Introduction to Assembly Programming

Luís Nogueira

lmn@isep.ipp.pt

2021/2022

Notes:

- Each exercise should be solved in a modular fashion. It should be organised in two or more modules and compiled using the rules described in a Makefile
- Unless clearly stated otherwise, the needed data structures for each exercise must be declared as global variables in the main C module
- The code should be commented and indented

1. Create a Makefile for compiling the following files: `asm.h`, `asm.s`, and `main.c`. The compilation process must keep debug information. Then, run the program in debug mode (with GDB).

```
/***********************asm.h************************/
#ifndef ASM_H
#define ASM_H
void sum(void);
#endif


/************************asm.s************************/
.section .data
 .global op1
 .global op2
 .global res

.section .text
 .global sum    # void sum(void)
 sum:
   movl op1(%rip), %ecx   #place op1 in ecx
   movl op2(%rip), %eax   #place op2 in eax
   addl %ecx, %eax        #add ecx to eax. Result is in eax
   movl %eax, res(%rip)   # copy the result to res
   ret


/************************main.c************************/
#include <stdio.h>
#include "asm.h"
```

```
int op1=0, op2=0,res=0;

int main(void) {
   printf("Valor op1:");
   scanf("%d",&op1);
   printf("Valor op2:");
   scanf("%d",&op2);
   sum();
   printf("sum = %d:0x%x\n", res,res);
   return 0;
}
```

2. Change the function `void sum` in the previous exercise to `int sum()`. The returned value should be stored in a local variable of the main function. In other words, the `res` variable is no longer needed.

3. Add an assembly function `int sum_v2()` to the previous exercise to perform the following operation: `(CONST - op1) - (CONST - op2)`. *CONST* should be a constant with value 15 declared in the Assembly module.

4. Add an assembly function `long sum_v3()` to the previous exercise to perform the following operation: `op4 + op3 - op2 + op1`, adding the needed variables to your program. The variables `op3` and `op4` should be declared in Assembly with type long, but should also be accessible from C.

   não passa unit test

   To print a signed 64 bits variable, use the "%ld"specifier. For unsigned longs, use the "%lu"specifier. More information can be fount at `http://man7.org/linux/man-pages/man3/printf.3.html`

5. Create a new program to manipulate short values. Implement, in Assembly, the function `short swapBytes()`. This function swaps the bytes of a 16 bit variable `short s`, that is, the most significant byte of `s` becomes the least significant byte and vice-versa. The function should return the new short value

   trocar registos %ah por %bl e %bh por %al

   To print a signed 16 bits variable, use the "%hd"specifier. For unsigned shorts, use the "%hu"specifier. More information can be fount at `http://man7.org/linux/man-pages/man3/printf.3.html`

6. Add a function `short concatBytes()` to the previous exercise. This function, that must be implemented in Assembly, concatenates two bytes `char byte1` and `char byte2` into a single short. The constructed short should be returned and printed in C.

   juntar bytes- esta funçao ira juntar 2 bytes (1 e 2) e depois devolver o valor

7. Add a function `short crossSumBytes()` to the previous exercise. This function, that must be implemented in Assembly, sums two short values, `short s1` and `short s2`, in a crossed fashion. The function should sum the most significant byte `s1` with the least significant byte of `s2` and vice-versa. The computed result should be returned in a single short.

   addcb

   addc somar com os vai (numeros maiores de 10)

   somar dois shorts somar-o byte mais significaivo da primeira variavel com o byte mais significativo da segund

8. Repeat the previous exercise but, this time, the needed variables must be declared in Assembly.

   criar as variaveis em assembly

9. Implement an Assembly function `long sum_and_subtract()` to perform the following operation: `C + A - D + B`. `A` is a 8-bit variable, `B` is a 16-bit variable, while `C` and `D` are both 32-bit variables. The function should return a 64-bit value that must be printed in C.

   2 testes falham

10. Implement an Assembly function `long long sum2ints()` to perform the following operation: `op1 + op2` (both 32-bit values declared in C). The function should return a 64-bit value that must be printed in C.

11. Create an Assembly function `char test_flags()` that sums two 32-bit variables, `int op1` and `int op2`, and check if such operation activates the carry and overflow flags. The function should return 1 if any of those flags is activated, or 0 otherwise. Test the function with several values and show the obtained results accordingly.

12. Implement an Assembly function `char isMultiple()` to check if the number `A` is multiple of `B`. The function should return 1 if that is the case, or 0 otherwise. Both `A` and `B` should be integer values declared in C.

13. Implement an Assembly function `int getArea()` to compute the area of a triangle. The base and height of the triangle are stored in two integer variables declared in C, `int base` and `int height`, respectively.

14. Repeat the previous exercise, but this time the base and height of the triangle are stored in two integer variables, `base` and `height`, declared in Assembly, but also accessible from C. The computed result should be printed in C.

15. Create an assembly function `int compute()` to perform the following operation: `((A * B) + C) / D` (all 32-bit variables).

16. Implement a function `int steps()` that, given a number (a 64-bit integer value stored in variable `long num`), computes its result according to the following set of successive steps:
    a) Multiplies by 3
    b) Adds 6
    c) Divides by 3
    d) Adds 12
    e) Subtracts `num`
    f) Subtracts 1

    The obtained result should be printed in C.

17. Implement a basic calculator with support for the following integer arithmetic operations: sum, subtraction, multiplication, division, modulus, powers of 2 and 3. Each of these operations should be implemented in a separate function in Assembly. The integer operands should be declared in C, while the computed result should be a 32-bit value declared in Assembly.

18. Create an Assembly function to perform the following operation:

$$\sum_{i=1}^{n} i^2 * A^2 / B$$

    `A` and `B` should be constants defined in Assembly, while `i` should be declared in C.

19. Consider that the air conditioning system "HotCold"needs:
    - three minutes to decrease one Celsius degree;
    - two minutes to increase one Celsius degree.

    Create an Assembly function `int needed_time()` that, given the `current` and the `desired` temperatures, computes the time (in seconds) required to change to the desired temperature. `current` and `desired` should be 16-bit variables. The function should return the computed result as a 32-bit value.

20. Create an Assembly function `char check_num()` that, given a 32-bit variable (`num`), returns:
    - 1, if `num` is even and negative;

- 2, if `num` is odd and negative.
- 3, if `num` is even and positive;
- 4, if `num` is odd and positive;

21. Your company will raise the salary of its employees according to the following table:

Code	Position	Raise in Salary
10	Manager	300 euros
11	Engineer	250 euros
12	Technician	150 euros
All other codes	All other positions	100 euros

Create an Assembly function `int new_salary()` that, given two 32-bits variables (`code` and `currentSalary`) declared in C, returns the new salary.

22. Code all these functions in Assembly and C. Compare the obtained results.

```
int f(){
    if (i == j)
        h = i - j + 1;
    else
        h = i + j -1;
    return h;
}
```

```
int f2(){
if (i > j)
        i = i - 1;
    else
        j = j + 1;
    h = j * i;
    return h;
}
```

```
int f3(){
    if (i >= j) {
        h = i * j;
        g = i + 1;
    }
    else {
        h = i + j;
        g = i + j + 2;
    }
    r = g / h;
    return r;
}
```

```
int f4(){
    if (i + j < 10)
        h = 4 * i * i;
    else
        h = j * j / 3;
    return h;
}
```