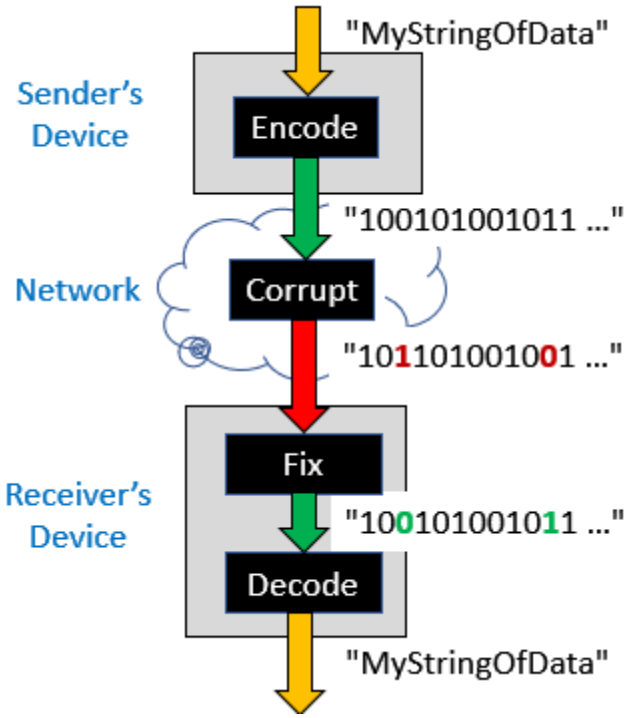
COMP2401 - Assignment #2

(Due: **Friday, October 4, 2019 @ 18:00 (6 PM)**

In this assignment, you will write three programs that involve string manipulation and array manipulation. The assignment will involve implementing a 12-bit Hamming Code encoder and decoder. ALL of your code must be written neatly with proper indentation and have a reasonable amount of comments. If you do not, you will be losing multiple marks.

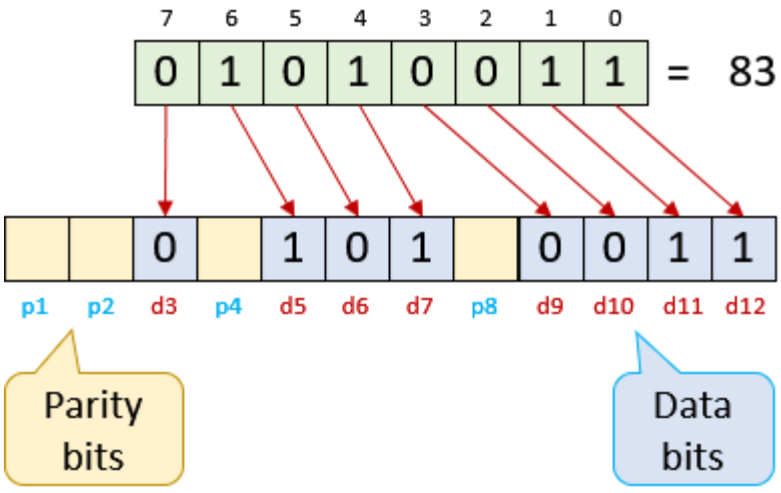
In telecommunications, data gets transmitted back and forth over networks (which may be wireless). When transmitting data, it is possible that the data can be corrupted. We will consider how to detect the occasional corruption of bits and how to repair them by using an error-correcting coding scheme called a ***Hamming Code***. We will consider a 12-bit Hamming Code … where 8 bits are ***data bits*** and 4 bits are known as ***parity bits***. The data bits represent the original data being sent over the network, while the parity bits are extra bits added to detect and repair the errors.



This assignment has multiple parts to it. First, you will write **encode.c** to encode the 8-bit data into 12 bits by adding parity bits. Then you will write **decode.c** to decode the 12-bit data back into 8 bits again. After you get this working, you will run a program given to you (called **corrupt.c**) which will *corrupt* your 12-bit hamming code sequence. Finally, you will *uncorrupt* the sequence by writing a **fix.c** program to identify and repair the corrupted bits.

(1) Encoding

To take an 8-bit byte value (e.g., such as an ASCII character) and encode it into a 12-bit hamming code sequence, you need to place the data and parity bits in specific locations as shown here.



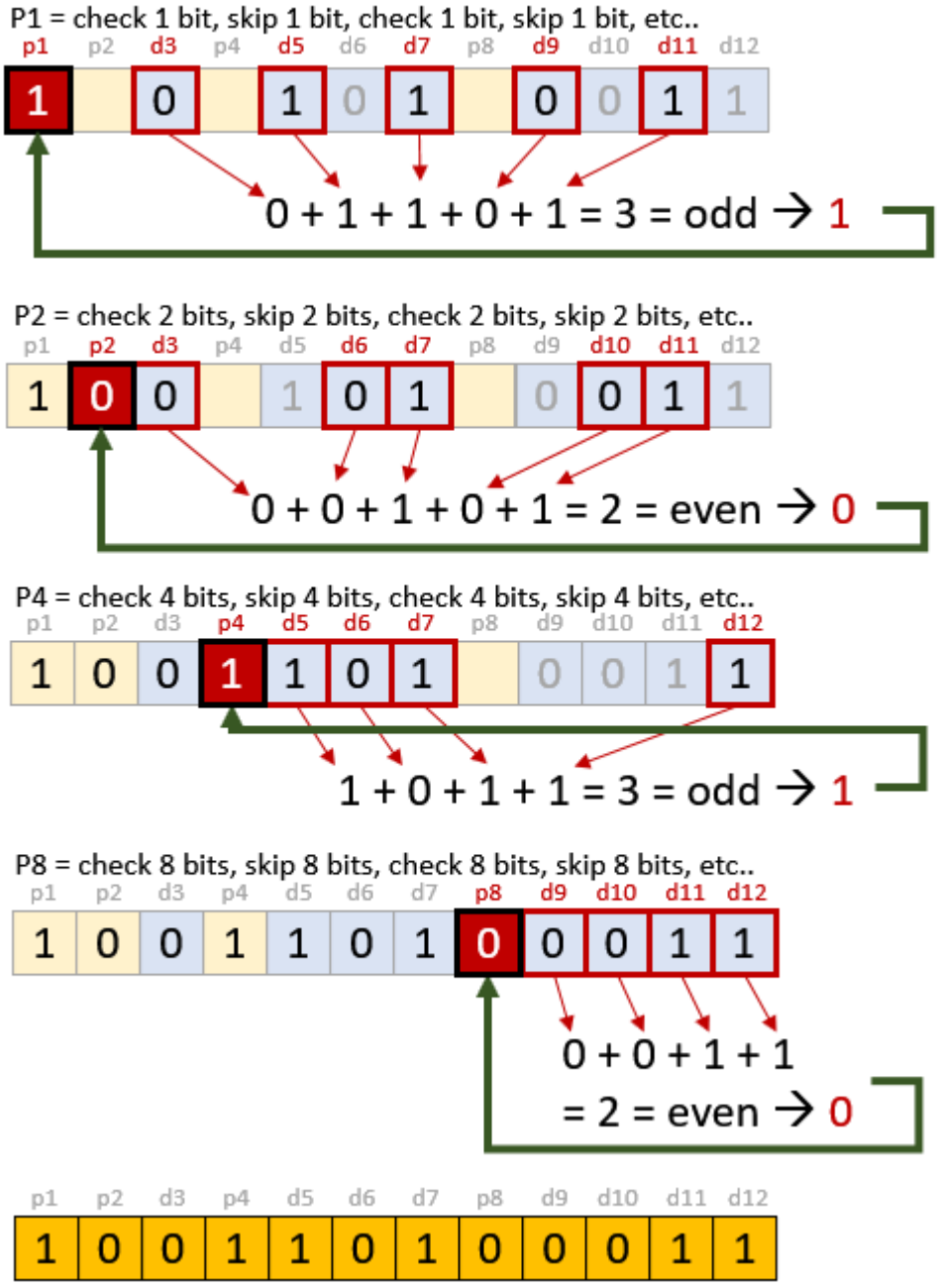
Consider the character '**S**' which has ASCII value **83** whose 8-bits are shown here in green. To convert to a hamming code, we will create an array to hold **12** bits and insert these **8** data bits into the positions indicated in blue. Then, the remaining **4** positions will be the parity bits (in yellow…although the values are not shown here).

Note that the numbering of the original bits is in the opposite direction as the hamming code bits.

Download the **encode.c** program. It contains a **main()** function that prompts the user for a string of (up to CHAR_LIMIT) ASCII characters which will be stored in the **transmitString** array. The code will repeatedly call the **encode()** function to encode each character from **transmitString** into a long bit sequence array called **encodedCharacters** … which is an array of exactly **12*charCount** chars such that each char is either a '**1**' or a '**0**'. Here, for example, is what will be stored if "ABC" is entered:

```
transmitString:      ABC
encodedCharacters:   100010010001010110010010010010000011
```

To do the encoding, you will need to determine the 8 **data** bits of each character and insert these into the **encodedCharacters** array at the correct location. Then you will determine the 4 **parity** bits and insert them into the correct locations as well. Here is how to determine each of the 4 (shown in red) parity bits. They are determined by adding up the data bits (as shown) … and then checking if the sum is odd or even. If the sum is odd, then a '**1**' is used as the parity bit, otherwise a '**0**' is used.



Consider an example. Assume that the user string "ABC" was entered … looking at the ASCII codes, we can determine the data bits (underlined) as well as the parity bits (not underlined) as follows:

65 = ASCII code **01000001** with hamming code sequence = **100010010001**
66 = ASCII code **01000010** with hamming code sequence = **010110010010**
67 = ASCII code **01000011** with hamming code sequence = **010010000011**

The encoded characters array will contain (12***charCount**) characters, plus an extra '\0' (I think that I made the array a little bigger for you just in case). So here is what we would have:

```
encodedCharacters[0] = "100010010001"
encodedCharacters[1] = "010110010010"
encodedCharacters[2] = "010010000011"
etc..
```

But you will not be hard-coding these values like shown here … you need to build up these strings. So, the first 39 chars in the **encodedCharacters** array will be as follows (notice the '\0' chars added):

100010010001\0010110010010\0010010000011\0

The code for storing these strings in the **encodedCharacters** array has been given to you in the **main()** function. You MUST NOT alter the **main()** function . You are just responsible for building up the **bitSequence** array in the **encode()** function. You can treat this **bitSequence** array as if it was a char array of size 13.

Test your code to make sure that it encodes the characters properly before you continue:

```
student@COMPXXXX:~$ ./encode
A
100010010001
student@COMPXXXX:~$ ./encode
ABC
100010010001010110010010010010000011
student@COMPXXXX:~$ ./encode
ThisIsFun
000010110100010011011000010111001001110011100011000010001001110011100011000110000110010011100101110011011110
student@COMPXXXX:~$
```

---

(2) Decoding

Download the **decode.c** program. It contains a **main()** function that prompts the user for a Hamming Code sequence (which is a string of **1**s and **0**s and MUST be a multiple of 12 in length). It then decodes the characters by extracting the data bits and determining which ASCII character it represents. Note that this is the reverse of what has been done in the **encode.c** program:

```
student@COMPXXXX:~$ ./decode
100010010001
A
student@COMPXXXX:~$ ./decode
100010010001010110010010010010000011
ABC
student@COMPXXXX:~$ ./decode
000010110100010011011000010111001001110011100011000010001001110011100011000110000110010011100101110011011110
ThisIsFun
student@COMPXXXX:~$
```

Note that the input string is a multiple of **12** characters, not **13**. That is, there is no **'\0'** character after each 12-bit sequence. The **'\0'** char was only required in **encode.c** for printing purposes. As a result, you can actually copy the output from **encode.c** and paste it as input to **decode.c** for testing. Do this instead of actually typing in all the **1**s and **0**s when you are testing things.

To decode, the **main()** function has been written such that it takes the bit sequence in from the user (as the **recvString** array). It then calls the **decode()** function which takes **12** characters from that string and returns the corresponding ASCII char that it represents. This char is then stored in the **decodedCharacters** array. You MUST NOT alter the **main()** function. Instead, just write the **decode()** function. You can treat the incoming **bitSequence** array as if it was a char array of size 12. You just need to extract the 8 data bits from the sequence and determine the character to return. Test your code before you continue.

---

(3) Corrupting

The **corrupt.c** program has been given to you. Like the **decode.c** program, it will take in a bit sequence in from the user (as the **recvString** array). It then randomly flips one bit in each of the 12-bit sequences. The bit flipped may be a data bit or it may be a parity bit. If a parity bit was flipped, this will not affect the decoded output … as would be shown in the **decode.c** program output. However, if a data bit is flipped, this will alter/corrupt the decoded results. Test this **corrupt.c** program by corrupting some encoded bits and then try decoding to see the result of the corruption.

```
student@COMPXXXX:~$ ./corrupt
100010010001
100011010001
student@COMPXXXX:~$ ./corrupt
100010010001010110010010010010000011
100010010011010100100100100100000001
student@COMPXXXX:~$ ./corrupt
000010110100010011011000010111001001110011100011000010001001110011100011000110000110010011100101110011011110
000010111100010011011010010111101001110011100011010001000100111001110000100011000010010011110101110001011110
student@COMPXXXX:~$
```

You should take the corrupted bit sequences and try them out with the **decode.c** program to see what the result is:

```
student@COMPXXXX:~$ ./decode
100011010001
a
student@COMPXXXX:~$ ./decode
100010010011010100100100100000001
CA
student@COMPXXXX:~$ ./decode
000010111100010011011010010111101001110011100011010001000100111001110000100011000010010011110101110001011110
\jyrIqDu.
student@COMPXXXX:~$
```

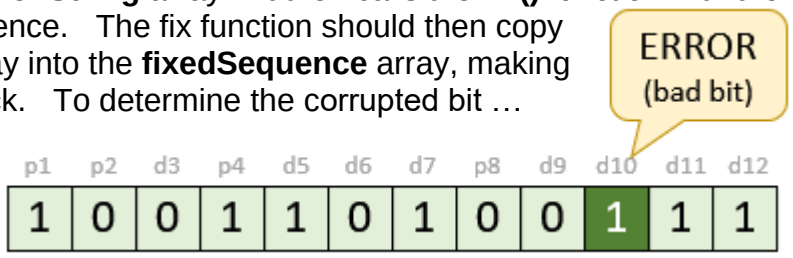
There really is not much else to do in this part of the assignment … no code to write … just test it out. Keep in mind that the code flips random bits … which will differ each time that you run. Also, in some cases, if a parity bit is flipped, then you may not see a change in the decoded output for that character … it may look the same.

---

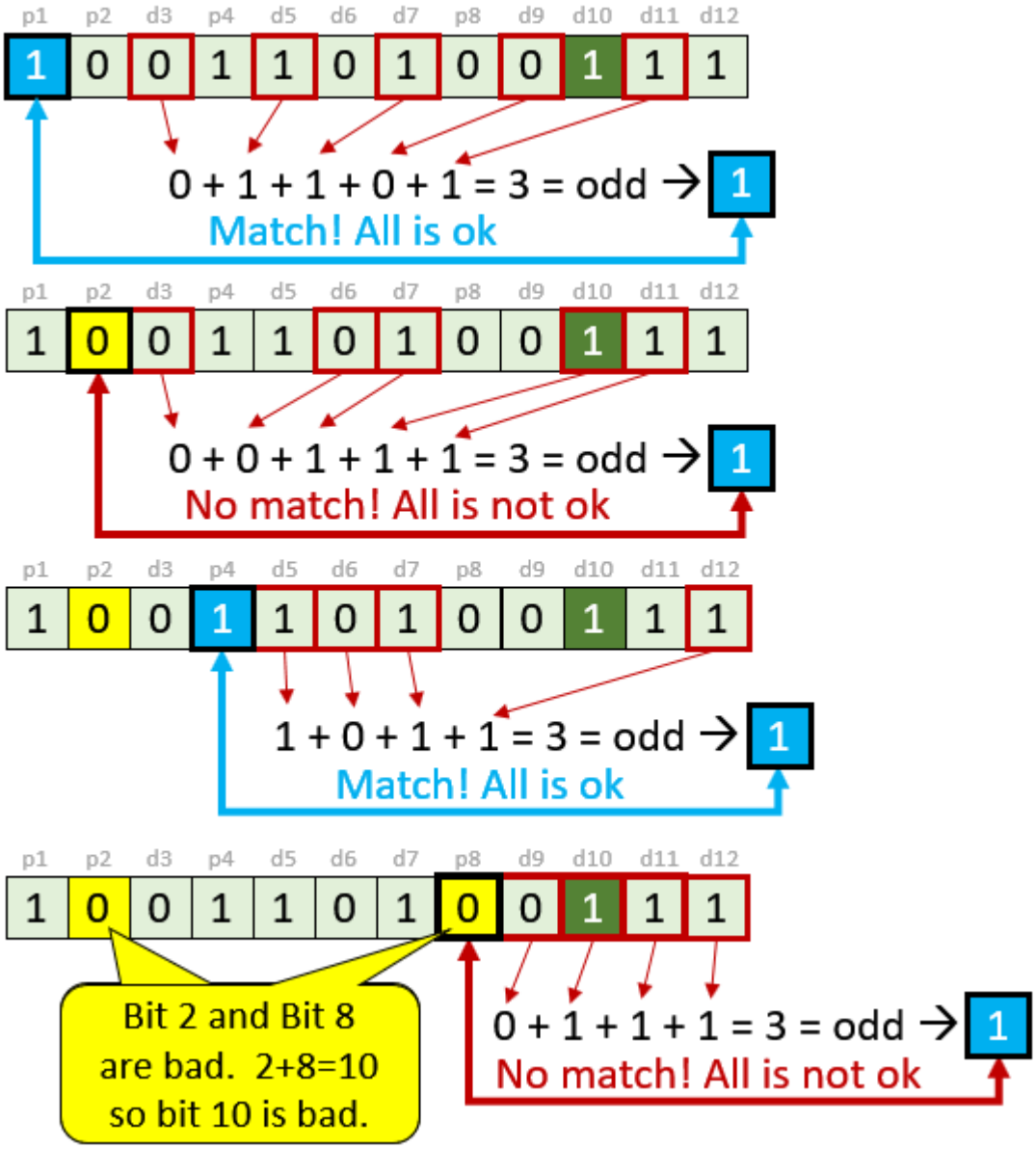
(4) Fixing

Now we will find those corrupted bits and fix them! Download the **fix.c** program. It contains a **main()** function that prompts the user for a corrupted Hamming Code sequence (which is a string of **1**s and **0**s, a multiple of 12 in length and comes from the **corrupt.c** program). It then finds the

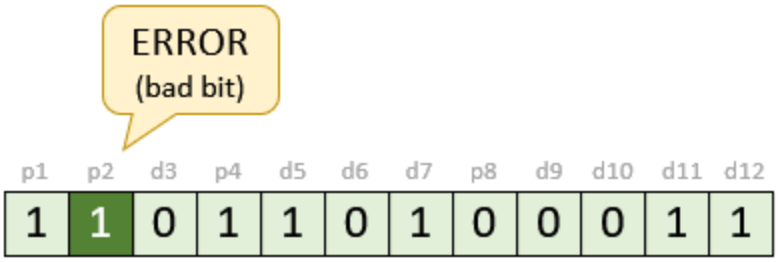
corrupted bit in each 12-bit sequence and fixes it by flipping the bit back. As before, the **main()** function takes in a bit sequence into the **brokenString** array. It then calls the **fix()** function with the 12 character array representing the bit sequence. The fix function should then copy over the bits from the **brokenSequence** array into the **fixedSequence** array, making sure that the broken bit has been flipped back. To determine the corrupted bit … see below.



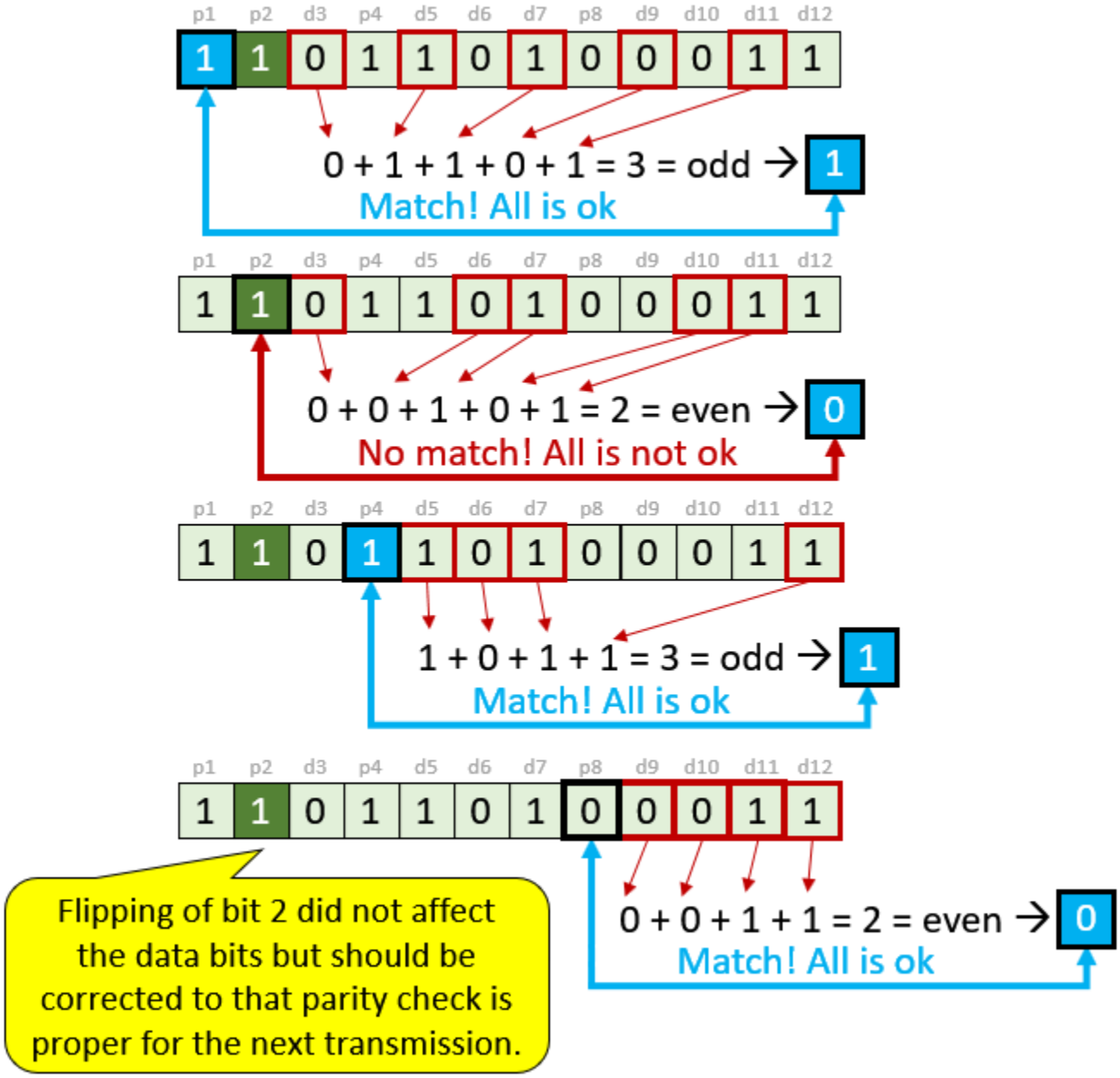
Assume, for example, that data bit **d10** has been flipped. As when encoding, you will need to add up the data bits that corresponds to each parity bit and determine if the total is odd or even. Then you can check if the parity bit matches. Below, this is done for all 4 parity bits. As it turns out, parity bits **p2** and **p8** (shown in yellow) are the ones that do not match. We then just sum up the unmatched parity bit numbers (i.e., 1, 2, 4 or 8). In this case, 2+8=10. This tells us that bit 10 is the bit that was flipped. So … we just need to flip bit 10 back and we are done ☺.



Here is another example, where we corrupted a parity (i.e. **p2**) bit instead of a data bit:



When we check the parity bits … we will find that only p2 does not match. That means that bit 2 is the flipped bit. We might not notice this on a decode operation, since none of the data bits were affected. However, if this bit sequence was to be sent out a second time over the network, we would want to make sure that all the parity bits have been corrected. So we should still flip back bit 2 to 0.



Test your code by using the corrupted bit sequences:

```
student@COMPXXXX:~$ ./fix
100011010001
100010010001
student@COMPXXXX:~$ ./fix
100010010011010100010010010010000001
100010010001010110010010010010000011
student@COMPXXXX:~$ ./fix
000010111100010011011010010111101001110011100010100010001111001110000100001100001000100111101011100010111110
000010110100010011011001001011101001110011100011000100010011100111000110000110000110010011100101110011011110
student@COMPXXXX:~$
```

As a final test, you could take the output of the **fix.c** program and run it through the **decode.c** program and you should get the original ASCII characters that the user encoded.

You can actually use pipes (i.e, | characters … more on this later in the course) to test all programs together, one after another without having to enter user strings each time like this:

```
student@COMPXXXX:~$ ./encode | ./decode
ThisIsATestSentenceForMyCode
ThisIsATestSentenceForMyCode
student@COMPXXXX:~$./encode | ./corrupt | ./decode
ThisIsATestSentenceForMyCode
ThmwHcVu{tSefTenseDosMyCodg
student@COMPXXXX:~$./encode | ./corrupt | ./fix | ./decode
ThisIsATestSentenceForMyCode
ThisIsATestSentenceForMyCode
student@COMPXXXX:~$
```

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files as a single **tar** file containing:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code
2. All of your **.c source** files and all other files needed for testing/running your programs.
3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM, which is **COMP2404A-F19**.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. **You will also lose marks if your code is not written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).
