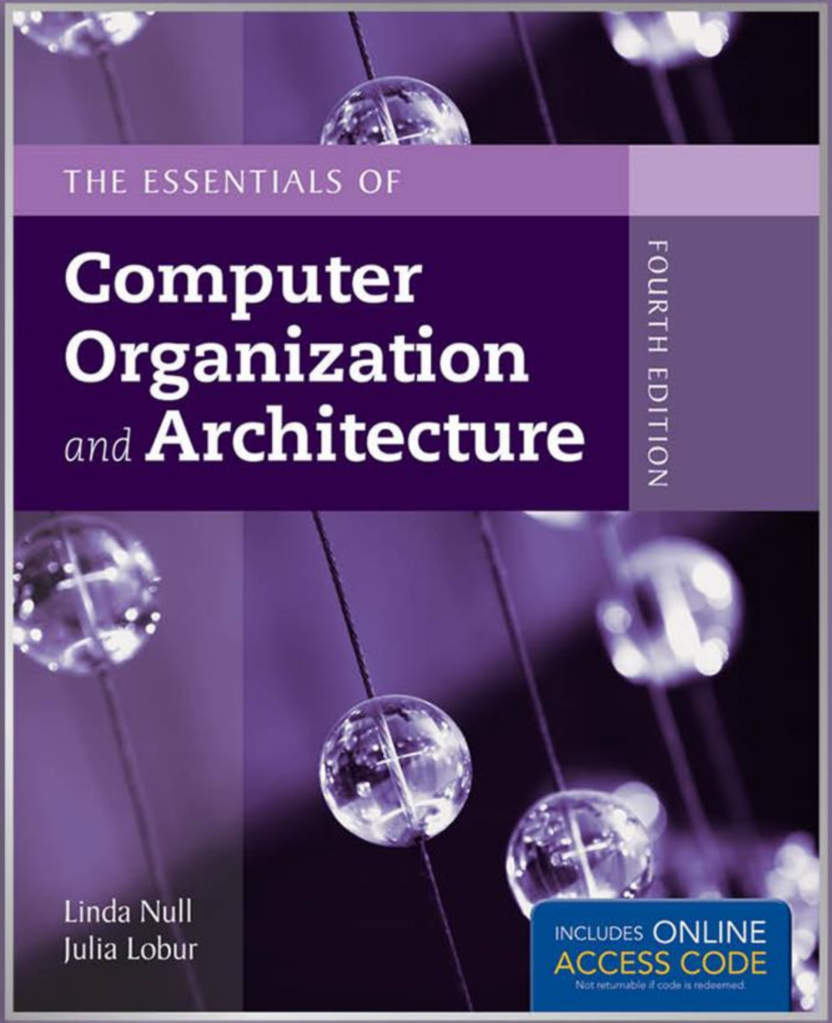
Chapter 2

Data Representation in Computer Systems



Chapter 2 Objectives

- Understand the concepts of error detecting and correcting codes.

2.8 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.

2.8 Error Detection and Correction

- Check digits, appended to the end of a long number, can provide some protection against data input errors.
  - The last characters of UPC barcodes and ISBNs are check digits.
- Longer data streams require more economical and sophisticated error detection mechanisms.
- Cyclic redundancy checking (CRC) codes provide error detection for large blocks of data.

2.8 Error Detection and Correction

- Checksums and CRCs are examples of *systematic error detection*.
- In *systematic error detection* a group of error control bits is appended to the end of the block of transmitted data.
  - This group of bits is called a *syndrome*.
- CRCs are polynomials over the modulo 2 arithmetic field.

The mathematical theory behind modulo 2 polynomials is beyond our scope. However, we can easily work with it without knowing its theoretical underpinnings.

2.8 Error Detection and Correction

- Modulo 2 arithmetic works like clock arithmetic.
- In clock arithmetic, if we add 2 hours to 11:00, we get 1:00.
- In modulo 2 arithmetic if we add 1 to 1, we get 0. The addition rules couldn't be simpler:

$$0 + 0 = 0 \qquad 0 + 1 = 1$$
$$1 + 0 = 1 \qquad 1 + 1 = 0$$

You will fully understand why modulo 2 arithmetic is so handy after you study digital circuits in Chapter 3.

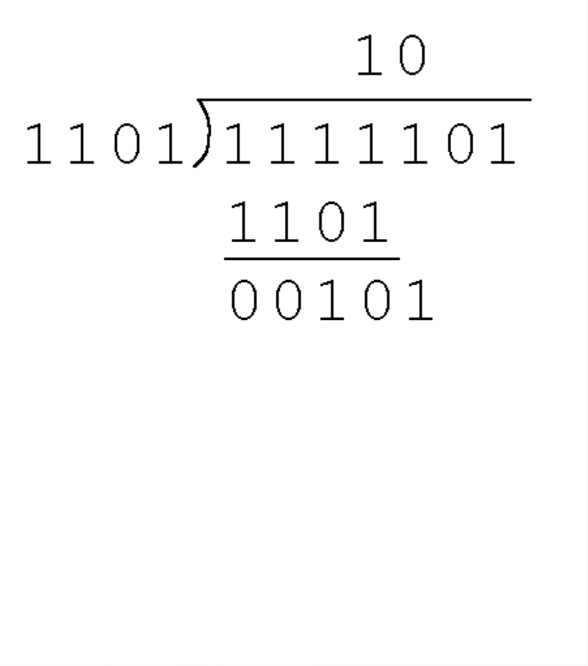
2.8 Error Detection and Correction

- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic.**
  - As with traditional division, we note that the dividend is divisible once by the divisor.
  - We place the divisor under the dividend and perform modulo 2 subtraction.

```
           1
1101)1111101
     1101
     ----
     0010
```

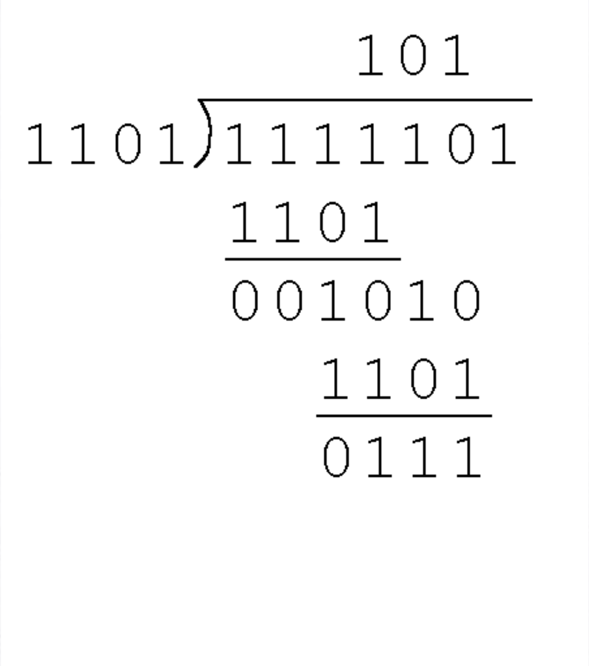
2.8 Error Detection and Correction

- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…**
  - Now we bring down the next bit of the dividend.
  - We see that 00101 is not divisible by 1101. So we place a zero in the quotient.



2.8 Error Detection and Correction

- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…**
  - 1010 is divisible by 1101 in modulo 2.
  - We perform the modulo 2 subtraction.



2.8 Error Detection and Correction

- **Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…**
  - We find the quotient is 1011, and the remainder is 0010.
- **This procedure is very useful to us in calculating CRC syndromes.**

```
           1 0 1 1
      _______________
1 1 0 1 ) 1 1 1 1 1 0 1
          1 1 0 1
          -------
          0 0 1 0 1 0
              1 1 0 1
              -------
              0 1 1 1 1
                1 1 0 1
                -------
                0 0 1 0
```

Note: The divisor in this example corresponds to a modulo 2 polynomial: $X^3 + X^2 + 1$.

2.8 Error Detection and Correction

- **Suppose we want to transmit the information string: 1111101.**
- **The receiver and sender decide to use the (arbitrary) polynomial pattern, 1101.**
- **The information string is shifted left by one position less than the number of positions in the divisor.**
- **The remainder is found through modulo 2 division (at right) and added to the information string: 1111101000 + 111 = 1111101111.**

```
           1011011
        __________
 1101 ) 1111101000
        1101
        ----
        001010
          1101
          ----
          01111
           1101
           ----
           001000
             1101
             ----
             01010
              1101
              ----
              0111
```

2.8 Error Detection and Correction

- **If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of zero.**
- **We see this is so in the calculation at the right.**
- **Real applications use longer polynomials to cover larger information strings.**
  - Some of the standard poly-nomials are listed in the text.

```
        1011011
     ----------
1101)1111101111
     1101
     ----
     001010
       1101
       ----
       01111
        1101
        ----
        001011
          1101
          ----
          01101
           1101
           ----
           0000
```

2.8 Error Detection and Correction

- Data transmission errors are easy to fix once an error is detected.
  - Just ask the sender to transmit the data again.
- In computer memory and data storage, however, this cannot be done.
  - Too often the only copy of something important is in memory or on disk.
- Thus, to provide data integrity over the long term, error *correcting* codes are required.

2.8 Error Detection and Correction

- Hamming codes and Reed-Solomon codes are two important error correcting codes.
- Reed-Solomon codes are particularly useful in correcting *burst errors* that occur when a series of adjacent bits are damaged.
  - Because CD-ROMs are easily scratched, they employ a type of Reed-Solomon error correction.
- Because the mathematics of Hamming codes is much simpler than Reed-Solomon, we will only look at Hamming codes.

2.8 Error Detection and Correction

- For the word 11010110, assuming even parity,
  - Bit 1 *contributes* to bits 3, 5, 7, 9, and 11, so its value is 1 to ensure even parity within this group.
  - Bit 2 *contributes* to bits 3, 6, 7, 10, and 11, so its value is 0.

				P8				P4		P2	P1
1	1	0	1		0	1	1		0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

What are the values for the other parity bits?

2.8 Error Detection and Correction

1	1	0	1	1	0	1	1	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

- The completed code word is shown above.

```
P1 = XOR of bits (3,5,7,9,11)
      = xor(0,1,0,1,1) = 1
P2 = XOR of bits (3,6,7,10,11)
      = xor(0,1,0,0,1) = 0
P4 = XOR of bits (5,6,7,12)
      = xor(1,1,0,1) = 1
P8 = XOR of bits (9,10,11,12)
     = xor(1,0,1,1) = 1
```

- Using the Hamming algorithm, we can not only detect single bit errors in this code word, but also correct them!

2.8 Error Detection and Correction

1	1	0	1	1	0	1	0	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

- Suppose an error occurs in bit 5, as shown above. Our parity bit values are:
  - Bit 1 checks 1, 3, 5, 7, 9, and 11. *This is incorrect as we have a total of 3 ones (which is not even parity).*
  - Bit 2 checks bits 2, 3, 6, 7, 10, and 11. The parity is correct.
  - Bit 4 checks bits 4, 5, 6, 7, and 12. *This parity is incorrect, as we 3 ones.*
  - Bit 8 checks bit 8, 9, 10, 11, and 12. This parity is correct.

2.8 Error Detection and Correction

1	1	0	1	1	0	1	0	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

- We have erroneous parity for check bits 1 and 4.
- With *two* parity bits that don't check, we know that the error is in the data, and not in a parity bit.
- Which data bits are in error? We find out by adding the bit positions of the erroneous bits.
- Simply, 1 + 4 = 5. This tells us that the error is in bit 5. If we change bit 5 to a 1, all parity bits check and our data is restored.

Hamming Code

- Single-error detection & correction

Bit Position											
1	2	3	4	5	6	7	8	9	10	11	12
1	1	0	0	0	1	0	0				
P1	P2	1	P4	1	0	0	P8	0	1	0	0
0	0	1	1	1	0	0	1	0	1	0	0

P1 = XOR of bits (3,5,7,9,11)
= xor(1,1,0,0,0) = 0
P2 = XOR of bits (3,6,7,10,11)
= xor(1,0,0,1,0) = 0
P4 = XOR of bits (5,6,7,12)
= xor(1,0,0,0) = 1
P8 = XOR of bits (9,10,11,12)
= xor(0,1,0,0) = 1

Range of Data Bits for *k* Check Bits

- For *k* check bits and *n* data bits
  - Total bits = n + k
  - Syndrome values has a range: $0 - (2^k-1)$
  - Reserve syndrome value '0'; thus, we are left with (2^k-1) values
  - To check all data bits
    $(2^k-1) \geq n + k$ OR $(2^k-1) - k \geq n$

Number of Check Bits, *k*	Range of Data Bits, *n*
3	2-4
4	5-11
5	12-26
6	27-57
7	58-120

Single-Error Correction, Double-Error Detection

- Add 1 more parity bit at the end
  - In previous example, P13
  - Data bits becomes: “001110010100P_{13}”
- To compute P13, XOR all 12 bits
  - Even parity
  - To check, the parity *P* over all 13 bits must be 0 (correct; even parity)
- Scenarios:
  - C = 0 and P = 0 ➔ no error
  - C ≠ 0 and P = 1 ➔ single error (detect & correct)
  - C ≠ 0 and P = 0 ➔ double error (detect only)
  - C = 0 and P = 1 ➔ single error at P13