CS1520 Practical 6

Alessandro Moura

February 22, 2018

1 Functions in assembly

Listing 1: read-ascii.asm

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Read a string from the terminal.
@ (only the first 99 characters are read)
@
@ arguments: r1: address of string used to
@                store result
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
read_str:
push {r0-r7, lr}
mov r0, #0      @ 0 = std input (terminal)
mov r2, #100    @ max num of bytes to read
mov r7, #3      @ 3 = "read" system call
svc #0          @ make the system call
pop {r0-r7, lr}
bx lr

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ Print an integer on the screen, followed
@ by a newline. (uses C library functrion)
@
@ arguments: r1: integer to be printed
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.global printf
print_num:
push {r0-r3, lr}
ldr r0, =fmt
bl printf
pop {r0-r3, lr}
bx lr
```

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
.global main
main:
ldr r1, =input
bl read_str   @ read input from terminal
ldrb r1, [r1] @ load the first character
              @   into r1
bl print_num  @ print the ASCII code of
              @ the first character

mov r7, #1
svc #0

.data
fmt: .asciz "%d\n"
input: .space 100
```

The program listed above reads input from a terminal and prints the ASCII code of the first character in the input. The program first defines two functions (`read_str` and `print_num`), and then uses them in the `main` routine.

```
read_str:
push {r0-r7, lr}
mov r0, #0      @ 0 = std input (terminal)
mov r2, #100    @ max num of bytes to read
mov r7, #3      @ 3 = "read" system call
svc #0          @ make the system call
pop {r0-r7, lr}
bx lr
```

The `read_str` subroutine reads one line of input from the terminal and stores it in the string whose address is given in r1. So before calling this subroutine, the address of the string must be put in r1.

In order to read the input from the terminal, we simply adapted the code from last week's practical that makes the right system call. We save the values of all the registers we use (and some we do not use) with the `push` instruction, right at the start of the function. We return from the function with `bx lr`, but before we do so, we restore the original values of the registers with the `pop` instruction.

```
.global printf
print_num:
push {r0-r3, lr}
ldr r0, =fmt
bl printf
pop {r0-r3, lr}
bx lr
```

The `print_num` function prints the number stored in r1 to the terminal. We do this by using the `printf` library function — an external function (not written by us) that we can use in our code. `printf` is a core function in the C language library, and it can print many kinds of data in a multitude of formats. The line `.global printf` informs the linker that we will use an external function. Here we only use `printf` to print an integer number followed by a newline. In order to do that, `printf` needs two arguments: the address of the string `"%d\n"` in r0, and the number to be printed in r1. We will not go into the details of how `printf` works; suffice it to say that the string in the first argument instructs `printf` to print the integer number in the second argument in the decimal format, followed by a newline.

`printf` changes the values of registers r0–r3, so we restore them before returning from the function.

```
ldr r1, =input
bl read_str    @ read input from terminal
ldrb r1, [r1] @ load the first character
               @   into r1
bl print_num   @ print the ASCII code of
               @ the first character
```

Our main routine first calls `read_str`, and then loads the first character of the input into r1. The instruction `ldrb r1, [r1]` may look strange at first sight, but it is simply storing the first byte of the input string (whose address is in r1) into r1. So after this instruction, register r1 no longer contains the address of the string — it stores the ASCII code of the first character instead. The `print_num` function is then called to print the number.

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 ASCII code of the first character. For example, if you type “ascii”, it will print the number 97, the ASCII code of ’a’.

2 Reading and writing integers

You will change the program above by first adding a function to read a number from the terminal; the function should then convert the string of digits it reads into the corresponding integer number, which you can assume to be positive. It should store that number in r0. You should adapt the code you developed in practical 4 for just this purpose, and make it into a function. Then change the main routine so that it reads a number from the terminal, and then prints that number multiplied by 3. You will use the functions in the example program above, so we suggest you use it as a starting point.

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
`bl func`	call function `func`
`bx lr`	return from function
`push`	push specified registers into stack
`pop`	pop specified registers from stack

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