# CompArch Project 2 Documentation

Frank Zhang, Allister Liu

Prof. Billoo

CU_ECE251

April 15, 2019

## Goal&Requirements

The goal of Project 2 is to build a program that will take in an input file containing at most 100 32-bit integers, one integer per line. The program will then save all the integers into the stack, and put them in increasing order. At last, the program will write all the sorted integers into a file.

We divide the input file into two cases and provides two programs to perform the sorting. If the input file is ASCII-encoded with each number sitting in one line, proj2-str handles this type of input by using certain string casting functions in the standard c library. If the input is pure binary with every 4 bytes representing an integer, proj2-bin handles this type by reading the bytes one-by-one and stick the 4 bytes together to obtain the complete integer. Proj2-str will output the file also in ASCII-encoded format, while proj2-bin will output the file in pure binary--in short, the format of the outputs are the same as inputs. The input file should be named either in-str or in-bin for the two programs, and they will output out-str or out-bin.

## Overall-Architecture

Both two programs are composed of three parts: reading and storing inputs, sorting, and loading and writing output. The second part, sorting, is the same for them, and handling different types of input is reflected in reading and writing the file. Prior to the reading, 412 bytes of space is allocated in the stack. 0-399th bytes are for storing the integer array, which is at most 100 in length, so 100*4bytes = 400bytes are preallocated. 400-403th bytes hold the pointer to the input file, 404-407th bytes hold the pointer to the output file, and 408-411th bytes hold the length of the int array. The file pointers are special data type in standard C library that abstracts the file to be processed in the program.

As mentioned, we use the standard C library to handle file I/O because it is much easier to use. Making system call to directly communicate with the OS, on the other hand, is much harder. the program will call the fopen() function in the standard C library open the input file in read mode, and it will return a file pointer that is mentioned before. It will be used by other file reading functions to obtain the integers, and thus It is stored into the stack. After initializing r7 to 0, r7 will be used as the length counter.

In proj2-str, in which the integers are represented by ASCII and dwell in lines, the program will use fgets() function from the C standard library to read the file line by line. This function takes the file pointer as input and returns a string of line, which is one integer. This string will be immediately passed into the atoi() function to convert the input to an integer, Then, the integer will be stored into the memory. atoi() does not belong to the standard library, but it appears in the Raspbian, and it conveniently takes a string and converts it into the integer, so we take its advantage. This reading process will loop with r7 increments by 1 each time. The loop end If fgets() returns a null pointer, which in Raspbian is 0. That means the file reaches the end, so the program will store the r7 value--the length of the array--into the stack, and begin sorting.

In proj2-bin, in which the integers are purely binary, fgetc() is used to obtain the data. It is like fgets(), but it returns a byte of data each time. Since an integer is 4 bytes, the program will call the fgetc() 4 times to get all 4 bytes and concatenate them to recover the integer. r8 is used as a scratch variable. After fgetc() is called for the first time, the highest byte of the integer is returned. This byte, currently staying at the 0-7th bit of r0, will be left shifted by 24 bits to return to its actual position. Then, r0 is added into r8. Then fgetc() is called for the second time to get the second highest byte. Left shift it by 16 bits and add to r8. Left shit the third highest byte by 8 and fourth highest byte by 0, and then add them into r8,

so finally the complete integer is in r8. Stores it into the stack, and the rest is the same as proj2-str. Usually when reaching the end of file, fgetc will return a EOF macro, which in Raspbian is defined as 0xFF. However, 0xFF can also appear in the integer, so we require that in-bin has 0x80000000 as the end terminator. This means -2^-32 cannot appear in the integer array, so the integer value range is restrained to -2^-32-1 ~ 2^-32 - 1, but this shouldn't affect the functionality too much.

The sorting algorithm we use--same for both programs--is bubble sort. Despite its low efficiency, it doesn't require any extra memory space, and it is easier to implement. Plus, We have already implemented one in the previous homework using LEGv8, so reusing that experience can help saving time. After storing all the inputs in the memory as integers, the program will first initiate the sorting process. It will check if there are more than 0 integer inputs -- if there are no inputs, the program will return a 1 as an error code. If the length is not 0, then r7-1 will be the highest index. After, r0 and r1 will be initiated to 0: r0 will be used as an index counter, and r1 will be used to keep track of the number of times which a swap action is performed. The address of the pointer to the stack will be saved in r8 to keep track of the location. Before every iteration, r0 and r7 will be compared to determine if the entire array is scanned. The program will load and compare the value every two adjacent integers and save them back to memory in the correct order until it reaches the end of the array -- then the value of r0 and r1 will be reset to start the next iteration. The sorting process is considered complete if there is no swap action happened in an entire iteration, where all the inputs are sorted in increasing order in the stack.

Finally, the program will use the write mode of the fopen() function to put all the reorganized integers into an output file. Under the write mode, fopen() will overwrite the file if it already exists, or create the file if it doesn't Like the input file, the output file pointer will

be stored into the stack, then r8 will be initialized to 0, it will be used as the counter to determine if the output process is complete. For each iteration, r8 will compare with the array length in the stack, and when they are equal, that means all the integers are written, and the program ends.

In proj2-str, we firstly use sprintf() function to convert the integer back to string. Originally we plan to use the itoa() function, but as a non-standard function, it doesn't appear in Raspbian, so we use sprintf() instead, which is standard. By calling the fputs() function, it will then write each integer, in the form of char, into the output file in increasing order. At last, calling the fclose() function will close the input and the output files, the stack and all the registers will be recovered, and finally, the program will return a 0 as the return code.

In proj2-str, we write the integer byte by byte using fputc(). To separate the highest bit out, the integer is left shifted by 24 bits. To separate the second highest bit out, the integer is right shifted by 8 bits, and then left shifted by 24 bits. To separate the third highest bit out, the integer is right shifted by 16 bits, and then left shifted by 24 bits. To separate the lowest bit out, the integer is right shifted by 24 bits, and then left shifted by 24 bits. They are written to the file in order from the highest to the lowest byte.

## Challenge

The part with the highest technicality is the file reading and writing, and without the standard library, this will become significantly difficult. Luckily, the Standard file I/O is provided in Raspbian for us to use. atoi() and sprintf() also bring us convenience in casting between integer and string. Without those function, we need much more lines of codes to do the conversion, which is also disastrous in assembly.

The sorting doesn’t present too much challenge. A trick that we need time to figure out is to limit the index i to length - 1. This is because we always fetch array[i] and array[i+1] from the stack, so if i reaching the length will cause the program to fetch values beyond the boundary. Other than that, sorting is done quickly thanks to the previous experience.

Another challenge is to find a way to recognize the end of the file in the binary input. 0xFF is rarely used as it doesn’t represent anything in standard ASCII. However, it can appear inside the bytes of a random integer, so we have to manually set the last integer to be 0x80000000, which is the smallest integer -2^32. This measure forbids -2^32 as a valid integer in the array, but it doesn’t affect too much since this number is rarely reached.

Before writing the code, we expected to see issues in comparing positive and negative integers, but after testing, we found out that CMP instruction can handle signed comparison, which saves us from doing complicated logic to compare signed numbers.