Assembly Project for CMPE 310

Assigned: Friday, September 20

Due: Friday, September 27

Assignment Description: Hamming (8,4) Error Correcting Code

Write an assembly language program that prompts the user for a hamming (8,4) encoded string (more on this at the Wikipedia link here) as follows:

Input Data: 00100110

The encoded string is expected to be a sequence of 1s and 0s and the user is expected to input them as ASCII characters. The assembly program must then proceed to check for the validity of user input. This routine that performs validity checks is provided with the sample skeleton code. Once the validity check is performed, the program must proceed to determine if the user input has a single bit error. If a single bit error exists, then its respective bit position must be detected, and the program must print out the bit position where the error has occurred. Further, if a single bit error has been detected, your program must correct the error and the correct binary sequence must be printed out as a sequence of ASCII characters. You can make use of the ASCII to binary conversion routine provided in the skeletal code, as a reference to perform the operation in reverse. Appropriate placeholders to store information about the position and final corrected sequence is provided in the .bss section of skeletal code.

The encoding format for user input is provided below for reference:

Encoding format:

Bit Position	7	6	5	4	3	2	1	0
Bit Order	P4	D4	D3	D2	P3	D1	P2	P1

Data bits are embedded in the encoded sequence and is specified by D4, D3, D2, and D1 in the above format. Parity bits are specified by P4, P3, P2, and P1. Hence for the example above (00100110), the data bits are

D4 – 0

D3 – 1

D2 – 0

D1 – 1

And the parity bits are:

P4 – 0

P3 – 0

P2 – 1

P1 – 0

Your hamming (8,4) implementation must follow the **ODD** parity scheme.

In this scheme, a parity bit (P4, P3, P2 or P1) will have the value 1 if there are an even number of 1s in the respective data bits, or in the case of P4, the overall parity for bit positions 0-6. The parity rule is applied for parity bits P3, P2, and P1 using the following sequence of data bits.

Parity Bit	Sequence of data bits		
P1	D4	D2	D1
P2	D4	D3	D1
P3	D4	D3	D2

P4 is set to the parity value based on the bit sequence D4, D3, D2, P3, D1, P2, and P1.

Since the scheme employs ODD parity, you can utilize the **JP** and **JNP** conditional jumps that check the parity bit (lower 8 bits of the register in a preceding instruction's result).

The program must additionally check for double-bit errors. However, since double bit errors can only be detected (not corrected) when one follows the Hamming (8,4) scheme, you will need to determine the correct parity value for P4 and verify if there are double bit errors in your parity check routine. For a single bit error, you will need to print out the corrected bit sequence. In the case of a double bit error, you need only print a message that says a double bit error has occurred. The message strings to be used in your program is also made available in the provided skeleton code. You will need to employ, conditional and unconditional jumps in your routine. You may optionally employ subroutine calls using the **CALL** instruction.

If the user input has a bit error in the parity bit P4, it must be addressed separately. You might encounter three possible scenarios:

1. The user input has a bit error only in the parity bit P4.
   In this case, you are required to print an appropriate message

   Overall Parity Error Detected

   And then proceed to print the corrected sequence with the correct parity bit in P4.

2. The user input has a bit error in the parity bit P4 and at another bit position

   In this case, you will treat the error as a single bit error and print the bit position of the erroneous bit (exclusive of the parity bit P4). You are also required to print out the corrected bit sequence with both the parity bit P4 and the erroneous bit corrected.

3. The user input has a bit error in the parity bit P4 and at two other bit positions

   This case is to be treated as a double bit error and the appropriate message is to be printed

   Two or more bit errors detected!

Guidelines/Hints

- The maximum length of user input is set to 19 characters inclusive of the newline character.
- The skeleton code provided, includes a routine that verifies if user input is within limits, i.e. if the user inputs more than 9 characters (inclusive of the newline character), an invalid message is printed. Furthermore, input validity check also determines if the user input invalid characters, i.e. characters other than ‘0’ or ‘1’
- The easiest way to examine the contents of a register bit-by-bit is to use successive **SHR** instruction to shift the least significant bit into the carry flag.
- Another option would be to utilize the **SHL** instruction to shift the most significant bit into the carry flag.
- The **XOR** between determined parity bits (in the order P3,P2,P1) and the parity bits embedded in user input, provides the position of a single bit error.
- You must also make your own test cases. We will test it with ours!
  Here are a few test cases with the expected output

```
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 010100101
Invalid Data!
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 918jdy10
Invalid Data!
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 00100110
No Error Detected
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 10100110
Overall Parity Error Detected
Corrected bit sequence: 00100110
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 01000110
Two or more bit errors detected!
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 01110110
Two or more bit errors detected!
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 11000110
Two or more bit errors detected!
[deepakk1@linux5 ~]$ ./hammingecc
Input Data: 00000110
Bit error detected at position: 5
Corrected bit sequence: 00100110
```

Compiling your code

You can compile your code on GL, for debugging using the following instructions assuming you are in the directory where the assembly program is located.

```
nasm -g -f elf -F dwarf hammingecc.asm

gcc -m32 hammingecc.o -o hammingecc

gdb -tui ./hammingecc
```

You can also run your executable using the following instruction

```
./hammingecc
```

Turning in your program

Use the UNIX submit command on the GL system to turn in your project. You must submit the assembly language program as hammingecc.asm. The class name for submit is **CMPE_310**. The name of the assignment is **proj2**.

Due to any reason if you are going to submit your project late, the project name will be **late**. You are also required to turn in a single pdf report file of the code and write-up. You must include a lab cover page in the report. The write-up should include the names of the various data labels and what they are used for, description of all the labels in your code, functionality of code between two labels, loops that you have used and how they are controlled etc. Most of this can also be used as comments in the code. Properly comment the code. The breakdown of the points are as follows:

Correctness 75%
Documentation (description, etc.), code comments, modularity 10%
Demo 15%

THE LABS ARE INDIVIDUAL EFFORTS: INSTANCES OF CHEATING WILL RESULT IN YOU FAILING THE COURSE.