Microprocessor-Based Systems Labs

Lab-session 2: Instruction Set.

Communications and data stored in memory devices under harsh environmental conditions can be susceptible to induced errors which corrupt the original information leading to erroneous data. As an example, in the Space domain, the Single Event Effects induced by heavy ions, protons and neutrons become an increasing limitation of the reliability of electronic components and can make a bit of an storage device to flip, changing a cell value passing from logic '0' to logic '1' or vice versa.

A function called EDAC (Error Detection and Correction) will add parity bits to the information in order to detect potential flips of up to 2 bits per dataword and correct automatically up to 1 bit per dataword. The aim of this session is to build an EDAC capability while applying great part of the 8086 assembly instruction set.

How does EDAC work?

EDAC uses binary **hamming codes** adding parity bits to control different bits positions with some overlapping.
Parity bit indicates whether the number of ones in a bunch of bits is even or odd. If after transmission or storage, one bit changes, then when re-computing the parity of that bunch of bits, the parity value will be different than the one indicated in the original parity bit and will mean that there was an error.
Example for general parity check:

1. Original 9-bits dataword is = "1 0 1 1 0 0 1 0 1".
2. We include 1 **parity** bit controlling all the bits in order to ensure that the number of logic '1's in 10-bits word including the parity bit shall be even.
    a. If the number of '1's in 9-bits dataword is odd, then parity bit is '1'.
    b. If the number of '1's in 9-bits dataword is even, then parity bit is '0'.

   Number of '1's in "1 0 1 1 0 0 1 0 1" → 5,
   5 is odd, then **Parity bit** equals to "**1**" and We add the parity bit in the last position, so now our dataword is 9-bits: "1 0 1 1 0 0 1 0 1 **1**"
   Now the number of '1's is even → 6
3. We transmit the data word and an error happens on the second bit ('0' turns '1'):
   "1 0 1 1 0 0 1 0 1 **1**" → "1 **1** 1 1 0 0 1 0 1 **1**"
4. We compute parity on the 10 bits:
   Number of '1's in "1 **1** 1 1 0 0 1 0 1 **1**" is → **7**
5. **7 is odd**, then we realize that there was an error in the transmission, although we do not know in which bit.

In this workshop we will implement capabilities to detect and also correct up to 1 bit error. For a 4-bits dataword, there will be 3 additional bits added for parity. Then, we have 7 bits words divided in:

- 4 DATA bits
- 3 PARITY bits

The parity bits will be added in the power of 2 positions. In our case, the parity bits will be in positions 1, 2 and 4. All other bits are data bits. Each parity bit, as shown in Table 1, will cover the following data bits:

Table 1 Hamming code parity bits H7,4

	P1	P2	D1	P4	D2	D3	D4
P1	x		X		X		X
P2		x	X			X	X
P4				x	X	X	X

Parity bit 1 (P1) covers data bits 1, 2, and 4
Parity bit 2 (P2) covers data bits 1, 3 and 4
Parity bit 4 (P4) covers data bits 2, 3 and 4

EXERCISE 1: Program dec2ASCII.asm (2 pts)

Implement a function to convert a 16-bits **decimal** number to string of bytes in ASCII format. The function shall receive the number to be converted in the register BX and shall return the converted ASCII string in DX:AX (segment:offset) ready to be printed out (DX shall store the segment and AX shall store the offset). The string shall end with symbol $ to close the string as required for the operating system SW interruption **INT 21H function 09h** (see annex).

The program shall print in screen some values converted. For example:

The number 65335 converted in ASCCI is-> 36h,35h,33h,33h,35h,'$'
When using the int 21 , ah=9, DX shall contain the offset of the string .
The characters printed after int21 execution are "65335".

EXCERCISE 2: Program Labs2a.asm (6 pts)

Implement a program in assembly language that creates a procedure function which receives a 4-bits binary chain and returns the corresponding EDAC hamming encoding as a 7-bits binary chain. The input 4-bits binary chain will be predefined in memory, where each binary digit will be allocated into a byte.

Matrix Multiplication and modulo (4 pts): To compute the parity bits in an automatic way, the 4-bits dataword shall be multiplied as a vector of size (1,4) times the following Generation Matrix of size (4,7):

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The value for this Generation Matrix is constant, and shall be also stored in memory.

Following the example, the input data word "1 0 1 1" will be multiplied as follows:

$$\begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 & 2 & 3 & 2 \end{pmatrix}$$

The output is the 7-bits in which the last three digits correspond to parity bits P1, P2 and P4. The result is obtained after computing modulo 2 of the matrix multiplication result:

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 2 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Print Results (2 pts): The program will show in the screen the computation result in the following format after re-collocating the parity bits in the proper position:

```
Input: "1 0 1 1"
Output: "0 1 1 0 0 1 1"
Computation:
      | P1  | P2  | D1  | P4  | D2  | D3  | D4

Word  | ?   | ?   |  1  | ?   |  0  |  1  |  1

P1    | 0   |     |  1  |     |  0  |     |  1

P2    |     | 1   |  1  |     |     |  1  |  1

P4    |     |     |     | 0   |  0  |  1  |  1
```

Notes:

- The procedure function of exercise 2 shall receive the 4-bits input number in registers DX:BX, 1 bit is stored per byte, and return in DX:AX the memory address of the first position where the 7 bytes output is stored (1 bit stored per byte).

- To get access to numbers in Generation Matrix the student shall use based-indexed addressing.

EXERCISE 3: Program Labs2b.asm (2 pts)

Modify the exercise 2 to ask a human user to introduce a number from 0 to 15 using the keyboard. This number will be used as the input for the EDAC encoding function. Use the operating system interruptions (see annex) to capture this number.
The ASCII code shall be translated into the corresponding decimal value. The decimal value shall be converted to binary format using the simple method of continually divide-by-2 until result is 0. Each binary digit will be stored in one byte in memory as in exercise 1. An error shall be printed in the screen if the user introduces a number bigger than 15 or lower than 0.

Example converting number 11 decimal into its binary representation "1 0 1 1":

11/2 = 5	Remainder = 1
5/2 = 2	Remainder = 1
2/2 = 1	Remainder = 0
1/2 = 0 (stops)	Remainder = 1

Binary → 1 0 1 1

In this case, there will be only two numbers to be printed which are 1 and 0. Remember that it is necessary to convert them to ASCII before printing.
Besides, each line to be printed shall have the symbol $ at the end of the characters' chain to allow execution of function number 9 of interruption INT 21h of DOS system (see annex and check alumno.asm used in Workshop 0).

DELIVERY: Date and contents:

Upload to Moodle a unique ZIP file containing only the makefile and the source files (.asm) of the exercises. Remember that only one member of the team can upload the file and the files shall contain the authors' name and the team number in the header.

Notice that the source files shall be correctly tabulated and commented. The lack of comments or poor quality ones will be graded negatively.

Deadlines to upload and submit the files: 05th of March 2018 at 23:55h

Annex: Functions of the operating system to print text and finish the program execution.

The DOS operating system facilitates most services through the 21h interruption. Before calling INT 21H instruction, the number of the requested function must be loaded in the AH register. In addition, each function may require other input parameters whose values must be stored in another series of registers. In the following lines you can find three functions of habitual use in the programs that are developed in the laboratory.

Function 2H: Sending an ASCII code to the peripheral screen

Description:	Print a unique character in the screen.

Input Parameters:	AH = 2h.
	DL = ASCII code to print.

Output Parameters:	None.
Modified Registers:	None.

Example:

```
mov ah, 2     ; Function number = 2
mov dl, 'A'   ; Character letter A to print
int 21h       ; Software interruption to the operating system
```

Function 9H: Sending an ASCII character string to the peripheral screen

Description:	Print in the screen a character string that finishes with character ASCII=$.
Input Parameters:	AH = 9h. DX = Offset in memory from the base-address of the data segment where the first character of the string is stored in memory.
Output Parameters:	None.
Modified Registers:	None.

Example:

```
.DATA
Texto DB "Hello world",13,10,'$' ; string ending with characters CR,LF y'$'
. CODE
.........
; If DS is the segment where the string to print is located:
mov dx, offset TextStr ; DX : offset to first position of the string text to print
mov ah, 9              ; Function number = 9 (print string)
int 21h                ; Software interruption to the operating system
```

Function 0AH: Reading characters type in keyboard peripheral

Description:	Captures characters introduced by keyboard, print them in screen as local echo and transfer them to the application. The function ends when the user types ASCII 13 (CR - enter). Operating system allows reading any number of typed characters up to the maximum specified in one of the input parameters. When maximum is reached, extra number of characters typed will not be stored neither showed as echo in the screen.
Input Parameters:	DX = Offset in memory from the base-address of the data segment where the typed characters will be stored. In the first byte of the pointed memory area shall be stored the maximum number of characters to capture.
Output Parameters:	In the second byte of the pointed memory area the operating system stores the real number of characters read/captured. From the third byte onwards, the operating system stores the string characters read/captures.
Modified Register:	None.

Example:

```
MOV AH,0AH              ; Function 0Ah Reading from keyboard
MOV DX,OFFSET NAME      ;Memory area allocation pointing to memory tag
NAME
MOV NAME[0],60          ;Maximum number of characters to capture = 60
INT 21H                 ; Software interruption to the operating system

;In NAME[1] the operating system stores the number of characters typed
;in NAME[2] up to NAME[61] the operating system stores the captured characters
```

Function 4CH: End of program

Description:	The function indicates that the program finished, returning a value of the execution and gives back control to the system.
Input Parameters:	AL = 00 (OK) or Error code.
Output parameters:	None.
Modified Registers:	None.

Example:

```
MOV ax, 4C00h  ; End of program
INT 21h        ; Software interruption to the operating system
```