CS1520 Practical 5

Alessandro Moura

February 15, 2018

1 Input and output in assembly

Listing 1: upper-or-lower.asm

```
.global main
main:
@ Read one line of input from the terminal
mov r0, #0      @ 0 = std input (terminal)
ldr r1, =input  @ where to store the input
mov r2, #99     @ max num of bytes to read
mov r7, #3      @ 3 = "read" system call
svc #0          @ make the system call

ldrb r3, [r1]     @ copy the first character to r3
cmp r3, #90       @ is character ASCII code greater than 'Z'?
ldrgt r1, =lower  @ if so, it must be lower-case character
ldrle r1, =upper  @ otherwise, it is upper-case

@ write the result message to terminal
mov r0, #1      @ 1 = std output (terminal)
mov r2, #12     @ number of bytes in the message
mov r7, #4      @ 4 = "write" system call
svc #0          @ make the system call

mov r7, #1      @ 1 = "exit" system call
svc #0          @ make system call

.data
upper: .asciz "upper case\n"
lower: .asciz "lower case\n"
input: .space 100  @ reserve 100 bytes of space,
                   @   initialised to 0
```

This program reads input from the keyboard, and assumes it consists of plain text. If the first letter of the input is an upper-case letter, it prints the message “upper case”; if it is a lower-case letter, it prints “lower case”.

```
@ Read one line of input from the terminal
mov r0, #0       @ 0 = std input (terminal)
ldr r1, =input @ where to store the input
mov r2, #99     @ max num of bytes to read
mov r7, #3       @ 3 = "read" system call
svc #0           @ make the system call
```

First we use the “read” system call to get input from the terminal. r0 has the code for the input device we want to read from — here we use code 0 for the standard output, which in this case means the terminal. The user’s input will be stored in the “input” string, defined at the end of the program. Only the first 99 bytes will be read — anything past that will be discarded.

```
ldrb r3, [r1]     @ copy the first character to r3
cmp r3, #90       @ is character ASCII code greater than 'Z'?
ldrgt r1, =lower @ if so, it must be lower-case character
ldrle r1, =upper @ otherwise, it is upper-case
```

We copy the first byte of the input into r3. Remember that a string is stored as an array of bytes, each byte representing a character through its ASCII code. If the ASCII code for the character is greater than the ASCII code for ’Z’, then we know that the character is not an upper case letter, and we assume that it is a lower case letter; otherwise, we assume it is a lower-case letter. Whatever the case, we store address of the string holding the appropriate message in r1. Note that we do not check for punctuation, spaces or other symbols — this is a very naive program!

```
@ write the result message to terminal
mov r0, #1       @ 1 = std output (terminal)
mov r2, #12      @ number of bytes in the message
mov r7, #4       @ 4 = "write" system call
svc #0           @ make the system call
```

This prints the message stored in r1, using the “write” system call. We have seen this in the previous practical, so we will not elaborate on this here.

```
upper: .asciz "upper case\n"
lower: .asciz "lower case\n"
input: .space 100  @ reserve 100 bytes of space,
                   @   initialisez to 0
```

Here we define the two message strings, and we also reserve 100 bytes of memory to store the input. The **.space** assembler directive also initialises the reserved memory to zero.

You will notice that when we read the input, we only read 99 bytes — one fewer byte than the memory we allocate to store it. This ensures that we always have a zero byte at the end of the string. In this case it does not matter, because we only ever look at the first character. But this is a good habit.

Go ahead and type the program. When it is executed, it will pause and wait for you to type something; after you press enter, it should print the appropriate message. Test it a few times to make sure it is working as expected.

2 Reading and writing integers

You will now write a program that reads a positive integer number from the keyboard and converts the string of digits into the corresponding value, storing it in a register. If the number is less than 100, your program should print the message "small number"; otherwise, it should print "large number".

This program is very similar to the example program in this practical — the input and output routines are almost identical. So you should start from a copy of the example program and modify it. Last practical you created a program to convert a string of digits into a number; you just need to adapt that code into your program.

The "read" system call includes the newline character at the end of the string. So for example, if you type the number 1024, this is what will be stored in the input string after the "read" system call is done:

Character	'1'	'0'	'2'	'4'	(newline)	···
ASCII	49	48	50	52	10	···

When you traverse the string to compute the number it corresponds to, you have to stop when you reach the newline character, whose ASCII code is 10. This is the main change you will have to make to your previous practical's string-to-number conversion code.

3 Reference

3.1 Assembly instructions

`mov r0, #10`	Store 10 in register r0
`mov r0, r1`	Copy the contents of register r1 to r0
`add r0, #10`	Add 10 to the value stored in r0
`add r0, r1`	Add the value stored in r1 to r0
`add r0, r1, r2`	Add r1 and r2, and store the result in r0
`sub r0, #10`	Subtract 10 from the value stored in r0
`sub r0, r1`	Subtract the value stored in r1 from r0
`sub r0, r1, r2`	Subtract r2 from r1, and store the result in r0
`mul r0, r1`	Multiply r0 by the value stored in r1 (`mul` only works on registers, and the two registers must be different)
`lsr r0, #2`	Right-shift the bits in r0 by 2 positions
`lsr r0, r1`	Right-shift the bits in r0 by the number of positions stored in r1
`lsl r0, #1`	Left-shift the bits in r0 by 1 position
`lsl r0, r1`	Left-shift the bits in r0 by the number of positions stored in r1
`svc #0`	Make a system call
`b loop`	Branch (jump) to the **loop** label
`cmp r0, r1`	Compare registers r0 and r1, storing the result in the CPSR register
`ldr r0, =var`	Load r0 with the address of **var**
`ldr r0, [r1]`	load r0 with the 4-byte content of the location in memory with address r1
`ldr r0, [r1], #4`	load r0 and increment r1 by 4 bytes
`str r0, [r1]`	store r0 at the location r1
`str r0, [r1], #4`	store r0 and at r1 and increment r1 by 4 bytes
`ldrb`	version of load instruction that loads one byte
`strb`	version of load instruction that stores one byte

3.2 Data definitions

`.data`	Start of the data section
`.word`	Followed by one or more 4-byte data declarations
`.byte`	Followed by one or more 1-byte data declarations
`.ascii`	Followed by a string
`.asciz`	Followed by a null-terminated string
`.space 100`	Reserve storage for 100 bytes, initialised to 0
`.align 2`	Aligns the next data definition to the 4-byte boundary

3.3 Conditional suffixes

Suffix	Meaning	Example
`eq`	Equal	`addeq r0, r1`
`ne`	Not Equal	`addne r0, r1`
`lt`	Less Than	`addlt r0, r1`
`le`	Less or Equal to	`addle r0, r1`
`gt`	Greater Than	`addgt r0, r1`
`ge`	Greater or Equal to	`addge r0, r1`

3.4 Unix commands

`ls`	List the contents of the current directory
`mkdir dirname`	Create a new directory called `dirname`
`cd dirname`	Move to the directory named `dirname`
`cd ..`	Move to the directory containing the current directory
`cp source_file dest_file`	Copy the file `source_file` to the file `dest_file`
`as -g -o bla.o bla.asm`	Use the assembler on the file `bla.asm` to create the object file `bla.o`
`gcc -g -o bla bla.o`	Create the executable file `bla` from the object file `bla.o`
`echo $?`	Print the last command's status to the screen

3.5 `gdb` commands

`gdb` *program*	start debugging program
`b main`	set a breakpoint at the start of the program
`b 17`	set a breakpoint at the line 17
`r`	run the program inside the debugger
`c`	continue running the program after stopping at a breakpoint
`s`	execute the next instruction and then stop
`i r`	get information on all registers
`i r r0`	get information on r0 register
`p/d (int)$r0`	print r0 as a signed integer
`i b`	get information on all breakpoints
`disable 2`	disable breakpoint 2
`enable 2`	enable breakpoint 2