National Technical University of Athens
Electrical and Computer Engineering

Embedded Systems

3rd Lab

WORKSHOP IN ASSEMBLY OF THE ARM PROCESSOR

Nanos Georgios
nanosgiwrgos1997@gmail.com

Issue 1: Conversion of input from terminal

We programmed an ARM processor assembly which If the input is larger, the characters that are left over are ignored. The following is done to this string conversion:

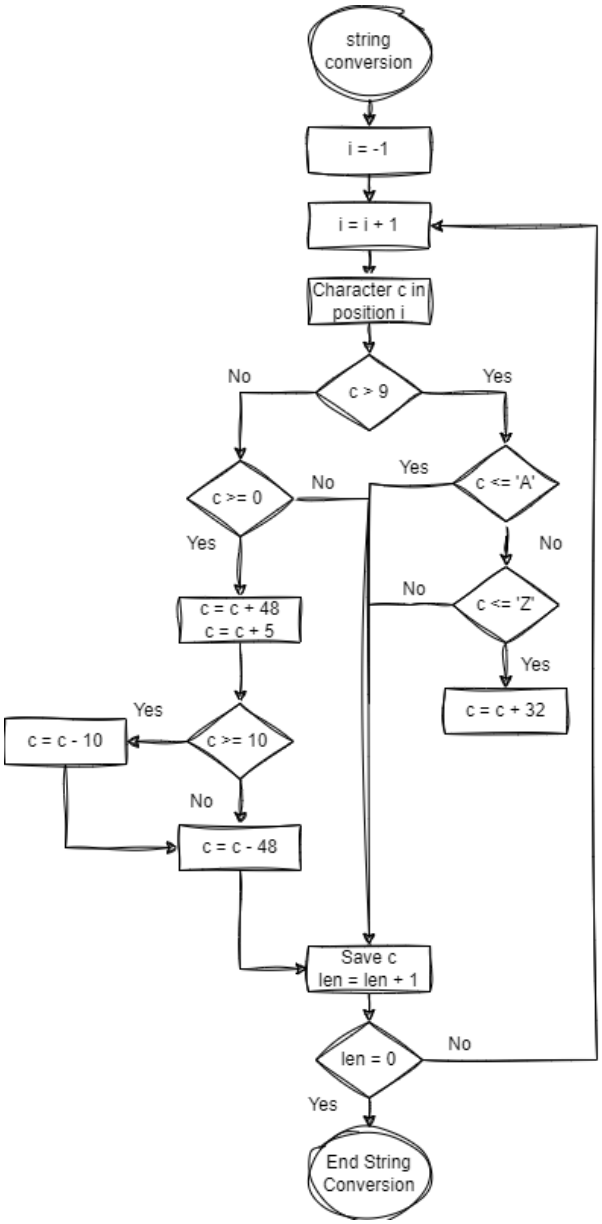
- If the character is a capital letter of the English alphabet then is converted to lower case and vice versa.
- If the character is in the range ['0', '9'], the following conversion is performed:
  1. $'0' \rightarrow' 5'$
  2. $'1' \rightarrow' 6'$
  3. $'2' \rightarrow' 7'$
  4. $'3' \rightarrow' 8'$
  5. $'4' \rightarrow' 9'$
  6. $'5' \rightarrow' 0'$
  7. $'6' \rightarrow' 1'$
  8. $'7' \rightarrow' 2'$
  9. $'8' \rightarrow' 3'$
  10. $'9' \rightarrow' 4'$
- The other characters remain unchanged.

The program is continuous and terminates when it receives as input a string of length one consisting of the character 'Q' or 'q'. The code referring to the conversion was written as a separate function. Our goal was to minimize the lines of code that required for the body of this function. The implementation of program was implemented using Linux system calls (read, write, exit). The compiling and running the programs was performed according to the instructions provided with the updated qemu environment and the use of the GNU Debugger (gdb). For proper use of gdb we run gcc with flag -g in the ask1.s file that is also in the zip. The steps in this programme are as follows:

- Writes via syscall write "Input a string of up to 32 chars long:"
- Reads via syscall read the user input.
- Checks if the length of the input is 2 (one character and the EOL). If so, checks if the character is q or Q. If it is, it calls syscall exit to terminate. If not, (in both cases) it passes to convert string function.

The convert string function takes as an argument to r3 the length of string length. The function runs through the characters present in the memory location of r1, until r3 equals 0. While it is doing this, it loads the element in memory location r1 to r0. The function has the following steps:

- Checks if it is greater than ASCII character 9. If it is, then it assumes it is a letter. If it is not, it checks if it is greater than the character 0. If it is, then it is definitely a number. If it is not, then, we don't change this character and go to the next one.
- If the character is a number, the new number we want is the original number multiplied by 5 and mod 10. Removes the character 0 in ASCII and does the addition with 5. If the number is greater than 10, then subtract 10 to get the desired result. Finally, we add the character 0 again to reconstruct our character.



- If the character is between 65 and 90, then it is a capital letter and add 32 to make it lower case.
- If the character is between 97 and 122, then it is a lowercase letter, so we subtract 32 to capitalize it.

Finally, we check if the string is exactly 32 characters long. Then, if the last character is not EOL, we add it at the end. Also, we need to clean up the remaining characters (if we have written more than 32) from stdin. We do this again with read(). If in read we get less than 32 characters, then we are good and we go back to the beginning to repeat the process. Otherwise we check the limiting case that there are exactly 32 characters that remaining. In this case the last character is EOL, so we return to the beginning of the program, otherwise we read and another.

Issue 2: Communication between guest and host machines via serial port

2 programs were created, one in C on the host machine and one in ARM assembly on the guest machine which communicate via virtual serial port. The program on the host machine receives as a string of up to 64 characters in size as input. This string is sent via a serial port to the guest machine, which in turn sends responds with which character of the string is the most frequent occurrence and how many times it appeared. The empty character (Ascii no 32) is excluded from the count. In the case where two or more characters have a maximum occurrence frequency, the program returns to the host, the character with the lowest ascii code. For communication and bus configuration, we used the library termios. On the guest machine, we called the external function tcsetattr to configure the necessary options. The selection of values for the flags was done after testing, having run c code on the guest machine. The extra commands we ran in the program c on the guest side for configuration are shown with comments in the host.c code.

```
root@debian-armel:~# gcc host.c -o host
root@debian-armel:~# ./host
Please give a string to send to guest:
bbbaaaaaaa
iflag: 0, oflag: 4, lflag: a22 cflag: 8bd
```

Having found the values for the flags, we placed them in the options of the of the final guest.s assembly code and thus we have succeeded in getting the host and guest to have a common serial port configuration. After configuring the serial port, we proceeded to implement the individual requirements:

- The host reads from stdin the string written to it by user from the terminal.
- Opens the port and sends the string to the guest, who presumably has already opened the port and is waiting in read mode to read.
- The guest reads the string, stores it in memory locations, runs it through and increments in a frequency table the occurrence values of each character it finds. This table has 256 entries, as many as the basic ascii codes and is initialized to zero.
- Then it runs through the frequency table and keeps the index of the table (which is the character's ascii code) and its value. (O table is traversed from zero up to 256, ignoring the blank character 32. This also ensures that in case of tie, the character with the smallest ascii code is returned.)
- It then writes these two values (char, freq) to the port, so that the host reads them and prints the result to stdout.

Issue 3: Linking C code to ARM processor assembly code

The program string_manipulation.c, opens an input file with 512lines, each line of which contains a randomly constructedstring, ranging in size from 8 to 64 characters. When running program, 3 output files are constructed:

1. The first contains the length of each line of the input file.
2. The second contains the strings of the input file, concatenated (concatenated) by 2.
3. The third contains the strings of the input file sorted in ascending alphabetical order.

To achieve the above objectives, the following functions are used strlen, strcpy, strcat and strcmp from the string.h library. Replaced the above functions with our own functions written in ARM assembly. Of the two ways presented to us for the assembly connection and C language, we have chosen and used strictly the second way. Specifically, first, for each function, we declared it as extern to the C code and its source code was written in another file that containing assembly code. It was necessary for the function to be declared in assembly code with the directive .global in order to be visible to the linker when linking the two files. Then we just compiled (-c flag of gcc) the string_manipulation.c file and the same for each .s of the functions. We linked the object files generated by the above steps, to produce the final executable. We did not need to change anything in the C code, other than to declare the functions as external and delete the #include . The declarations of the strlen, strcpy, strcat and strcmp functions are in the form exactly the same as the one specified when we run the man string command. Specifically, we have the following implementations:

strlen

1. We initialize the counter to zero.
2. We run through our string and as long as we don't see the null character, we increase the counter by one.
3. We return the counter.

strcpy

1. We save the base address of the dst string.
2. We run through our src string and as long as we don't see the null character, we save the character in our dst string.
3. Return the base address of our dst string

strcat

1. We save the base address of the dst string.
2. We run our dst string until we read the null, character. Thus, we save the address of the end of the dst string.
3. We traverse our src string and as long as we don't see the null character, we store the character at the end of our dst string.
4. We return the base address of the dst string.

strcmp

1. We run both strings in parallel. As we read equal characters we move on to the next one.
2. If the char of string 1 is greater than the char of string 2, then we return one.
3. If the reverse is true, we return minus one.
4. If we get to the end of both strings it means they are equal, so return zero.

In the zip file there is a makefile for proper compilation and linking of the individual files. The generated executable is named string_manipulation.out. We used the two example input files with named rand_str_input_first and rand_str_input_sec and the results are shown in the results folder.