CS1520 Practical 1

Alessandro Moura

January 24, 2018

1 Simple arithmetic in assembly

Listing 1: hello.asm

```
.global main @ mark this as the entry
main:        @  point of the program
@ calculate the sum S = m + (m+1) + ... + n
@ using the formula S = (n+m)*(n-m+1)/2

mov r4, #1    @ m=1
mov r5, #10   @ m=10

mov r0, r4    @ r0 = m
add r0, r5    @ r0 = m+n
mov r1, r5    @ r1 = n
sub r1, r4    @ r1 = n-m
add r1, #1    @ r1 = n-m+1
mul r0, r1    @ r0 = (m+n)(n-m+1)
lsr r0, #1    @ r0 = (m+n)(n-m+1)/2

mov r7, #1    @ make the system call
svc #0        @  to exit the program
.end
```

The program listed above computes the sum S of all the (positive) integers from m to n, using the formula

$$S = \frac{(m+n)(n-m+1)}{2}.$$

In the example listed, the values of m and n are stored in the registers r4 and r5, and they are set to 1 and 10, respectively.

Let us walk through this program and see how it works.

```
.global main @ mark this as the entry
main:        @  point of the program
```

The operating system needs to know where to start executing your program: the *entry point* of the program. These two lines tell the OS that the program starts from the line immediately following `main:`.

The "@" symbol marks the start of a comment: the assembler ignores the text from the "@" symbol to the end of the line. You should comment your programs: assembly programs are hard to understand if you do not comment them extensively.

```
mov r4, #1    @ m=1
mov r5, #10   @ m=10
```

Number 1 (representing m) is stored in register r4, and number 10 (representing n) is stored in r5. This version of the instruction `mov` only works with a rather limited range of numbers. To simplify our life, let us assume it only works in the range 0–255 (although it is actually more complicated than that). We will later see how to circumvent this restriction.

```
mov r0, r4    @ r0 = m
add r0, r5    @ r0 = m+n
```

The first step is to compute the value $(m + n)$, by first copying m (in r4) into r0, and then adding n to r0. Notice that `mov` here uses two registers as operands; this version of `mov` always works, without restrictions. The `add` instruction adds the number stored in r5 to r0; notice that after the `add` instruction is executed, r0 is changed, but r5 remains the same.

```
mov r1, r5    @ r1 = n
sub r1, r4    @ r1 = n-m
add r1, #1    @ r1 = n-m+1
```

This computes the term $(n - m + 1)$, storing the result in register r1. `sub` works similarly to `add`; it subtracts the second operand from the first.

```
mul r0, r1    @ r0 = (m+n)(n-m+1)
```

Now we multiply the two terms we computed previously. `mul` is similar to `add`, but it computes the product of the two operands. However, unlike `add` and sub, mul can only be used with registers, not with raw numbers. In addition, the two registers must be different.

```
lsr r0, #1    @ r0 = (m+n)(n-m+1)/2
```

The final operation we need to do now is to divide by 2 the product of the two terms, which we computed with the previous instruction. The version of the raspberry pi we are using does not have an integer division instruction; but division by 2 is equivalent to shifting all bits of the binary representation of the number to the right by one position. So we use the lsr instruction (*logical shift right*) to do that.

```
mov r7, #1    @ make the system call
svc #0        @  to exit the program
```

Now that we are done, we have to tell the operating system that it can end this program, and reclaim resources used while it was running (such as memory). To do that, we make a *system call*, which is a request for some service from the OS's kernel. Each system call is identified by a numerical code. We want the system call that terminates the program; the code for that system call is 1. So we store the number 1 in register r7, and then we call the instruction `svc #0`, which passes control of our program to the OS kernel. The kernel looks at r7, sees the code for the "exit" system call, and terminates the program.

1.1 Where is the output?

Although our program computes the desired result, how can we see what that result was after the program runs? We haven't learned how to do IO in assembly yet, so we cannot print the result to the screen yet. But when the exit system call terminates a program, the number stored in register r0 is used as the "result" of the program. After we run any command in the terminal, we can see what the result of the command was, by typing

```
echo $?
```

This prints the numerical value corresponding to the result of the last command executed. Here we use the program's result to communicate the result of our computation.

So now type the program in your text editor, saving it as `sum.asm`. Then assemble and link it by using the `build` command you created in the last practical:

```
./build sum
```

Alternatively, you can use the `as` and `gcc` commands directly. Either way, this creates the executable file `sum`, that you run using

```
./sum
```

When you run the program, no result is printed. Now type the command

```
echo $?
```

If everything went well, you should see the number 55 printed on the screen.

Change the values of m and n (that is, the values stored in r4 and r5 at the start of the program), assemble and link the program again, then repeat the above steps, and see how the result changes. Choose small values for m and n, so that you can calculate the correct result by hand and compare with the output of your program to test it.

Warning: the Unix system assumes that the status returned by a program as an 8-bit number, so the command `echo $?` only works correctly if the result is in the range 0–255. If the result is outside of this range, the command will report an incorrect result.

2 Try it yourself

Now try to create a program that works like `sum.asm`, but computes the expression A:

$$A = x^2 + 2axy + y^2.$$

Store the values of x, y and a in the registers r10, r11 and r12, respectively. Use $x = 2$, $y = 3$ and $a = 4$ (with these values, the correct result is $A = 61$). As before, the final result should be in register r0 before the program terminates.

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

3.2 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