CS1520 Practical 3

Alessandro Moura

February 1, 2018

1 Loops in assembly

Listing 1: loop-sum.asm

```
.global main
main:
@ Compute the sum 1 + 2 + 3 + ...+ N

mov r0, #0  @ r0 will accumulate the sum
mov r1, #10 @ r1 is N
mov r2, #1  @ r2 is the loop index

loop:
add r0, r2  @ r0 = r0+r2
add r2, #1  @ r2 = r2+1
cmp r2, r1
ble loop    @ repeat the loop, if r2 <= N

mov r7, #1
svc #0
.end
```

This program computes the sum

$$S = 1 + 2 + 3 + \cdots + N.$$

N is stored in r1, and the result is placed in r0.

We now discuss how the program works.

```
.global main
main:
@ Compute the sum 1 + 2 + 3 + ...+ N

mov r0, #0  @ r0 will accumulate the sum
mov r1, #10 @ r1 is N
mov r2, #1  @ r2 is the loop index
```

This defines the entry point of the program, and initialises the registers we will use. r2 is the loop index, which starts with value 1, and will get incremented to 2, 3, 4, and so on in successive iterations of the loop.

```
loop:
```

This defines the label "loop", which marks a location in the program that we can later jump to. The label can be any name you choose (with some restrictions).

```
add r0, r2  @ r0 = r0+r2
add r2, #1  @ r2 = r2+1
```

Here we add the loop index to the current value of the sum, and then we increment the index. In this way, as we go through the loop multiple times, the values 1, 2, 3, etc are added to r0.

```
cmp r2, r1
```

Now we have to decide whether to go through the loop again. The **cmp** instruction compares the values in registers r2 and r1, and stores the result of this comparison in the special CPSR register (Current Program Status Register). Conditional instructions after this will either execute or do nothing, depending on the result of this comparison.

```
ble loop    @ repeat the loop, if r2 <= N
```

This is a conditional version of the **branch** instruction. The unconditional branch instruction is **b**. The

```
b loop
```

would always send the program to the location in the code where the label **loop** is defined. But this would be an infinite loop: the program would keep executing the instructions between the loop label and the branch instruction, and it would never terminate. Instead, we use the conditional version **ble** of the branch instruction, which only branches if the two registers compared

with **cmp** satisfy the condition "less than or equal to" — hence the "le" suffix attached to the "b" mnemonic. So branching happens if the number in r2 is less than or equal to r1. Since r2 is continuously increasing, while r1 is fixed, this condition will eventually be violated, and then the loop will terminate.

In general, almost all instructions in ARM assembly can be made conditional by adding appropriate suffixes to them. These are all the suffixes we will need in this course:

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`

For example, the instruction `addgt r0, r1` will add r1 to r0, but only if the first operand of the last **cmp** instruction was greater than the second.

Branching works by manipulating the value of the **pc** register (also known as r15). This register stores the address in memory of the next instruction to be executed by the CPU. By default, after executing an instruction, the pc register is automatically incremented to the address of the next instruction in the program. The branch instruction overrides this, and stores in pc the address of the first instruction following the corresponding label.

```
mov r7, #1
svc #0
```

This ends the program, using the (lower 8 bits of) register r0 as the result; that is the number you see when you use the `echo $?` command.

Go ahead and type this program, save it in a file named `loop-sum.asm`, then assemble and link it in the usual way, by using the "build" command you created in the first practical. Execute it with

`./loop-sum`

and then see the result with

`echo $?`

The result should be 55.

2 Debugging

A **debugger** is a program that allows you to step through the execution of a program and inspect its registers, variables and other information while the program is executing. As the word suggests, debuggers are extremely helpful in helping find and fix bugs; but they are also an invaluable as a means of seeing what your program is doing, even when there is nothing wrong with it.

We will now use the **gdb** debugger, to examine the execution of the `loop-sum` program. `gdb` is a command-line tool, and you control it using simple commands typed in the terminal. To start it, just type `gdb` followed by the name of the executable:

```
gdb loop-sum
```

You will see a "welcome message" on the screen, and then you will be presented with a `(gdb)` prompt where you issue commands to `gdb`. The `loop-sum` program is not running yet. When we run the program through `gdb`, we want it to pause right at the start, so that we can inspect the registers and see how they are affected as the program progresses. So our first action is to set a **breakpoint** at the start of the program. Breakpoints are locations in the program where we want execution to stop. To create a breakpoint at the start of the program, that is, just after the `main` symbol, use the command

```
breakpoint main
```

This can be abbreviated to:

```
b main
```

After pressing `enter`, you should see something like this:

```
(gdb) b main
Breakpoint 1 at 0x103ec: file loop-sum.asm, line 6.
```

Note that you do *not* type the "(gdb)" prompt!

We can now run the program inside gdb, using the command `run`, or its abbreviation `r`

```
(gdb) r
Starting program: /home/pi/cs1520/loop-sum
Breakpoint 1, main () at loop-sum.asm:6
6       mov r1, #10 @ r1 is N
```

The program runs, and then pauses after executing the first instruction, the instruction `mov r0, #0` in line 5. `gdb` then shows the line of code where

execution has paused — line 6 in our example. Note that the instruction shown, setting r1 to 10, has *not* executed yet: this is the instruction that will be executed next.

We can now inspect the contents of the registers, using the command `info registers`, or its abbreviation, `i r`

```
(gdb) i r
r0             0x0        0
r1             0xbefff144         3204444484
r2             0xbefff14c         3204444492
r3             0x103e8    66536
r4             0x0        0
r5             0x0        0
r6             0x102c0    66240
r7             0x0        0
r8             0x0        0
r9             0x0        0
r10            0xb6ffc000         3070214144
r11            0x0        0
r12            0xb6fb1000         3069906944
sp             0xbeffeff0         0xbeffeff0
lr             0xb6e8c294         -1226259820
pc             0x103ec    0x103ec <main+4>
cpsr           0x60000010         1610612752
```

The `i r` command gives us a list of all registers and their current values, in hexadecimal (first column) and decimal (second column) format. As we can see, r0 was set to zero by the first instruction in the program, but r1 and r2 are still uninitialised, and contain trash values (they may be different for you).

We can now step through the program line by line, and see what happens to the registers. The command `step`, or its abbreviation `s`, executes the next instruction in the program, and then pauses again:

```
(gdb) s
7       mov r2, #1  @ r2 is the loop index
```

This executes the `mov r1, #10` instruction, and advances to the line in the program where the next instruction is located. We can check that the r1 register has indeed been set to 10 using the `i r` command again:

```
(gdb) i r
r0             0x0        0
r1             0xa        10
r2             0xbefff14c         3204444492
....
```

From now on we will omit the registers that do not interest us. We can also print the information for just one register, using the command `i r r2`, for example.

Stepping one more time should set r2 to 1:

```
(gdb) s
loop () at loop-sum.asm:10
10      add r0, r2  @ r0 = r0+r2

(gdb) i r r2
r2             0x1        1
```

Notice that when you step, `gdb` goes to the next instruction, not the next line in your code: blank lines, lines consisting entirely of comments, and label definitions are skipped.

We don't always want to step through a program instruction by instruction. For example, we want to inspect the registers after each iteration of the loop in our `loop-sum` program, but we do not want to be stepping through each individual instruction in the loop. So we define another breakpoint at the end of the loop, at instruction `ble loop`, in line 13 (if that instruction is in a different line in your code, use that one instead).

```
(gdb) b 13
Breakpoint 2 at 0x10400: file loop-sum.asm, line 13.
```

Now we can continue the execution of the program with the `continue` command, or its abbreviation `c`, and examine the values of the registers after one loop iteration.

```
(gdb) c
Continuing.

Breakpoint 2, loop () at loop-sum.asm:13
13      ble loop    @ repeat the loop, if r2 <= N
```

```
(gdb) i r
r0             0x1        1
r1             0xa        10
r2             0x2        2
....
```

So after the first iteration, r0 is 1, and the index r2 has been incremented to 2. If we continue execution, we will go through another iteration:

```
(gdb) c
Continuing.

Breakpoint 2, loop () at loop-sum.asm:13
13      ble loop    @ repeat the loop, if r2 <= N

(gdb) i r
r0             0x3        3
r1             0xa        10
r2             0x3        3
....
```

So now r0 has accumulated the sum $1+2=3$, and the index increased to 3. So we can see that the loop is working as intended. Repeat this a couple more times to convince yourself.

Now that we are satisfied that the program is working as it should, we just want to jump to a location after the loop, when the computation of the sum is completed. To do this, we create yet another breakpoint:

```
(gdb) b 15
Breakpoint 3 at 0x10404: file loop-sum.asm, line 15.
```

But we do not want to have to go through the loop many more times before we hit the breakpoint at line 15: at this point, we want to skip the loop breakpoint. To do this, we use the `disable` command, which "turns off" a breakpoint. But what breakpoint do we want to disable? The `info breakpoints`, or `i b` command, lists all the breakpoints you have created.

```
(gdb) i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000103ec loop-sum.asm:6
```

```
        breakpoint already hit 1 time
2       breakpoint     keep y   0x00010400 loop-sum.asm:13
        breakpoint already hit 3 times
3       breakpoint     keep y   0x00010404 loop-sum.asm:15
```

We see that breakpoint number 2 is the one we want to disable.

```
(gdb) disable 2
```

We can now let the program execute, and it will keep going until it reaches line 15, at which point we can look ar the final result of the computation.

```
(gdb) c
Continuing.

Breakpoint 3, loop () at loop-sum.asm:15
15      mov r7, #1

(gdb) i r
r0             0x37     55
r1             0xa      10
r2             0xb      11
....
```

As we see, r0 stores the correct value 55. If we continue again, the program will reach the end and terminate.

```
(gdb) c
Continuing.
[Inferior 1 (process 826) exited with code 067]
```

We can exit `gdb` with the `quit` command, or restart the program from the beginning with `run` (or `r`). The program can be restarted at any point, even before it has terminated.

3 Dividing integer numbers

Your task now is to create an assembly program that divides two integer numbers a and b, and computes the quotient (a/b) and the remainder of the division. For example, let us choose $a = 13$ and $b = 3$. Then their quotient is $q = 4$, and the remainder is $r = 1$, because

$$13 = 4 \times 3 + 1.$$

In general, q and r are defined by

$$a = q \cdot b + r.$$

You will implement a very simple algorithm to divide two positive integer numbers. To describe how the algorithm works, consider the example $a = 13$ and $b = 3$. Now answer the question: how many b's do you need to add so that the sum becomes greater than a? Let m be the minimum number of b's you need to add to get a number greater than a. The quotient q is given by $m - 1$. In our example, $m = 5$, because

$$3 + 3 + 3 + 3 = 12 < 13,$$

but

$$3 + 3 + 3 + 3 + 3 = 15 > 13.$$

Test other numbers to convince yourself that this works for any positive numbers a and b.

The program you write should have a loop which keeps adding b to a register, and testing if the accumulated value has become greater than a. If it is, break out of the loop. Your code should also keep track of haw many times you have been through the loop — that is the value of m, form which you get the quotient q. Once you have q, the remainder r can be computed directly by

$$r = a - q \times b.$$

In your program, store the quotient in r0 and the remainder in r1; a and b should be placed in r2 and r3. Use the debugger to see the values of r0 and r1 after the computation is done, and check that they are correct.

Warning: this algorithm for division, while correct, is very inefficient. If you are curious, look up the binary version of the long division algorithm, which is much faster than this one. If you want a little bit of an extra challenge, try implementing it in assembly (you will need some instructions we have not seen in the course yet).

3.1 Checking for errors

What happens if we try to divide by zero? At the moment, your program will go into an infinite loop, and will never terminate (can you see why?). It should abort instead. Add code to your program that tests if $b = 0$, and terminate if that is the case, setting r0 and r1 to zero.

4 Reference

4.1 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

4.2 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