CS1520 Practical 4

Alessandro Moura

February 8, 2018

1 Strings and arrays in assembly

Listing 1: upper-case.asm

```
.global main
main:
ldr r1, =msg  @ load address of string to r1

loop:
ldrb r2, [r1] @ load the current character to r2
cmp r2, #0    @ does it have ASCII code 0 ?
beq finish    @ if it does, end the loop

@ test if the character is a lower-case letter
@ if it is not, skip the "sub" instruction below
cmp r2, #97
blt store_char
cmp r2, #122
bgt store_char

sub r2, #32  @ replace r2 with the ASCII code for
             @ the corresponding upper-case letter
store_char:
strb r2, [r1], #1 @ store the character at the string,
b loop            @ go to the next character, and repeat


finish:
@ Print the string to the screen
mov r0, #1   @ the output device (1 = standard output)
ldr r1, =msg @ the address of the string to be printed
mov r2, #15  @ the number of bytes in the string
mov r7, #4   @ 4 = the code for the system call "write"
svc #0       @ make the system call
```

```
mov r7, #1    @ 1 = the code for the system call "exit"
svc #0        @ make the system call

.data
msg:   .asciz "What's up doc?\n"  @ the string
```

The program listed above converts the lower-case letters of a string to upper-case letters, leaving the other characters unchanged; it then prints the converted string to the screen. We will now discuss how the program works, focusing on the new assembly features introduced in this practical.

```
.global main
main:
ldr r1, =msg  @ load address of string to r1
```

This defines the entry point of the program, and initialises the registers we will use. We initialise r1 with the address of the location in memory where the string **msg** is stored. We will need to traverse this string, character by character, and r1 will be used to point to the current character we are working with. After the **ldr** instruction, r1 points to the first character in the string.

```
loop:
ldrb r2, [r1] @ load the current character to r2
```

At the beginning of the loop, we load r2 with the character r1 is currently pointing at. The **ldrb** instruction loads a single byte; to load an entire word (4 bytes), we would use the **ldr** version of the instruction.

```
cmp r2, #0    @ does it have ASCII code 0 ?
beq finish    @ if it does, end the loop
```

Before we do anything else, we have to check if we reached the end of the string. This is signalled by the “character” with ASCII code 0. Attention: this is **not** the character corresponding to digit ’0’ (whose ASCII code is 48); ASCII code 0 does not correspond to any printable character, and is used solely to mark the end of a string. If we find the 0 character, we break out of the loop by jumping to the **finish** label.

```
cmp r2, #97
blt store_char
cmp r2, #122
bgt store_char

sub r2, #32   @ replace r2 with the ASCII code for
              @ the corresponding upper-case letter
```

We will only change characters that are lower-case letters; all other characters are unchanged. The ascii code for ’a’ is 97, and for ’z’ is 122. So all characters whose ASCII codes fall outside of the range [97,122] will stay the same. The two comparisons test this, and if the character in r2 does is not in the range, we skip the operation `sub r2, #32`, which does the lower-case to upper-case conversion. This is indeed the conversion: the ASCII code for 'a' is 97, and for 'A' is 65; so if you subtract 32 from 'a', you get 'A'. The same is true of all the other letters.

```
store_char:
strb r2, [r1], #1 @ store the character at the string,
b loop            @ go to the next character, and repeat
```

Whether r2 was a lower-case letter or not, we always store its (possibly upper-case-converted) value back at the string, at the same address. We also advance r1 to the next character, so that the next time we go through the loop, we will operate on the next character. The store and the increment in r1 are done in the same instruction:

`strb r2, [r1], #1`

This stores a single byte in r2 at the location in memory whose address is in r1, and then increments r1 by 1. After this, we go back to the beginning of the loop, and operate on the next character.

```
finish:
@ Print the string to the screen
mov r0, #1   @ the output device (1 = standard output)
ldr r1, =msg @ the address of the string to be printed
mov r2, #15  @ the number of bytes in the string
mov r7, #4   @ 4 = the code for the system call "write"
svc #0       @ make the system call
```

This prints the string to the terminal. This works by setting up a **system call**, which is a request to the operating system (more specifically, the kernel). In Linux, we make system calls using the **svc #0** instruction. Linux has hundreds of system calls; to tell it which one we want, we store the appropriate code in the r7 register. We use code 4, which corresponds to the `write` system call. `write` allows us to send a given number of bytes to some output device. In order to use this system call, we need to give Linux three pieces of information, stored in the registers r0, r1 and r2:

- r0 stores the code for the output device we want to print to. Here we use code 1, which means "standard output" — by default, the terminal in which the program is executing.
- r1 stores the address of the string we want to print.
- r2 stores the number of bytes we want to send to the output device. The message in the example is 15 bytes (characters) long — including the newline character (`\n`).

Once all this is set up, we make the system call with the **svc #0** instruction.

```
mov r7, #1   @ 1 = the code for the system call "exit"
svc #0       @ make the system call
```

This ends the program, and is yet another example of a system call. This one does not need any extra arguments, though, so we just put the code for "exit" in r7, and make the call.

```
.data
msg:    .asciz "What's up doc?\n"  @ the string
```

This defines the string **msg**. The `asciz` directive inserts a 0 character at the end of the string, to mark where it ends. Also notice the `\n` character in the string: this is not two characters, but actually just one, the newline character. The assembler interprets the backslash followed by 'n' as a a newline character (ASCII code 10). It is there so that after the output is printed, the terminal jumps to the next line.

Go ahead and enter this program in the text editor, assemble and execute it. It should output the text

```
WHAT'S UP DOC?
```

2 Conversion from string to integer

Your task now is to write an assembly program to convert a string of digits into its corresponding integer number. For example, the string **"107"** consists of the characters '1', '0' and '7', followed by the null character; it corresponds to the integer number 107.

The algorithm for doing this is simple. First, initialise to 0 the register that will eventually hold the result; let us say we use r0 for this. Then traverse the string, and for every digit you find, multiply the current value r0 by 10, and then add the numerical value of the digit you found (so, for example, digit '3' has numerical value 3). Make sure you exit the loop when you find the end of the string. In order to figure out the numerical value of a digit, use the fact that the ASCII code for digit '0' is 48, for digit '1' is 49, and so on.

Before you start coding, use pen and paper to "run" this algorithm yourself on a small number, like 107, to see how and why it works. Once you understand it, write the program that implements it. Assume for now that all characters are digits (except the null character at the end) — that is, this is a positive number. Test it with small numbers, and use the debugger or the `echo $?` trick to see the result (remember, `echo $?` only works for 8-bit positive numbers!).

2.1 Negative numbers

Now improve on your program to include the case of negative numbers, for example "-107". You now have to test if the first character is the minus sign '-' (ASCII code 45), and record the result in some register — for example, store 1 in r11 if the first character is a minus sign, and 0 otherwise. Then do exactly the same thing as you did before to read the digits. After you finish traversing the string, reverse the sign of r0 if you did find a minus sign at the beginning of the string (you will know this by inspecting r11). Note that if there is a minus sign, you will need to skip that character before you go into the loop.

Reversing the sign of an integer number can be done in a number of ways. The easiest one is to use the **rsb** instruction, which is just like **sub**, but it subtracts its arguments in the reverse order. So `sub r0, #0` will subtract 0 from r0, and therefore will do nothing; but `rsb r0, #0` subtracts r0 from 0 and stores the result in r0, so it reverses the sign of r0.

Use the debugger to inspect r0 as a negative number; the `echo $?` command does not work for negative values. `gdb` by default prints the registers as positive integers. To see the value in r0 as a signed integer (positive or negative), use the gdb command

```
p/d (int)$r0
```

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
`.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