

# CMSC 313 Lecture 09

- **Project 2 Questions**
- **Project 3**

## Project 2: BSD Checksum

**Due: Thursday September 30, 2004**

### Objective

The objective of this programming project is to practice designing your own loops and branching code in assembly language and to gain greater familiarity with the i386 instructions set.

### Background

Checksums can be used to detect corrupted files. A file might be corrupted during transmission through a network or because the disk drive where the file is stored is damaged.

The BSD Checksum algorithm uses a 16-bit checksum. Initially, the value of the checksum is zero. For each byte of the file (in sequential order), the checksum is rotated 1 bit to the right and the byte is added to the checksum. The value of the checksum after the last byte of the file has been processed is the checksum of the file.

If the checksum of a file changes, then you know that its contents have been altered. However, it is possible for two different files to have the same checksum (since there are only 64K different values for a 16-bit checksum but many more possible files). So, having the same checksum does not guarantee that the file has not been corrupted. A well-designed checksum algorithm should be able to indicate the most common types of file corruption (e.g., transposed bits, single bits flipped).

### Assignment

Write an assembly language program that computes the BSD checksum (algorithm given above) of the `stdin` file. You should output the checksum as a 16-bit binary number to `stdout`. The intention is for you to use Unix input/output redirection:

```
./a.out <ifile >ifile.checksum
```

The value of the checksum can be examined using the `hexdump` command.:

```
hexdump ifile.checksum
hexdump -e '1/2 "%u\n"' ifile.checksum
```

The first `hexdump` command gives the result in hexadecimal. The second `hexdump` command gives the value in decimal. (It is a challenge to alias the second command in Unix.)

Some details:

- Your program must read a block of bytes from the input. You should not read from the input one byte at a time. (It would be terribly inefficient).
- You may assume that when the operating system returns with 0 bytes read that the end of the input file has been reached.
- On the other hand, you may not assume that the end of the file has been reached when the operating system gives you fewer bytes than your block size.

### Implementation Notes

- You can check your program using the `sum` command which prints out the BSD checksum of the file in decimal. (No, you may not call the Unix `sum` command from your program.)
- Look up the rotate right instruction in the Intel manual to make sure that you are using the correct rotate instruction.
- The BSD checksum algorithm involves adding an 8-bit value to a 16-bit value. Make sure you are doing this correctly.

- You will have two nested loops. The outer loop reads blocks from the input until the end of the file. The inner loop processes one character at a time. Decide ahead of time how the loops are controlled, which value is stored in which register or memory location.
- Record some sample runs of your program using the Unix script command.

### **Turning in your program**

Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) the assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is `cs313_0101`. The name of the assignment name is `proj2`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj2 checksum.asm typescript
```

## Project 3: An Error-Correcting Code

**Due: Thursday October 14, 2004**

### Objective

The objectives of this programming project are 1) for you to gain familiarity with data manipulation at the bit level and 2) for you to write more complex assembly language programs.

### Background

In Project 2, we saw that checksums can be used to detect corrupted files. However, there is not much we can do after we have detected the corruption. An error-correcting code is able to fix errors, not just detect them.

In this project, we will use a 31-bit Hamming code that can correct a 1-bit error in each 32-bit codeword. Each 32-bit codeword encodes 3 bytes of the original data. The format of the codeword is shown on the next page.

### Assignment

Write an assembly language program that encodes the input file using the codeword format described below. As in Project 2, use Unix input and output redirection:

```
./a.out <infile >infile.ham
```

Some details:

- Your program must read a block of bytes from the input. You should not read from the input one byte at a time or three bytes at a time. (That would be terribly inefficient.)
- You may assume that when the operating system returns with 0 bytes read that the end of the input file has been reached. On the other hand, you may not assume that the end of the file has been reached when the operating system gives you fewer bytes than your block size. Similarly, you may not assume that the operating system will comply with your request for a number of input bytes that is divisible by 3.
- The 32-bit codewords must be written out in little-endian format.

The C source code for two programs `decode.c` and `corrupt.c` are provided in the GL file system in the directory: `/afs/umbc.edu/users/c/h/chang/pub/cs313`. These two programs can be used to decode an encoded file and to corrupt an encoded file. You can use these programs to check if your program is working correctly. Both programs use I/O redirection.

Record some sample runs of your program using the Unix `script` command. You should show that you can encode a file using your program, then decode it and obtain a file that is identical to the original. Use the Unix `diff` command to compare the original file with the decoded file. You should also show that this works when the file is corrupted.

### Implementation Notes

- The parity flag PF is set to 1 if the result of an instruction contains an even number of 1's. Unfortunately, PF only looks at the lowest 8 bits of the result. For this project, you will need to compute 32-bit parities. Here's a simple way to compute the parity of the EAX register.

```
mov    ebx, eax
shr    eax, 16
xor    ax, bx
xor    al, ah
jp     even_label
```

Note that the EAX and EBX registers are modified in this process, so you may need to use different registers.

- A main issue in this project is handling the "extra characters" at the end of a block of input after you have processed all the 3-byte "groups". E.g., if your block size is 128, then you will have 2 characters left over after processing 42 three-byte groups ( $42 \times 3 = 126$ ). These 2 extra characters must be grouped with the first character of the next block (if there is a next block). Think about this situation *before* you begin coding.

- Another main issue is the last 32-bit word output by your program. Note that the bits m1 and m0 must be set *before* you compute the parity bits p4, p3, p2, p1 and p0.

### Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) the assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is `cs313_0101`. The name of the assignment name is `proj3`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj3 encode.asm typescript
```

### Codeword format

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
| a7 | a6 | a5 | a4 | a3 | a2 | a1 | a0 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | p4 | b0 | c7 | c6 | c5 | c4 | c3 | c2 | p3 | c1 | c0 | m1 | p2 | m0 | p1 | p0 | 0 |

bit 0 is not used and always holds a 0.

1st byte of data = a7 a6 a5 a4 a3 a2 a1 a0  
 2nd byte of data = b7 b6 b5 b4 b3 b2 b1 b0  
 3rd byte of data = c7 c6 c5 c4 c3 c2 c1 c0

p4, p3, p2, p1 and p0 are used to ensure that these bit positions have an even number of 1's:

p0: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31  
 p1: 2 3 6 7 10 11 14 15 18 19 22 23 26 27 30 31  
 p2: 4 5 6 7 12 13 14 15 20 21 22 23 28 29 30 31  
 p3: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31  
 p4: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

m1 and m0 are only used in the last word of the encoded file. They depend on the original file size (in number of bytes).

m1 m0 = 00 if the file size mod 3 is 0  
 m1 m0 = 01 if the file size mod 3 is 1  
 m1 m0 = 10 if the file size mod 3 is 2