# A1: Detecting and Correcting Errors

We live in an noisy world. Ultimately, bits are implemented using physical mechanisms that are susceptible to this random noise. The field of coding theory exists to shield us from this noisy world and deliver perfect bits always.

In this assignment, you will implement some simple schemes for error detection and correction that will develop and reinforce your skills for manipulating bits.

## Preliminaries

This assignment must be done ALONE. You may discuss questions and potential solutions with your classmates, but you may not look at their code or their solutions. If in doubt, ask the instructor.

We assume you will use the CSUG machines. You can use other machines, but we will assume your assignment runs correctly on the CSUG machines. Always test on the CSUG machines before submitting.

## Unpacking the assignment

The `a1.zip` file is available in Blackboard. Download it.

Assuming you're using your laptop (or your own machine), you'll need to copy the downloaded file to the CSUG machines. The instructions that follow assume you're going to work on the assignment on the CSUG machines.

To copy the `a1.zip` file to the CSUG machines from your machine, use the command `scp` (in Linux/Mac/Windows) or its equivalent. Usually the command looks like:

```
scp a1.zip username@cycle1.csug.rochester.edu:
```
Then logon to CSUG using `ssh` as usual.

The command to unpack a zip file on Linux, is:

```
unzip a1.zip
```
This will unpack the contents of `a1.zip` on a Linux system. You should see a directory `a1`. Go into the directory, the command is `cd a1`.

If you have difficulty setting up an environment to work on the assignment, reach out to the TAs as soon as possible.

## Smoke tests

Compile the `a1.c` file as follows:

gcc a1.c -o a1

Then run it:

./a1

You should see:

```
FAIL: check_even_parity(0x0): Checking 255 == 1
FAIL: check_even_parity(0x1f): Checking 255 == 0
FAIL: check_even_parity(0x80): Checking 255 == 0
PASS: set_even_parity(0x80): Checking 0 == 0
FAIL: set_even_parity(0x1f): Checking 0 == 1
PASS: set_even_parity(0x0): Checking 0 == 0
PASS: create_mp_code_word(0): Checking 0x0 == 0x0
FAIL: create_mp_code_word(15): Checking 0x0 == 0x7f
FAIL: create_mp_code_word(5): Checking 0x0 == 0x55
FAIL: decode(0x7f): Checking 0 == 15
FAIL: decode(0x7d): Checking 0 == 15
FAIL: decode(0x7e): Checking 0 == 15
FAIL: decode(0x6f): Checking 0 == 15
PASS: create_secded_code_word(0): Checking 0x0 == 0x0
FAIL: create_secded_code_word(15): Checking 0x0 == 0xff
PASS: decode_secded(0): Checking 0x0 == 0x0
FAIL: decode_secded(0xff): Checking 0x0 == 0xf
FAIL: decode_secded(0xff): Checking 0x0 == 0xf
FAIL: decode_secded(0xfc): Checking 0x0 == 0xff
FAIL: decode_secded(0xdd): Checking 0x0 == 0xff
```

Your environment is all setup, and you can edit `a1.c` and solve the assignment.

If you encounter errors reach out to the TAs or post a question on Blackboard.

## Submission

To submit your solution, run `./package.py`. This will give you `a1.zip`. Upload this file to Gradescope. Your assignment will be evaluated for correctness.

The autograder will reject files not prepared using `package.py`.

# Simple parity

The most basic error *detection* scheme is one that uses a single parity bit. Given a bitstring, it's size is extended by one bit. The value of this new bit is chosen so that the number of 1s is even (called an even parity scheme) or odd (called an odd parity scheme).

For example, if you wanted to transmit the values 0 to 15 which need four bits, then using a single parity bit, you would actually transmit 5 bits. Then, if you chose to implement an even parity scheme, the value of the parity bits would be as follows for some values (the parity bit is usually the most significant bit):

| Value | Parity Bit | Transmitted Code Word |
|---|---|---|
| 0000 | 0 | 00000 |
| 0001 | 1 | 10001 |
| 0111 | 1 | 10111 |
| 1111 | 0 | 01111 |

An even parity scheme means that the number of 1s in the transmitted column is always even. Thus, if any single bit flips due to noise, an error will be detected since the number of ones in the received value will be odd.

The parity scheme is used most commonly in serial communications (e.g. Arduino to PC) transmitting 7 bits using 1 bit parity scheme.

A limitation of using a single parity bit is that you can only detect single bit errors. (Can you see why the scheme fails if two bits are corrupted?) Moreover, you cannot correct the error automatically since you cannot identify which bit was corrupted.

## Implementation

Implement the functions `check_even_parity` and `set_even_parity` to compute even parity.

# Single-bit error detection and correction

Using multiple parity bits, it is possible to both detect single-bit errors as well as identify the bit that was corrupted and correct it. In this assignment, we will use transmit the values 0 to 15 using three additional parity bits, storing the values in the format below:

| b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
|---|---|---|---|---|---|---|---|
| X | p3 | p2 | p1 | v4 | v3 | v2 | v1 |

where the values of the parity bits are computed as follows:

```
  p1 = v1 ^ v2 ^ v4
  p2 = v1 ^ v3 ^ v4
  p3 = v2 ^ v3 ^ v4
```

Ignore the value of bit 7.

This scheme can now both detect and correct single bit errors. To see how, assume you receive a corrupted code word. You extract the lower four bits and recompute the parity bits and compare them with the received parity bits. If two or more parity bits are different, then an error has occured, and the bit that was corrupted can be identified using the table below (`X` indicates a difference in the received and recomputed parity bits):

```
| p1 | p2 | p3 | Corrupt bit |
|----|----|----|-------------|
| X  | X  |    | v1          |
| X  |    | X  | v2          |
|    | X  | X  | v3          |
| X  | X  | X  | v4          |
```

(updated: The original instructions for `v2` and `v3` were incorrect)

All you have to do is flip the value of the corrupted bit to obtain the original value.

Can you reason what should happen when only one of the parity bits is wrong?

### Implementation

Implement the `create_mp_code_word` and `decode` functions.

## Single Error Correction, Double Error Detection

Like the simple parity scheme, the multiple parity bits scheme cannot handle two (or more) bits getting corrupted. However, adding an additional parity bit that covers all the bits can enable the detection of double bit errors.

```
  p p3 p2 p1 v4 v3 v2 v1
```

Here `p = p3 ^ p2 ^ p1 ^ v4 ^ v3 ^ v2 ^ v1`.

Now, to detect double bit errors, when you receive a code word `v`, simply decode it as usual to a value, say `w`.

Re-encode `w`, and compare the re-encoded parity bits of `w` to the ones that were received. Let's name the parity bits of `w` as `wp`, `wp1`, `wp2`, `wp3`.

If the parity bits of `v` and `w` are the same, no errors occured, just return the lowest 4 bits of `w`.

If the parity bits are different, but `p` and `wp` (i.e. the overall parity bits) are the same, return the lowest 4 bits of `w`.

Otherwise, an uncorrectable double bit error has occurred, return `255`.

### Implementation

Implement the `create_secded_code_word` and `decode_secded` functions.

## Download

The assignment is available on Blackboard, it is attached to the announcement.