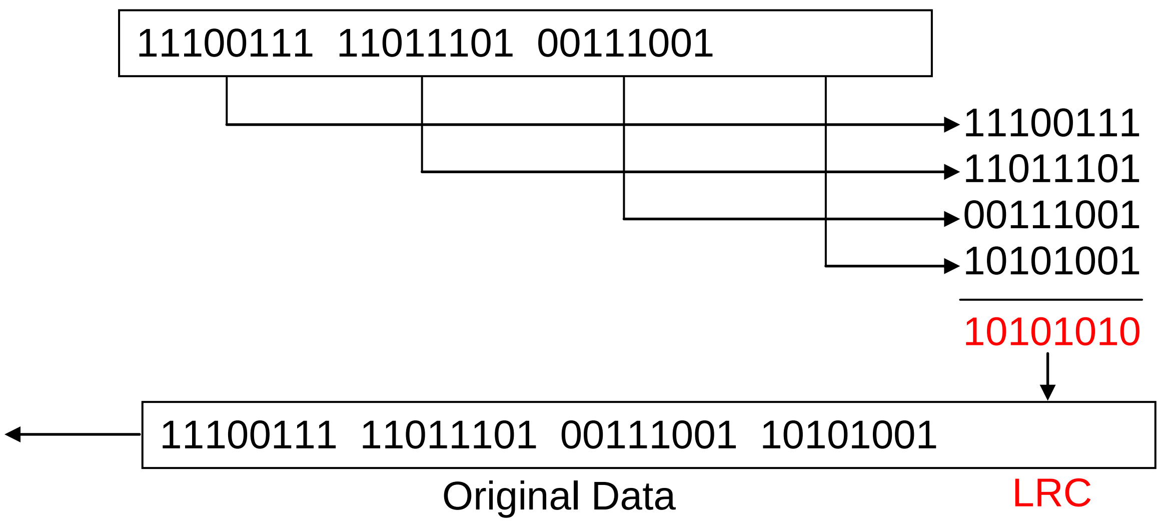
- **·Parity (VRC)**
- **·Longitudinal Redundancy Checks (LRC)**
- **·Cyclic Redundancy Checks (CRC)**
- **Checksum**

- Data transmission can contain errors
  - Single-bit
  - Burst errors of length n (n: distance between the first and last errors in data block)
- How to detect errors
  - If only data is transmitted, errors cannot be detected
    ◊ Send more information with data that satisfies a special relationship
    ◊ Add redundancy

- Vertical Redundancy Check (VRC)

– Append a single bit at the end of data block such that the number of ones is even ◊ Even Parity (odd parity is similar)
  0110011 ◊ 01100110
–
  0110001 ◊ 01100011
– VRC is also known as Parity Check
– Performance:
  » Detects all odd-number errors in a data block

- Longitudinal Redundancy Check (LRC)
  – Organize data into a table and create a parity for each column



- Cyclic Redundancy Check

Cyclic Redundancy Check (CRC)

- Parity check is based on addition; CRC is based on binary division
- A sequence of redundant bits (a CRC or CRC remainder) is appended to the end of the data unit
- These bits are later used in calculations to detect whether or not an error had occurred

CRC Steps

- On sender's end, data unit is divided by a predetermined divisor; remainder is the CRC

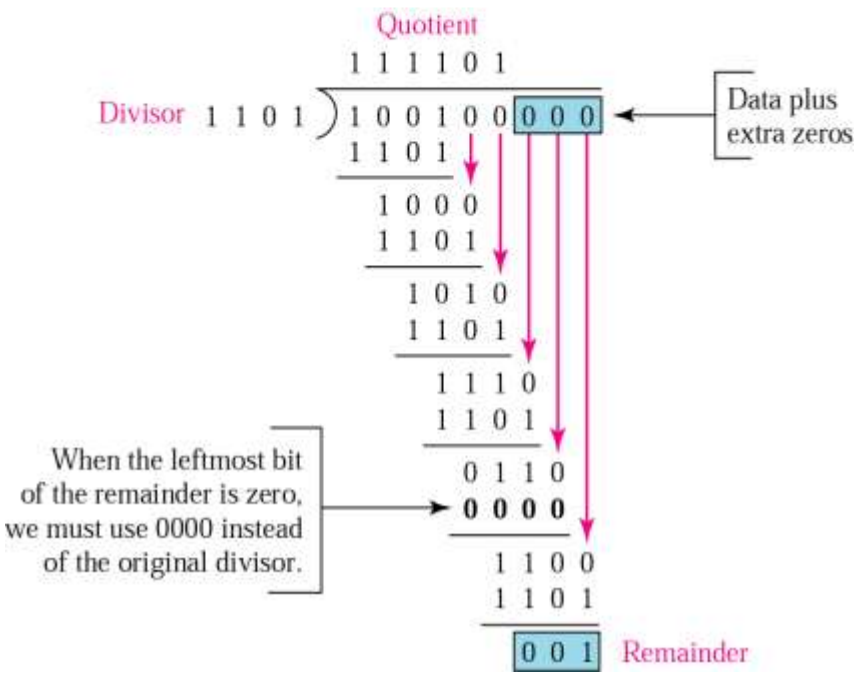
- When appended to the data unit, it should be exactly divisible by a second predetermined binary number
- At receiver's end, data stream is divided by same number
- If no remainder, data unit is assumed to be error-free

CRC Steps

- On sender's end, data unit is divided by a predetermined divisor; remainder is the CRC
- When appended to the data unit, it should be exactly divisible by a second predetermined binary number
- At receiver's end, data stream is divided by same number
- If no remainder, data unit is assumed to be error-free

CRC Generator

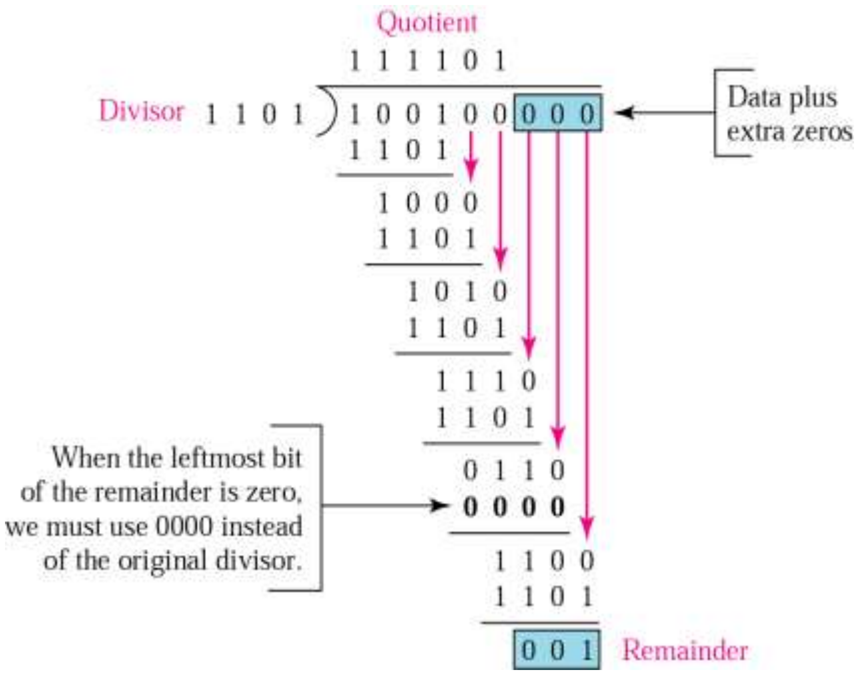
- Uses modulo-2 division
- Resulting remainder is the CRC



CRC Checker

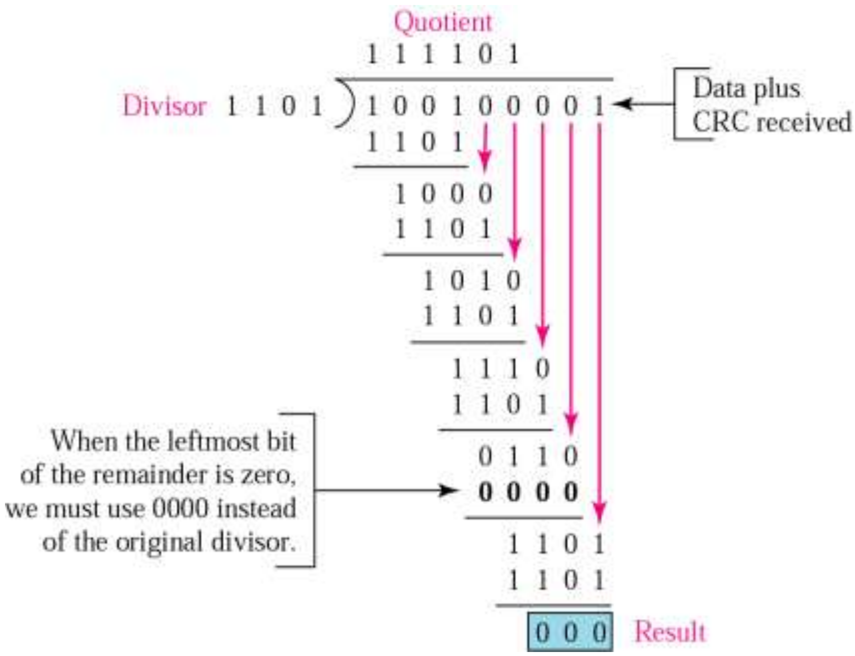
- Performed by receiver
- Data is appended with CRC
- Same modulo-2 division
- If remainder is 0, data are accepted

- Otherwise, an error has occurred

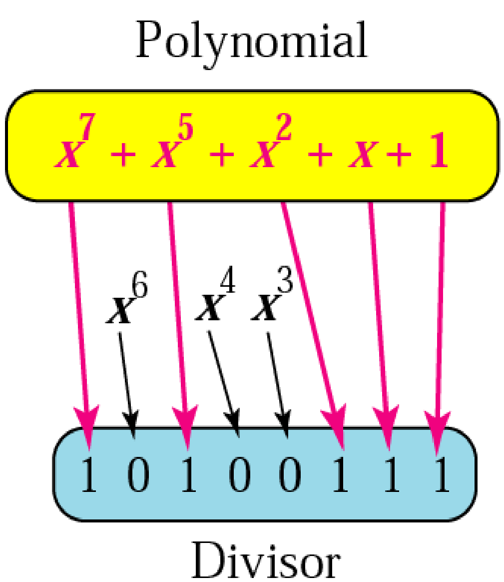


CRC Checker

- Performed by receiver
- Data is appended with CRC
- Same modulo-2 division
- If remainder is 0, data are accepted
- Otherwise, an error has occurred



Polynomials
- Used to represent CRC generator
- Cost effective method for performing calculations quickly



CRC Performance

- Can detect all burst errors affecting an odd number of bits
- Can detect all burst errors of length less than or equal to degree of polynomial
- Can detect with high probability burst errors of length greater than degree of the polynomial

- **Checksum**

- Performed by higher-layer protocols
- Also based on concept of redundancy

Checksum Generator

- At sender, checksum generator subdivides data unit into *k* equal segments of *n* bits
- Segments are added together using one's complement arithmetic to get the sum
- Sum is complemented and becomes the checksum, appended to the end of the data

Checksum Checker

- Receiver subdivides data unit in *k* sections of *n* bits
- Sections are added together using one's complement to get the sum
- Sum is complemented
- If result is zero, data are accepted; otherwise, rejected

Performance

- Detects all errors involving odd number of bits, most errors involving even number of bits
- Since checksum retains all carries, errors affecting an even number of bits would still change the value of the next higher column and the error would be detected
- If a bit inversion is balanced by an opposite bit inversion, the error is invisible

Error Correction

- Requires more redundancy bits; must know not only that an error had occurred, but *where* the error occurred in order to correct it
- Correction simply involves flipping the bit
- Hamming code may be applied to identify location where error occurred by strategically placed redundancy bits

Redundancy Bits

11	10	9	8	7	6	5	4	3	2	1
d	**d**	**d**	r_8	**d**	**d**	**d**	r_4	**d**	r_2	r_1

Example Hamming Code

- For a seven-bit data sequence

r1: bits 1, 3, 5, 7, 9, 11
r2: bits 2, 3, 6, 7, 10, 11
r3: bits 4, 5, 6, 7
r4: bits 8, 9, 10, 11

Redundancy Bits

11	10	9	8	7	6	5	4	3	2
d	d	d	r_8	d	d	d	r_4	d	r_2

Example Hamming Code

r_1 will take care of these bits.

11		9		7		5		3		1
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

r_2 will take care of these bits.

11	10			7	6			3	2	
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

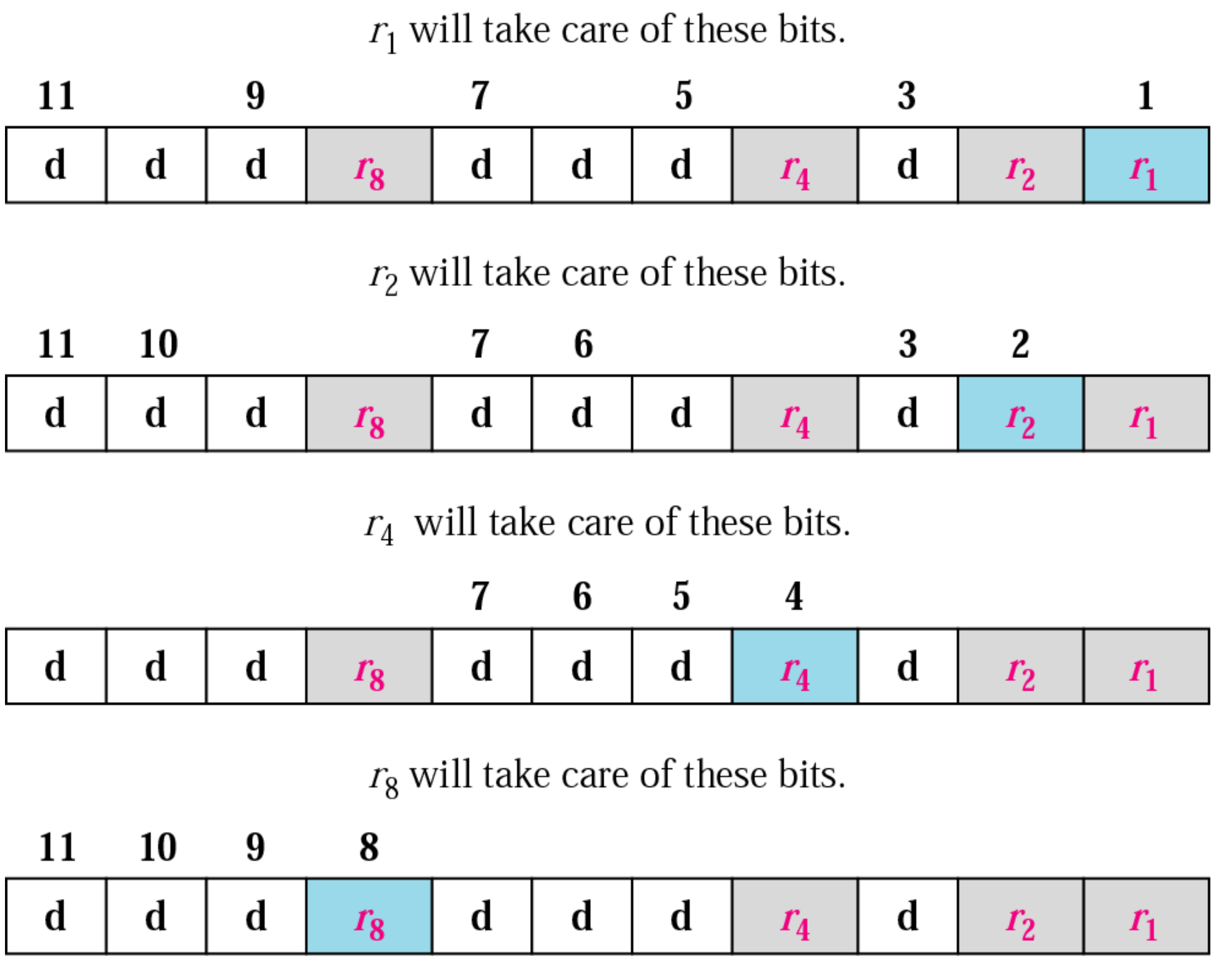
r_4 will take care of these bits.

				7	6	5	4			
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

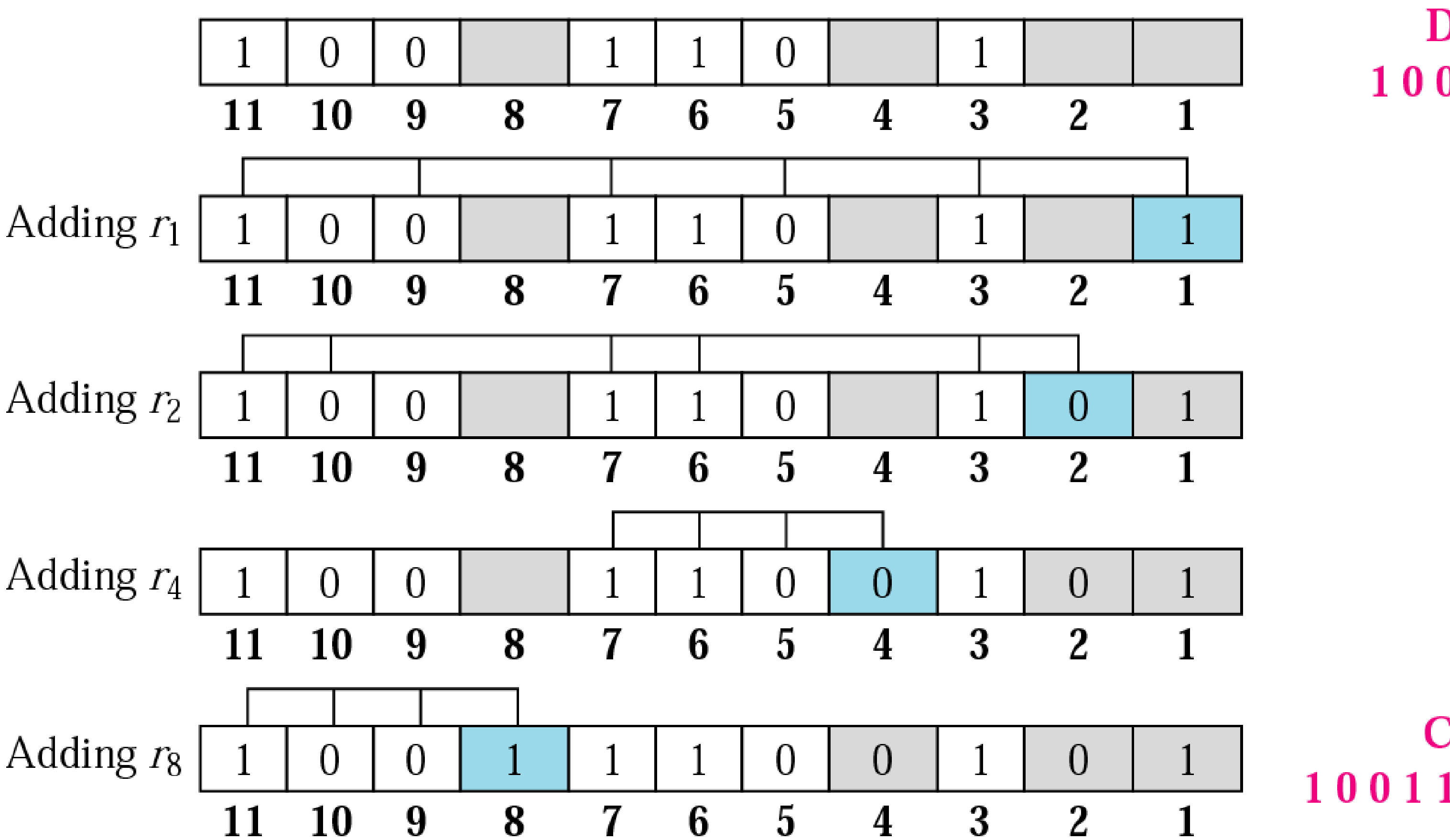
r_8 will take care of these bits.

11	10	9	8							
d	d	d	r_8	d	d	d	r_4	d	r_2	r_1

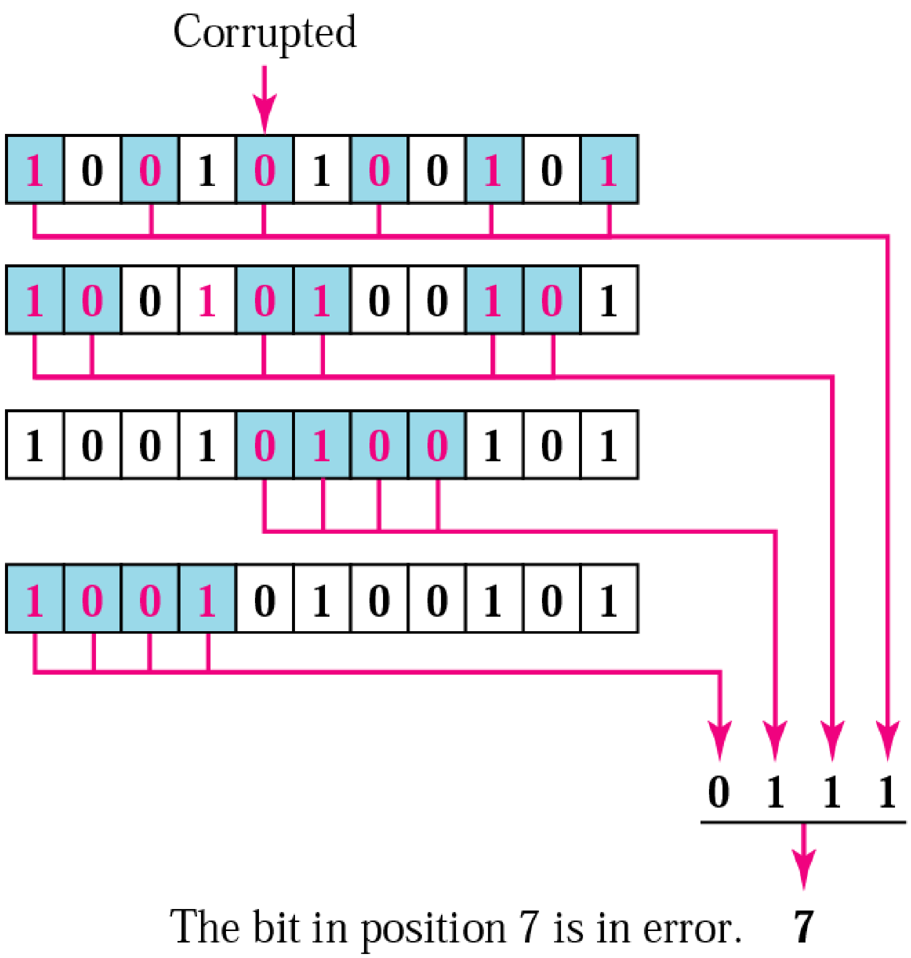
Example Hamming Code



Redundancy in bit calculation



Error Detection using Hamming



Burst Error Correction

- By rearranging the order of bit transmission of the data units, the Hamming code can correct burst errors
- Organize *n* units in a column and send first bit of each, followed by second bit of each, and so on
- Hamming scheme then allows us to correct the corrupted bit in each unit