2.7 Error Detection and Correction

Parity

Even parity: Add a bit to make the number of ones (1s) transmitted even.

Odd parity: Add a bit to make the number of ones (1s) transmitted odd.

Example and ASCII A is coded 100 0001

	Parity	ASCII 'A'	Binary	Hex
Even parity:	0	100 0001	0100 0001	41
Odd parity:	1	100 0001	1100 0001	C1

In general, the error correction-detection bits are called a ***syndrome***, they are ***redundant***, they are computed so both transmitter and receiver can compute the syndrome independent of a ***noisy*** transmission line.

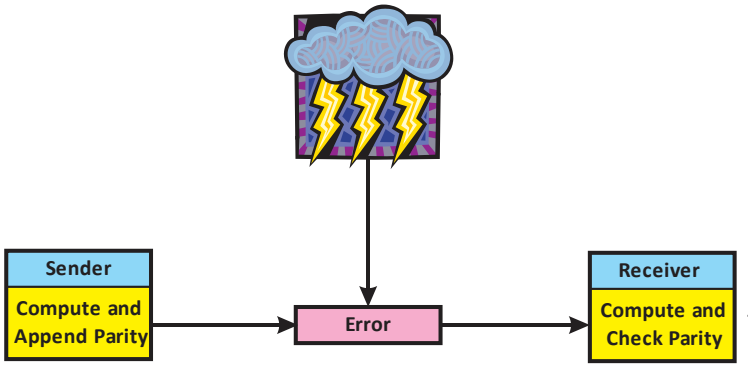


Figure 1-1. Sender, Receiver and a Noisy Transmission Line.

2.7.1 Cyclic Redundancy Check

- 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:

$$0 + 0 = 0 \quad 0 + 1 = 1$$

$$1 + 0 = 1 \quad 1 + 1 = 0$$

Modulo 2 Arithmetic

Example 2.38: Find the sum of 1011_2 and 110_2 modulo 2.

$$\begin{array}{cccccc} & & 1 & 0 & 1 & 1 \\ + & & & 1 & 1 & 0 \\ \hline & & 1 & 1 & 0 & 1 \end{array}$$

Example 2.39: Find the quotient and remainder when 1001011_2 is divided by 1011_2 using modulo 2 arithmetic.

$$\begin{array}{cccc|ccccccc} & & & & & & & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ & & & & 1 & 0 & 1 & 1 & & & \\ \cline{5-8} & & & & 0 & 0 & 1 & 0 & 0 & 1 & \\ & & & & & & 1 & 0 & 1 & 1 & \\ \cline{7-10} & & & & & & 0 & 0 & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array}$$

- 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  0  1  1
1  1  0  1 | 1  1  1  1  1  0  1  0  0  0
           | 1  1  0  1
           -------------
              1  0  1  0
              1  1  0  1
              ----------
                 1  1  1  1
                 1  1  0  1
                 ----------
                    1  0  0  0
                    1  1  0  1
                    ----------
                       1  0  1  0
                       1  1  0  1
                       ----------
                          0  1  1  1
```

- 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.
- 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 below.

```
                    1  0  1  1  0  1
1  1  0  1 | 1  1  1  1  1  0  1  1  1  1
           | 1  1  0  1
           -------------
              1  0  1  0
              1  1  0  1
              ----------
                 1  1  1  1
                 1  1  0  1
                 ----------
                    1  0  1  1
                    1  1  0  1
                    ----------
                       1  1  0  1
                       1  1  0  1
                       ----------
                          0  0  0  0
```

- Real applications use longer polynomials to cover larger information strings.
  - Some of the standard polynomials are listed in the text.

2.7.1 Hamming Codes

- 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.

- Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.
- The *Hamming distance* between two code words is the number of bits in which two code words differ.

This pair of bytes has Hamming distance of 3:	1	0	0	0	1	0	0	1
	1	0	1	1	0	0	0	1

- The minimum Hamming distance for a code is the smallest Hamming distance between ***all*** pairs of words in the code.
- The minimum Hamming distance for a code, $D(min)$, determines its error detecting and error correcting capability.
- For any code word, X, to be interpreted as a different valid code word, Y, at least $D(min)$ single-bit errors must occur in X.
- Thus, to detect k (or fewer) single-bit errors, the code must have a Hamming distance of $D(min) = k + 1$.
- Hamming codes can detect

  Detect $\boldsymbol{D(min) - 1}$

  Correct $\boldsymbol{\left\lfloor \frac{D(min) - 1}{2} \right\rfloor}$

- Thus, a Hamming distance of 2*k* + 1 is required to be able to correct *k* errors in any data word.
- Hamming distance is provided by adding a suitable number of parity bits to a data word.

- Suppose we have a set of n-bit code words consisting of m data bits and r (redundant) parity bits.
- Suppose also that we wish to detect and correct one single bit error only.
- An error could occur in any of the n bits, so each code word can be associated with n invalid code words at a Hamming distance of 1.

- Therefore, we have $n + 1$ bit patterns for each code word: one valid code word, and n invalid code words
- Using n bits, we have 2^n possible bit patterns. We have 2^m valid code words with r check bits (where $n = m + r$).
- For each valid codeword, we have $(n + 1)$ bit patterns (1 legal and n illegal).
- This gives us the inequality:

$$(n + 1) \times 2^m \leq 2^n$$

- Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^m \leq 2^{m+r}$$

or

$$\boldsymbol{(m + r + 1) \leq 2^r}$$

  - This inequality gives us a lower limit on the number of check bits that we need in our code words.

- Suppose we have data words of length $m = 4$. Then:

$$(4 + r + 1) \leq 2^r$$

  implies that r must be greater than or equal to 3.
  - *We should always use the smallest value of r that makes the inequality true.*
- This means to build a code with 4-bit data words that will correct single-bit errors, we must add 3 check bits.
- Finding the number of check bits is the hard part. The rest is easy.

- Suppose we have data words of length $m = 8$. Then:

$$(8 + r + 1) \leq 2^r$$

  implies that r must be greater than or equal to 4.
- This means to build a code with 8-bit data words that will correct single-bit errors, we must add 4 check bits, creating code words of length 12.

- So how do we assign values to these check bits?

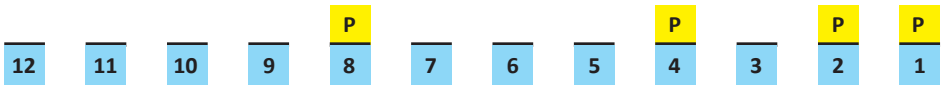
- With code words of length 12, we observe that each of the bits, numbered 1 through 12, can be expressed in powers of 2. Thus:
  - $1 = 2^0$
  - $2 = 2^1$
  - $3 = 2^1 + 2^0$
  - $4 = 2^2$
  - $5 = 2^2 + 2^0$
  - $6 = 2^2 + 2^1$
  - $7 = 2^2 + 2^1 + 2^0$
  - $8 = 2^3$
  - $9 = 2^3 + 2^0$
  - $10 = 2^3 + 2^1$
  - $11 = 2^3 + 2^1 + 2^0$
  - $12 = 2^3 + 2^2$

    - 1 (= 2^0) contributes to all of the odd-numbered digits, including digits 1, 3, 5, 7, 9, and 11.
    - 2 (= 2^1) contributes to the digits, 2, 3, 6, 7, 10, and 11.
    - 4 (= 2^2) contributes to the digits, 4, 5, 6, 7, and 12
    - 8 (=2^3) contributes to the digits 8, 9, 10, 11, and 12

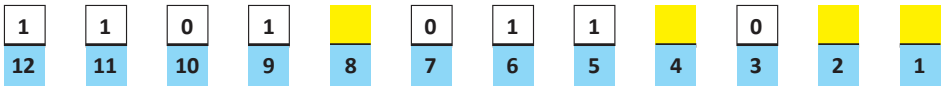
2^3	2^2	2^1	2^0				
0	0	0	1	1			
0	0	1	0	2	2		
0	0	1	1	3	3		
0	1	0	0	4		4	
0	1	0	1	5		5	
0	1	1	0	6	6	6	
0	1	1	1	7	7	7	
1	0	0	0	8			8
1	0	0	1	9			9
1	0	1	0	10	10		10
1	0	1	1	11	11		11
1	1	0	0	12		12	12
1	1	0	1	13		13	13
1	1	1	0	14	14	14	14
1	1	1	1	15	15	15	15

- We can use this idea in the creation of our check bits.
- Using our code words of length 12, number each bit position starting with 1 in the low-order bit.
- Each bit position corresponding to a power of 2 will be occupied by a check bit.
- These check bits contain the parity of each bit position for which it participates in the sum.

- Parity will be stored in bits whose position is an even power of 2, in bit positions 1, 2, 4, 8, …

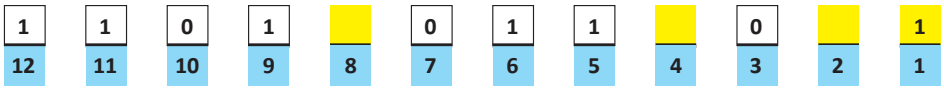


- Since 1 (=2^0) contributes to the values 1, 3 , 5, 7, 9, and 11, bit 1 will check parity over bits in these positions.
- Since 2 (= 2^1) contributes to the values 2, 3, 6, 7, 10, and 11, bit 2 will check parity over these bits.
- Since 4 (= 2^2) contributes to the values 4, 5, 6, 7, and 12 and 11, bit 4 will check parity over these bits.
- Fill in the data bits, 11010110, starting from the left, in bit position 12, and moving to the right. Put no data bits in the parity positions, marked in yellow.,



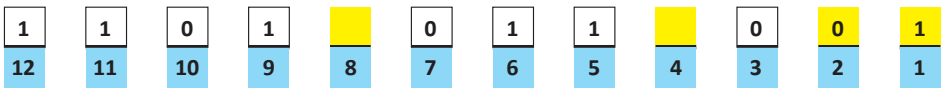
- Now compute the parity bit in position 1. Recall that bit 1 checks bits 3, 5, 7, 9 and 11

1	1	0	1	0	1	Value
11	9	7	5	3	1	Bit Position



- Bit 2 checks the bits 2, 3, 6, 7, 10, and 11, so its value is 0.

1	0	0	1	0	0	Value
11	10	7	6	3	2	Bit Position



- Bit 4 checks the bits 5, 6, 7, and 12, so its value is 1.

1	0	1	1	1	Value
12	7	6	5	4	Bit Position

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

- Bit 8 checks the bits 9, 10, 11, and 12, so its value is also 1.

1	1	0	1	1	Value
12	11	10	9	8	Bit Position

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

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

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

Valid Message

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

Invalid Message

- 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).*

1	1	0	0	0	0	Does not match transmitted parity.
11	9	7	5	3	1	Bit Position

  - Bit 2 checks bits 2, 3, 6, 7, 10, and 11. The parity is correct.

1	0	0	1	0	0	Matches transmitted parity
11	10	7	6	3	2	Bit Position

  - Bit 4 checks bits 4, 5, 6, 7, and 12. *This parity is incorrect, as we 3 ones.*

1	0	1	0	0	Does not match transmitted parity.
12	7	6	5	4	Bit Position

- Bit 8 checks bit 8, 9, 10, 11, and 12. This parity is correct.

1	1	0	1	1	Matches transmitted parity
12	11	10	9	8	Bit Position

- Putting the parity bits adjacent and in descending order we identify the bad bit

0	1	0	1	5, the position of the bad bit
8	4	2	1	Bit Position

- 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 are restored.

2.7.3 Reed-Solomon

- Read for yourself.

3. Conclusion

- Computers store data in the form of bits, bytes, and words using the binary numbering system.
- Hexadecimal numbers are formed using four-bit groups called nibbles.
- Signed integers can be stored in one's complement, two's complement, or signed magnitude representation.
- Floating-point numbers are usually coded using the IEEE 754 floating-point standard.