Arquitectura de las Computadoras.
Plancha 2 - 2014.
Assembler de x86_64.

Nota

- Los registros `RAX`, `RCX`, `RDX`, `RSI`, `RDI`, `R8`, `R9`, `R10` y `R11` y sus subregistros **no** son preservados en llamadas a funciones de librería ni en los servicios del núcleo. Si son necesarios, lo mejor es guardarlos en el *stack*.

1) Una forma de imprimir un valor entero es realizando una llamada a la función `printf`. Ésta toma como primer argumento un puntero a caracter indicando el formato y luego una cantidad variable de argumentos que serán impresos.

La forma de llamarla en ensamblador es como sigue:

```
.data
fmt: .asciz "%ld\n"
i: .quad 0xdeadbeef
.text
.global main
main:
     movq $fmt, %rdi  # el primer argumento es el formato
     movq $1234, %rsi # el valor a imprimir
     xorq %rax, %rax  # cantidad de valores de punto flotante
     call printf
     ret
```

Modifique el código para imprimir:

- El valor del registro `rsp`.
- La dirección de la cadena de formato.
- La dirección de la cadena de formato en hexadecimal.
- El quad en el tope de la pila.
- El quad ubicado en la dirección `rsp` + 8.
- El valor i.
- La dirección de i.

2) Las instrucciones `ROL` y `ROR` rotan los bits de su operando a izquierda y derecha, dejando el bit izquierdo –respectivamente, el derecho– en la bandera de acarreo (`CARRY`) del registro de estado. Además existe la instrucción `ADC` *opo, opd* que calcula opd←*opo*+*opd*+`CARRY`.

Use esto para encontrar cuántos bits en uno tiene un entero de 64 bits(`quad`) almacenado en el registro `rax`.

3) Utilizando las instrucciones de cadena `CMPS`, `SCAS`, `REPE` y demás de la familia, implemente funciones que realicen lo siguiente:

- Busquen un caracter dentro de una cadena apuntada por `rdi`.
- Comparen dos cadenas de longitud `rcx` apuntadas por `rdi` y `rsi`.

Luego, implemente (utilizando las funciones anteriores) el algoritmo de "fuerza bruta"

```
int fuerzabruta(char *S, char *s, int lS, int ls)
{
    int i, j;
    for(i=0; i<lS-ls+1; i++)
        if(S[i]==s[0]) {
            for(j=0; j<ls && S[i+j]==s[j]; j++)
                ;
            if(j==ls) return i;
        }
    return -1;
}
```

en ensamblador.

Este ejercicio se debe realizar sin uso de stack.

4) En el programa que sigue, `funcs` implementa `void (*funcs[])()={f1, f2, f3}`. Complételo para que la línea con el comentario corresponda a `funcs[entero]()`. Use el código más eficiente.

```
   .data
   fmt:
        .string "%d"
   entero:
        .long -100
   funcs:
        .quad f1
        .quad f2
        .quad f3
   .text
   f1: movl $0,%esi; movq $fmt, %rdi; call printf; jmp fin
   f2: movl $1,%esi; movq $fmt, %rdi; call printf; jmp fin
   f3: movl $2,%esi; movq $fmt, %rdi; call printf; jmp fin

 .globl main
 main:

      pushq %rbp; movq %rsp,%rbp

 ## Leemos el entero
      movq $entero, %rsi
      movq $fmt, %rdi
      xorq %rax,%rax
      call scanf

      xorq %rax,%rax

 ## COMPLETAR CON DOS INSTRUCCIONES !!!!!!!!!!!!
      jmp *%rdx
 fin:
      movq %rbp,%rsp; popq %rbp; ret
```

5) Las funciones `setjmp` y `longjmp` permiten hacer saltos no locales. De esta forma `setjmp` "guarda" el estado de la computadora y luego `longjmp` lo restaura. Implemente `setjmp` y `longjmp`. Llámelas `setjmp2` y `longjmp2`.
Ver `/usr/include/setjmp.h` .