HPPS 2024
Assignment two
Intepreting Data

Marcus Frehr Friis-Hansen lns611
Noah Wenneberg Junge qxk266
Group 7

December 5, 2024

Introduction

The code developed is functionally correct and simple but slightly annoying to test. Simply type `make all` in the terminal while in the correct directory, and then sequentially execute all executable files to test them all.

2.1 Implement `read_uint_be()` and `write_uint_be()`

The code is functionally correct. We based it on `read_uint_le` and `write_uint_le` and simply changed it so that it would work for Big Endian rather than Little Endian, meaning that the most significant byte would be stored at the lowest address. The syntax was checked using chatgbt.

Testing confirmed correct conversion of big-endian integers to little-endian and ASCII formats using the test programs `uint_be_to_uint_ascii` and `uint_be_to_uint_le`. No major issues were identified. furthermore, the implementation is efficient and concise.

Task 2.2: Implement `read_double_bin()` and `write_double_bin()`

The functions were implemented using `fread()` and `fwrite()` to handle double-precision numbers. Each double occupies 8 bytes, and little-endian byte order was assumed, as it is most commonly used on most computers today. `read_double_bin()` also uses `feof` to handle any error where `fread()` doesn't read anything. This error handling was developed using ChatGbt.

Both functions are functional and were validated using the program `double_bin_to_double_ascii` . Tests confirmed correct reading and writing of binary doubles. The implementation is pretty straightforward and no changes are necessary.

Task 2.3: Implement `write_double_ascii()`

To create this function we used `fprintf()` to write double-precision numbers in ASCII format. The \\`%g` format specifier was used for compact representation.

The function works as intended and was tested using `double_bin_to_double_ascii` and similar programs.

Task 2.4: Implement `read_double_ascii()`

In this function we used `fscanf()` for efficient parsing of ASCII-formatted doubles. It correctly handles optional signs and decimal points. The function is functional and has been validated using the program `double_ascii_to_double_bin`. ChatGbt was used to verify correct syntax.

The function is compact and efficient. No improvements are necessary.

Task 2.5: Implement `avg_doubles`

The program `avg_doubles` computes the arithmetic mean of binary double-precision numbers. It uses `read_double_bin()` to read numbers and `write_double_ascii()` to output the mean in ASCII format.

The program works correctly. Tests confirmed correct computation of the mean, including cases where no input is provided. ChatGbt was used to handle error cases, and to verify correct syntax.

Specific Questions

1. **How much larger are data files in ASCII integer format compared to binary integer format?**
   In ASCII format, each character in a given integer takes up about 1 byte of memory. A 32-bit binary integer occupies 4 bytes regardless of weather it is representing 1 or $2,147,483,648$. So for smaller number representations, ASCII integers generally occupy less bytes, while for bigger numbers ASCII integers can occupy up to 10 bytes. Eg. representing $2,147,483,648$ in ASCII format.

2. **How much larger are data files in ASCII floating-point format compared to binary floating-point format?**
   ASCII floating-point numbers are larger. A double in binary format occupies 8 bytes, whereas its ASCII representation can occupy up to 24 bytes.

3. **Is it possible to add a function `read_uint()` that can handle both binary and ASCII formats?**
   Because the formats are not self-describing it would not be possible. The function

would not be able to distinguish between binary and ASCII representations without additional metadata.

4. **Are there any floating-point values expressible in binary format but not in ASCII format?**
   No. Every binary floating-point value can be expressed in ASCII format, though precision may be limited by the ASCII representation for really long(small) floating point values.

The implementations are complete and functional. Tests confirm correctness for all tasks, and the code follows all given requirements.