Information Processing 3 — Week 12 projects

Projects can be completed at any time during the last five weeks of the course. Each project is worth a certain number of points. You can complete as many projects as you like but the total number of points that you can obtain is capped at 50.

1 Parity bits [5 points]

A common error encountered when storing or transmitting data is to find a single bit inverted (a 0 changed to a 1, or a 1 changed to a 0). The simplest way to detect this kind of error is to add one additional bit to each word of data, and to set the additional bit to either 0 or 1 so that the total number of 1s in the word is always even. If a word of data (including its parity bit) is ever encountered that has an odd number of 1s, then one of the bits in the data must have been inverted.

original message	with parity ↓ bits	
01000110	11000110	11000110
00011110	00011110	00011110
01011010	01011010	01011000 ← error
01100001	11100001	11100001
01110000	11110000	11110000
01110000	11110000	11110000
01100001	11100001	11100001

1.1 Project

Implement a pair of functions that can generate and check parity for 7-bit ASCII characters.

`int getParity(int data)` calculates a parity bit for the 8-bit byte `data`. It returns 0 if `data` already contains an even number of 1s, or 1 if `data` contains an odd number of 1s.

`getParity(0b0100100)` → 0
`getParity(0b0110100)` → 1
`getParity(0b0110101)` → 0

`int addParity(int data)` calculates the parity bit for `data` and then returns an integer equal to `data` but with the parity bit added as the most significant bit of the byte, ensuring that the result has an even number of 1s.

`addParity(0b0100000)` → 0b10100000
`addParity(0b0100100)` → 0b00100100
`addParity(0b0110100)` → 0b10110100
`addParity(0b0110101)` → 0b00110101
`addParity(0b1110101)` → 0b11110101

Check your functions using the following program which adds parity bits to a string and then deliberately inverts some of the bits before printing out the string with parity checking.

```
char text[48] = "Peter piper picked a peck of pickled peppers.\n";

for (int i = 0;  i < sizeof(text);  ++i)
    text[i] = addParity(text[i]);

for (int i = 0;  i < sizeof(text);  ++i)
    putchar(getParity(text[i]) ? '?' : text[i] & 0x7F);

text[ 5] ^= (1 << 2);
text[ 7] ^= (1 << 3);
text[11] ^= (1 << 5);
text[13] ^= (1 << 7);
text[17] ^= (1 << 4);

for (int i = 0;  i < sizeof(text);  ++i)
    putchar(getParity(text[i]) ? '?' : text[i] & 0x7F);
```

1.2 Hints

There are many ways to calculate parity for a word of data. One way is to count the number of bits that are set to 1 and return the least significant bit of the result. (This can be very efficient. If you 'and' an integer with itself minus 1 you will remove the least significant 1 from it.)

Another way is to 'exclusive-or' all the bits in the word together. (This can be very efficient too. Three bit-wise 'exclusive-or' operations, three bit shifts, and one 'and' operation are sufficient to calculate the result.)

2 Block parity [15 points]

Parity can work not just 'across' a single byte of data (*lateral* parity), but also between bits in the same position in successive bytes of the message (*longitudinal parity*). For example, 'exclusive-or' between the least significant bit of each byte in the message would calculate a parity bit that detects an error in any of the least significant bits.

For longitudinal parity to be useful we should limit the number of bits that it tries to protect. In the previous question, each parity bit verifies the 7 bits in a single character. We can choose to insert a longitudinal parity word after each seven bytes of the message, to verify the correctness of each bit position for a block of 7 message bytes.

Block parity combines both a parity bit in each byte of message data and a parity word for each block of seven bytes of message data. The parity bits identify damaged bytes and the parity words identify damaged bit positions within their block of bytes.

Knowing *both* the damaged word *and* the damaged bit position allows an error to be *corrected*. An incorrect parity bit in a byte identify it as damaged, and an incorrect bit in the parity word for its block identifies which bit was damaged within the damaged byte.

```
01000110
01010110
01011010
01100001
01110000
01110000     longitudinal
01100001     parity
01001010 ←   word

01000110  1  lateral
01010110  0  parity
01011010  0  bits
01100001  1  …
01110000  1
01010000  1  ← (incorrect)
01100001  1
01001010  1  parity word
   ↑
(incorrect)
```

Block parity can reliably correct only a single-bit error. A second error within the block will either cause a parity error to vanish or will cause two incorrect lateral parity bits and two incorrect longitudinal bit positions, leading to an ambiguous situation. In either case the double-bit error cannot be corrected.

2.1 Project

Write a pair of functions `encodeBlocks()` and `decodeBlocks()`.

`size_t encodeBlocks(byte input[], size_t length, byte output[])` accepts a 7-bit ASCII `input` message of the given `length` and copies it to the `output` array. While copying it adds a parity bit to each byte in the message (in the most significant bit) and after each 7 `input` message bytes it inserts a parity word (every eighth byte in the output array will be a parity word). It should return the number of bytes that were written to the `output` array.

(Assume that the caller has allocated enough space in the `output` array to hold both the original message bytes and the additional parity words that are added to it. You can also assume that the `length` of the message is a multiple of 7 so that the encoded message will always end with a parity word.)

`void decodeBlocks(byte input[], size_t length, byte output[])` accepts an `input` message and copies it to the `output` array. The `input` message consists of 8-bit bytes with parity (in the most significant bit) and also contains a parity word after every seven message bytes (every eighth byte of `input` is a parity word). While copying the data the function should use the parity bits and parity words to repair damaged bytes.

(Assume that the caller has allocated enough space in the `output` array to hold the decoded message. You can also assume that the `length` of the `input` message is a multiple of 8 so that it will always end with a parity word.)

You can use the following program to check your implementation.

```
typedef unsigned char byte;

    byte input [64] = "Peter piper picked a peck of pickled peppers.\n\000";
    byte medium[64];   // the data during transit or storage (possibly with errors)
    byte output[64];   // the data after reception or reading

    int len = encode(input, 48, medium);  // send or store 48 bytes of data

    decode(medium, len, output);          // receive or read some uncorreupted data
    for (int i = 0;  output[i];  ++i)
        putchar(getParity(output[i]) ? '?' : output[i] & 0x7F);

    medium[ 5] ^= (1 << 2);    // introduce some single-bit errors into the data
    medium[14] ^= (1 << 3);    // maximum one error per parity block
    medium[24] ^= (1 << 5);    // this one damages a parity word
    medium[33] ^= (1 << 7);    // this one flips a parity bit
    medium[42] ^= (1 << 4);

    for (int i = 0; i < len;  ++i)      // print corrupted data, for laughs
        if ((i & 7) != 7)               // not a parity word
            putchar(getParity(medium[i]) ? '?' : medium[i] & 0x7F);

    decode(medium, len, output);        // receive/read corrupted data and repair it
    for (int i = 0;  output[i];  ++i)
        putchar(getParity(output[i]) ? '?' : output[i] & 0x7F);
```

My implementation of these functions prints the index and mask of every damaged byte that is repaired, like this:

```
$ ./12-2-block
Peter piper picked a peck of pickled peppers.
Peter?piper p?cked a ?eck of ?ickled ?eppers.
fixed error in byte 5 bit 04
fixed error in byte 13 bit 08
fixed error in byte 21 bit 20
fixed error in byte 29 bit 80
fixed error in byte 37 bit 10
Peter piper picked a peck of pickled peppers.
```

2.2 Hints

You can reuse the functions from the previous question to implement parity for each byte.

If you use `getParity()` on a byte with a correct parity bit the result will always be 0.

If you calculate the expected parity word in `decode` and then 'exclusive-or' it with the parity word stored in the received message, the result will be 0 to indicate no error or will contain a single set bit corresponding to the bit position that was damaged.

Developing `encode` and `decode` in stages makes the problem easier; for example:

1. Make them copy their input to the output unmodified and check the content of `output` is correct.
2. Insert an empty parity word after each seven bytes in `encode` and remove it again in `decode`.
3. Calculate the parity words properly in `encode` and check that they correct when removing them in `decode`.
4. Calculate and parity for each byte in `encode` and check and remove the parity in `decode`.
5. In `decode`, use incorrect parity bits and words to repair single-bit errors.

3 ISBN checksum [10 points]

Almost every book published in the last 50 years has an *International Standard Book Number* (ISBN). ISBNs are 10 digits long for books published before 2007 (ISBN-10) and 13 digits long for books published in 2007 or later (ISBN-13). The last digit in these ISBNs is a *check digit*, calculated from the other digits, that can be used to validate the integrity of the number.

ISBN-10: 0−306−40615−2 ISBN-13: 978−0−306−40615−7

check digit check digit

ISBN-10 check digits are chosen such that the weighed sum of the digits in the number is exactly divisible by 11. The first digit is weighted by $\times 10$, the second by $\times 9$, and so on, down to the last digit d_{10} (the check digit) which is weighted $\times 1$. In other words, if d_i is the i^{th} digit (counting from the left) then d_{10} is chosen such that:

$$\sum_{i=1}^{10} (11-i) \times d_i = 0 \pmod{11}$$

To calculate the check digit:

1. let s be the weighted sum of the first 9 digits, $s = \sum_{i=1}^{9} (11-i) \times d_i$
2. let t be the distance between s and a multiple of 11: $t = 11 - (s \bmod 11)$
3. since s might be 0, and therefore t might be 11, let the check digit d be: $d = t \bmod 11$

For example, to generate the ISBN-10 check digit for: 0−306−40615−2

1. $s = (10 \times 0) + (9 \times 3) + (8 \times 0) + (7 \times 6) + (6 \times 4) + (5 \times 0) + (4 \times 6) + (3 \times 1) + (2 \times 5) = 130$
2. $t = 11 - (130 \bmod 11) = 11 - 9 = 2$
3. $d = 2 \bmod 11 = 2$

If the check digit d is equal to 10 then it is represented by the character 'X'.

ISBN-13 check digits are chosen such that the weighted sums of the 13 digits is exactly divisible by 10. The weights of the digits alternate between 1 and 3.

$$d_1 + 3d_2 + d_3 + 3d_4 + d_5 + 3d_6 + d_7 + 3d_8 + d_9 + 3d_{10} + d_{11} + 3d_{12} + d_{13} = 10 \pmod{10}$$

To calculate the check digit:

1. let s be the weighted sum of the first 12 digits
2. let t be the distance between s and a multiple of 10: $t = 10 - (s \bmod 10)$
3. since s might be 0, and therefore t might be 10, let the check digit d be: $d = t \bmod 10$

For example, to generate the ISBN-13 check digit for: `978-0-306-40615-7`

1. $s = 9 + (7 \times 3) + 8 + (0 \times 3) + 3 + (0 \times 3) + 6 + (4 \times 3) + 0 + (6 \times 3) + 1 + (5 \times 3) = 93$
2. $t = 10 - (93 \bmod 10) = 10 - 3 = 7$
3. $d = 7 \bmod 10 = 7$

3.1 Project

Write a program that reads lines containing digits (or 'X') from standard input. While reading the digits it should ignore spaces and dashes '-'. At the first non-digit, non-space, non-dash character the program should check how many digits were read.

- If the line begins with 9 digits then the program calculates the ISBN-10 check digit and then prints the entire 10-digit ISBN.
- If the line begins with 10 digits then the program verifies the ISBN-10 check digit and then prints the ISBN followed by either "`ok`" or "`invalid`".
- If the line begins with 12 digits then the program calculates the ISBN-13 check digit and then prints the entire 13-digit ISBN.
- If the line begins with 13 digits then the program verifies the ISBN-13 check digit and then prints the ISBN followed by either "`ok`" or "`invalid`".

If the line contains anything else then the program should print "`Not an ISBN: `" followed by the entire contents of the line that was read.

A file called '`12-3-isbn-test.in`' has been provided to help check that your program is working. Another file called '`12-3-isbn-test.out`' has been provided that contains the expected output.

3.2 Hints

A single function can both generate and verify check digits. To generate the check digit, calculate it using the first 9 (ISBN-10) or 12 (ISBN-13) digits of the ISBN. To verify the check digit, calculate the check digit using all 10, or 13, digits. The result (if the check digit is correct) should be zero, because the sum of all the digits will be exactly divisible by 11 (ISBN-10) or 10 (ISBN-13).

For example, `int isbn10(char *digits, int n)` can calculate a check digit (between `'0'` and `'X'`) for `n` single `digits`. If `n` is 9 then `digits` does not include the check digit and the result will be the required check digit that should be appended to the ISBN. If `n` is 10 then `digits` includes the check digit and the result will be `'0'` if the check digit (included in the calculation) is correct. (For this reason you should be prepared to read `'X'` as one of the incoming digits, with a value of 10).

The same applies to a function `int isbn13(char *digits, int n)`.

4 Cyclic redundancy check [15 points]

A *cyclic redundancy check* (CRC) is a checksum used in networks and disk drives to detect data corruption. A CRC value is calculated and attached to every block of data as it is transmitted. At the receiver, the CRC value is re-calculated and compared to the one sent by the transmitter to detect damage.

CRC-32 is the algorithm used to check the integrity of packets sent over an Ethernet network. It is designed so that the checksum is very easy to calculate as data is being transmitted or received one bit at a time.

CRC-32 treats the outgoing data as a huge binary dividend and then calculates the remainder when that number is divided by the 33-bit divisor `0x104C11DB7`. The division is done similarly to the way 'long division' is done by hand, by repeatedly subtracting the divisor from the dividend. The 'subtraction' ignores borrows between digits and is implemented using an 'exclusive-or' operation. This allows it to be implemented one bit at a time.

1. A 32-bit `crc` is initialised with the value `0xFFFFFFFF`.
2. For each single bit b that is transmitted:
   - b is 'exclusive-or'ed with the most significant bit of the `crc`.
   - If the result is 0 then
     - `crc` is shifted left one bit.
   - Else, if the result is 1 then
     - `crc` is set to `crc` shifted left one bit and 'exclusive-or'ed with `0x04C11DB7`.
3. At the end of the transmission the final CRC value is calculated from `crc` as follows:
   - `crc` is set to its one's complement (each bit is inverted).
   - The order of the bits in `crc` is reversed.

Note that bytes are sent over a network least significant bit first and so the CRC calculation should process the bits in order from the least significant (bit 0) to the most significant (bit 7).

4.1 Project

Write some functions that calculate CRC values for strings, blocks of data, and the contents of files.

`unsigned int crcData(char *data, size_t length)` calculates a CRC from the `length` bytes of data stored at address `data`.

`unsigned int crcString(char *string)` calculates a CRC from the bytes in the `string`.

`unsigned int crcPath(char *filename)` calculates a CRC from the contents of the file with the given `filename`.

You can test your functions with the following program.

```
int main(int argc, char **argv) {
    printf("%08x\n", crcData("", 0));
    printf("%08x\n", crcData("\xff\xff\xff\xff", 4));
    printf("%08x\n", crcData("ABC", 3));
    printf("%08x\n", crcData("123456789", 9));
    printf("%08x\n", crcString("It is easier to fool the people than it is "
                               "to convince them that they have been fooled."));

    for (int i = 1;  i < argc;  ++i)
        printf("%08x %s\n", crcPath(argv[i]), argv[i]);
```

```
    return 0;
}
```

The results should be as follows:

```
$ ./12-4-crc 12-3-isbn-test.in
00000000
ffffffff
a3830348
cbf43926
1ed46cc2
3b31214e 12-3-isbn-test.in
```

4.2 Hints

One way to structure your code is to write one function for each 'level' in the hierarchy of processing that the above program needs to calculate the various CRC values.

- `crcBit(int b)` could process a single bit `b` as described in the algorithm above.
- `crcByte(int b)` could process each bit in `b`, in the correct order, using `crcBit(b >> n)`.
- `unsigned int crcData(char *data, size_t len)` could initialise the `crc` value, process each of the `length` bytes `data` using `crcByte()`, then finalise the CRC value as described in the algorithm above.
- `unsigned int crcString(char *string len)` could process the contents of the `string` using `crcData()`.
- `unsigned int crcFile(FILE *file)` could initialise the `crc` value, process each byte in the `file`, then finalise the CRC as described above.
- `unsigned int crcPath(char *filename)` could open the `filename`, process the contents with `crcFile()`, and then close the file.

To finalise the CRC you need one's complement and a function to reverse all the bits in a 32-bit integer. One's complement is trivial as it is a built-in operator in C. Reversing the bits in a 32-bit integer can be done easily in five steps using shifts, 'and's and 'or's. To reverse an integer `i`:

- Swap the two 16-bit halves of `i` using two bit shifts, two 'and's, and an 'or'; store the result back in `i`.
- Swap adjacent bytes in `i` using two bit shifts, two 'and's, and an 'or'; store the result back in `i`.
- Swap adjacent nybbles in `i` using two bit shifts, two 'and's, and an 'or'; store the result back in `i`.
- Swap adjacent pairs of bits in `i` using two bit shifts, two 'and's, and an 'or'; store the result back in `i`.
- Swap adjacent bits in `i` using two bit shifts, two 'and's, and an 'or'; store the result back in `i`.

The bits in `i` have now been reversed. (Of course, any other mechanism that you want to implement will be fine too.)

If you are interested in the mathematics behind the CRC, a gentle introduction can be found here:

`https://www.lammertbies.nl/comm/info/crc-calculation`

5 Error correction with Hamming codes [20 points]

Computer memory is not reliable. Some estimates suggest that about one bit per hour per gigabyte of memory is flipped by high-energy particles ('cosmic rays') arriving from outer space. Mission-critical computers such as database and storage servers use *error correcting code* (ECC) memory that can repair a single-bit error in a word of data when it is read from memory.

A *Hamming code* is a multi-bit parity code. Each word in memory stores both N data bits and M parity bits. Each bit position in the word is assigned a number, between 1 and $N+M$. The parity bits each cover about half of the bits in the word. They are arranged so that when a single bit error occurs the incorrect parity bits will form a binary number equal to the bit position of the damaged bit. This works even if the damaged bit is one of the parity bits.

To see how the numbering must work, first consider the parity bits. If one of them is damaged then it must identify itself and no other bit. We therefore must number them with powers of 2. Bit positions 1, 2, 4, 8, etc., will therefore be where the parity bits are stored.

If two parity bits are incorrect then they must sum to the number of the incorrect bit. For example, if parity bits numbered 1 and 4 are incorrect, then bit number 5 must be the one that is faulty. We therefore require bit 5 to be included in the calculation of both parity bit 1 and parity bit 4, but none of the other parity bits. It should now be obvious that each bit position must be included in the calculation of each parity bit that is needed to describe that position as a binary number. For example, bit position 12 must be included in the calculations of parity bits numbered 4 and 8, because $4+8=12$.

Following these rules, here is how the 12 bits that are required to store an 8-bit byte must be arranged. The parity bits are called p_0 to p_3 and the data bits are called d_0 to d_7. The arrangement is shown both horizontally and vertically, to make the patterns easier to see.

bit number:	12	11	10	9	8	7	6	5	4	3	2	1
contents:	d_7	d_6	d_5	d_4	p_3	d_3	d_2	d_1	p_2	d_0	p_1	p_0
p_0		o		o		o		o		o		o
p_1		o	o			o	o			o	o	
p_2	o					o	o	o	o			
p_3	o	o	o	o	o							

		p_3	p_2	p_1	p_0
1	p_0				o
2	p_1			o	
3	d_0			o	o
4	p_2		o		
5	d_1		o		o
6	d_2		o	o	
7	d_3		o	o	o
8	p_3	o			
9	d_4	o			o
10	d_5	o		o	
11	d_6	o		o	o
12	d_7	o	o		

To store the byte `0b01111010` we would calculate the parity bits as follows:

bit number:	12	11	10	9	8	7	6	5	4	3	2	1
contents:	0	1	1	1	p_3	1	0	1	p_2	0	p_1	p_0
p_0		1		1		1		1		0		0
p_1		1	1			1	0			0	1	
p_2	0					1	0	1	0			
p_3	0	1	1	1	1							

The final 12-bit word stored in memory would be: `011111010010`

Consider the result of bit 6 being flipped by a 'cosmic ray'. While reading the word, a parity check would identify two incorrect parity bits:

bit number:	12	11	10	9	8	7	6	5	4	3	2	1	
contents:	0	1	1	1	p_3	1	1	1	p_2	0	p_1	p_0	
p_0		1		1		1		1		0		0	
p_1		1	1			1	1			0	1		← wrong
p_2	0					1	1	1	0	← wrong			
p_3	0	1	1	1	1								

Parity bits p_2 and p_1 are now incorrect, identifying bit number $2^2 + 2^1 = 6$ as being the error location.

5.1 Project

Use an array `unsigned short memory[16]` to implement ECC memory. A pair of functions `store()` and `load()` write bytes into, and read bytes from, that memory.

`int store(size_t address, int i)` stores the 8-bit integer `i` into the `memory` at the given `address`, along with four additional bits that implement a Hamming code. For convenience it should return `i` as its result.

`int load(size_t address)` reads the 8-bit integer from the given `address` in the `memory` and uses the additional four parity bits to correct any single-bit errors that are detected. (If there is an error, write the corrected word back into the `memory`.)

Use the following program to test the error correction in your storage service:

```
int main(int argc, char **argv) {
    char *s = "hello, world\n";
    int i = 0;
    while (store(i, s[i])) ++i;
    for (int c, i = 0;  (c = load(i));  ++i) putchar(c);
    for (int n = 0;  n < 10;  ++n) {
        int addr = random() % 14;      // manufacture a
        int bit  = random() % 12;      // single-bit
        memory[addr] ^= (1 << bit);    // memory error
        for (int c, i = 0;  (c = load(i));  ++i) putchar(c);
    }
    // the contents of memory should still be intact
    for (int c, i = 0;  (c = load(i));  ++i) putchar(c);
    return 0;
}
```

My implementation of this prints a '!' every time a memory error is corrected. The output is:

```
$ ./12-5-hamming
hello, world
h!ello, world
hello, wo!rld
...
!hello, wor!ld
he!llo, world
hello, world
```

5.2 Hints

The physical arrangement of bits in the word is not important. Only the logical numbering of the bits matters. In the example above, the 12-bit word 011111010010 (with parity bits underlined) could be physically stored in a more convenient order such as 101001111010 or 011110101010.