

Functions with Parameters and Local Variables in Assembly

Bitwise Operations in C and Assembly

Luís Nogueira

lmn@isep.ipp.pt

2022/2023

Notes:

- Each exercise should be solved in a modular fashion, organized in a set of files and with an associated Makefile.
- The source code must be commented and have a consistent indentation
- When there are subpoints in the exercises they should be placed in different folders called `ex11a`, `ex11b` ...
- Unless clearly stated otherwise, the needed data structures for each exercise must be passed as parameters to the implemented functions and no global variables should be used, neither in C or Assembly

Implement the following functions in Assembly and test their behavior by using them in a C program:

1. Implement the function `long cube(int x)` that returns the cube of the integer number 'x' passed as a parameter.
2. Implement the function `long sum_n2(int n)` that returns the sum of the squares of integers 1 to *n* ('n' is passed as a parameter).
3. Implement the function `int greatest(int a, int b, int c, int d)` that returns the greatest of four integer numbers passed as parameters.
4. Implement the function `int sum_smaller(int num1, int num2, int *smaller)` that returns the sum of the two numbers, `num1` and `num2`, and place the smaller of the two in the memory area pointed to by `smaller`.
5. Implement the function `int inc_and_square(int *v1, int v2)` that increases by one the value pointed to by `v1` and returns the square of `v2`.
6. Implement the function `int test_equal(char *a, char *b)` that detects if two strings are equal. If the strings are equal, the function should return 1, or 0 otherwise.
7. Implement the function `int count_odd(char *vec, int n)` that given the start address of a vector of chars with 'n' elements, returns the number of odd numbers in the vector.

8. Implement in Assembly the function `int calc(int a, int * b, int c)` with the behaviour presented in Listing 1. Note: Use in Assembly the required local variables.

Listing 1: calc.c

```
int calc(int a, int *b, int c)
{
    int z=a*2-(*b);
    return (c-4)*z;
}
```

9. The function `print_result(...)` in Listing 2 prints the result of an arithmetic operation. Place it on your C main module.

Listing 2: print_result.c

```
void print_result(char op, int o1, int o2, int res)
{
    printf("%d %c %d = %d\n", o1, op, o2, res);
}
```

Implement in Assembly the function `int calculate(int a, int b)` with the behaviour presented in Listing 3. Note: Use in Assembly the required local variables.

Listing 3: calculate.c

```
int calculate(int a, int b)
{
    int diff,product;
    diff=a-b;
    product=a*b;
    print_result('*', a, b, product);
    print_result('-', a, b, diff);
    return (a-b)-(a*b);
}
```

10. Implement in Assembly the functions `int incr(short *p1, char val)` and `int call_incr()` with the behaviour presented in Listing 4. Note: Use in Assembly the required local variables.

Listing 4: incr.c

```
int incr(short *p1, char val)
{
    int x = (int)*p1;
    int y = x + val;
    *p1 = (short)y;
    return x;
}

int call_incr(short w)
{
    short x1 = w+3;
    int x2 = incr(&x1,0xC3);
    return (x1 + x2);
}
```

11. Implement in Assembly the functions `void proc(int x1, int *p1, int x2, int *p2, short x3, short *p3, char x4, char *p4)` and `int call_proc()` with the behaviour presented in Listing 5. Note: Use in Assembly the required local variables.

Listing 5: `proc.c`

```
void proc(int x1, int *p1, int x2, int *p2, short x3, short *p3, char x4, char *p4)
{
    *p1 = x1 + x2;
    *p2 = x2 - x1;
    *p3 = x3 + x2;
    *p4 = x4 * 3;
}

int call_proc(int a, int b, short c, char d)
{
    int x1 = a, x2 = b;
    short x3 = c;
    char x4 = d;
    proc(x1,&x1,x2,&x2,x3,&x3,x4,&x4);
    return (x1 + x2) * (x3 - x4);
}
```

12. Implement the function `int count_bits_one(int x)` that counts the number of active bits (with the value 1) in a number x .
- In C.
 - In Assembly.
 - Use the Assembly function developed in b) in another Assembly function `int vec_count_bits_one(short *ptr, int num)` that counts the total number of active bits in a vector of shorts.
13. Implement in C the functions:
- `int rotate_left (int num, int nbits)` - this function rotates the value `num`, `nbits` to the left.
 - `rotate_right (int num, int nbits)` - this function rotates the value `num`, `nbits` to the right.
14. Implement the function `int reset_bit(int *ptr, int pos)` that, given a pointer to an integer, places '0' on the bit given by `pos` (a value within $0 \dots 31$). The function should return 1 if the bit was altered or 0 if the bit was already zero.
- In C.
 - In Assembly.
 - Use the Assembly function developed in b) in another Assembly function `void reset_2bits(int *ptr, int pos)` that resets two bits. The function should reset the bits n and $31 - n$.
15. Implement the function `int activate_bits(int a, int left, int right)` that should 'activate' all the bits to the left of `left` and to the right of `right` on the number `a` (excluding the bits `left` and `right`).
- In C.
 - In Assembly.
 - Use the Assembly function developed in b) in another Assembly function `int activate_invert_bits(int a, int left, int right)` that also inverts the result of the previous function.

16. Implement the function `int join_bits(int a, int b, int pos)` that has as a purpose to return a number composed by the bits $b_{31}b_{30}...b_{pos+1}a_{pos}...a_1a_0$
- In C.
 - In Assembly.
 - Use the Assembly function developed in b) in another Assembly function `int mixed_sum(int a, int b, int pos)`. The function should return the sum of `int join_bits(int a, int b, int pos)` with `int join_bits(int b, int a, int pos)`
17. Considering that the 32 bits on an unsigned integer represent a date like this:
- Bits 0 to 7 are the month
 - Bits 8 to 23 are the year
 - Bits 24 to 31 are the day
- Implement the function `unsigned int greater_date(unsigned int date1, unsigned int date2)` that returns the greater of the two dates passed as parameters.
- In C.
 - In Assembly.
18. Implement the function `void changes(int *ptr)` that inverts the third byte of an integer, but only when the value of the second byte is greater than 15.
- In C.
 - In Assembly.
 - Use the Assembly function developed in b) in another Assembly function `void changes_vec(int *ptrvec, int num)` that applies the operation to a vector of integers.
19. Implement the function `void add_byte(char x, int *vec1, int *vec2)` that has as parameters the addresses of two vectors, `vec1` and `vec2` and a byte `x`. The function should add `x` to the first byte of each element of `vec1` and store the result on `vec2`. All the other bytes should remain unchanged. It is assumed that on the first element of `vec1` is the number of integers on the vector (excluding the first element).
- In C.
 - In Assembly.
20. Implement the function `int sum_multiples_x(char *vec, int x)` that given `vec`, the address of a zero terminated byte vector, and `x`, an integer, returns the sum of all the element of `vec` that are multiples of the second byte of `x`.
- In C.
 - In Assembly.
21. Implement the function `int distance(char *a, char *b)` that detects the distance between two strings. If the strings are equal, the function should return 0. If the strings differ in `n` chars the distance is `n`. If the strings do not have the same length, the distance should be -1.

Examples:

```
distance("ab","ab") -> 0
distance("ab","aa") -> 1
distance("ba","ab") -> 2
```

- In C.
- In Assembly.