Plancha 2:
Ensamblador de x86_64

2020 – Arquitectura del Computador

Licenciatura en Ciencias de la Computación

1. Introducción

Esta plancha de ejercicios introduce la arquitectura de CPU x86_64 y su lenguaje de ensamblador. El ensamblador para esta arquitectura viene en dos variantes de sintaxis distintas: la de Intel y la de AT&T. Nosotros usaremos exclusivamente la segunda, por ser la que emplean por defecto las herramientas de GNU.

> **Nota:** los registros `rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10` y `r11`, y sus subregistros, **NO** son preservados en llamadas a funciones de biblioteca ni en llamadas al sistema (servicios del núcleo de sistema operativo). Si se necesita preservar los valores de estos registros, lo mejor es guardarlos en la pila.

2. Procedimiento

Resuelva cada ejercicio en computadora. Cree un subdirectorio dedicado para cada ejercicio, que contenga todos los archivos del mismo. Para todo ejercicio que pida escribir código, genere un programa completo, con su función `main` correspondiente; evite dejar fragmentos sueltos de programas.

Asegúrese de que todos los programas que escriba compilen correctamente con `gcc`. Se recomienda además pasar a este las opciones `-Wall` y `-Wextra` para habilitar advertencias sobre construcciones cuestionables en el código.

Es probable que necesiten también utilizar la opción `-no-pie` al momento de compilar si aparece un error referido a la sección `data`. Esto puede ocurrir dependiendo de la versión de compilador que tengan instalado.

3. Ejercicios

1) A continuación se presenta el código fuente de dos programas en ensamblador de x86_64. ¿Qué devuelve cada uno de ellos como código de salida? Escriba programas equivalentes para cada uno en lenguaje C.

a)

```
        .global main
main:
        movb $0xFF, %al
        ret
```

b)

```
        .global main
main:
        movb $0xFE, %al
        movb $-1, %bl
        addb %bl, %al
        incb %bl
        ret
```

2) ¿Cómo se puede obtener el código de salida de un programa? Explique dos maneras distintas.

3) En el ejercicio 1 de la plancha 1, se pidió completar expresiones que usaran operadores de bits y números literales del lenguaje C.

a) Traduzca estas expresiones a ensamblador, escribiendo un programa para cada una de ellas. Asegúrese de usar registros de un tamaño adecuado. Ejecute los programas con GDB para analizar el valor resultante en cada registro.

b) ¿Es necesario usar GDB en este caso? ¿Podría utilizarse el código de salida como en los ejercicios anteriores?

4) El producto tiene dos versiones: mul (producto unsigned) e imul (producto signed). Complete el siguiente fragmento de código para que el resultado final quede en la porción correspondiente de rax (i.e.: en el primer caso, si se hace `print $rax` desde gdb se debe obtener -2).

```
.globl main
main:
        movl $-1,%eax # Solo para este tamaño el mov pone en 0
                      # la parte alta del registro.
        movl $2, %ecx
        imull %ecx

        #completar para que el resultado correcto como signed quede en rax
        ...
        ...

        xorq %rax,%rax
        movw $-1,%ax
        movw $2, %cx
        mulw %cx

        #completar para que el resultado correcto como unsigned int
        #quede en eax
        ...
        ...

        ret
```

5) La función factorial se puede escribir en lenguaje C con recursión:

```
unsigned long fact1(unsigned long x)
{
    if (x <= 1) {
        return x;
    }
    return x * fact1(x - 1);
}
```

y también con un bucle:

```
unsigned long fact2(unsigned long x)
{
    unsigned long acc = 1;
    for (; x > 1; x--) {
        acc *= x;
    }
    return acc;
}
```

Implemente ambas versiones en ensamblador. Se sugiere empezar por `fact2`. Acompáñelas del siguiente archivo en C para probarlas:

```
#include <stdio.h>

unsigned long fact1(unsigned long);
unsigned long fact2(unsigned long);

int main(void)
{
    unsigned long x;
    scanf("%lu", &x);
    printf("fact1: %lu\n", fact1(x));
    printf("fact2: %lu\n", fact2(x));
    return 0;
}
```

Deberá entonces mantener dos o más archivos separados: uno para la función `main` en C y uno o más para las funciones en ensamblador. Para generar el binario ejecutable, debe pasar los nombres de todos archivos de implementación como argumentos de `gcc`:

```
$ gcc fact.s main.c
```

6) Una forma de imprimir un valor entero es realizando una llamada a la función `printf`. Esta toma como primer argumento una cadena de C (las cuales se representan como un puntero a caracter) indicando el formato y luego una cantidad variable de argumentos que serán impresos. La signatura en C es la siguiente:

```
int printf(const char *format, ...);
```

La forma de llamarla en ensamblador es como sigue:

```
        .data
format: .asciz "%ld\n"
i:      .quad 0xDEADBEEF
```

```
        .text
        .global main
main:
        movq $format, %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
```

Agregue más llamadas a `printf` en el código para imprimir:

a) El valor del registro `rsp`.

b) La dirección de la cadena de formato.

c) La dirección de la cadena de formato en hexadecimal.

d) El *quad* en el tope de la pila.

e) El *quad* ubicado en la dirección `rsp` + 8.

f) El valor `i`.

g) La dirección de `i`.

7) Las instrucciones `rol` y `ror` toman dos operandos:

```
rol cantidad_a_rotar, registro
ror cantidad_a_rotar, registro
```

y rotan los bits del segundo operando a izquierda y derecha, respectivamente, la cantidad de veces indicada en el primer operando. Además, dejan el bit izquierdo o el derecho, según corresponda, del segundo operando en la bandera de acarreo (*carry*) del registro de estado.

Además, existe la instrucción `adc` (*add with carry*), que toma dos operandos:

```
adc op_origen, op_destino
```

y calcula *op_destino* ← *op_origen* + *op_destino* + *acarreo*.

a) Use estas instrucciones para permutar la mitad alta y baja de un entero de 64 bits (*quad*) almacenado en el registro `rax`.

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

8)

a) Implementar en ensamblador un programa que busque un caracter dentro de una cadena apuntada por `rdi`.

b) Implementar en ensamblador un programa que compare dos cadenas de longitud `rcx` apuntadas por `rdi` y `rsi`.

c) Una vez implementados y testeados los programas anteriores, utilizarlos para implementar en ensamblador el algoritmo de "fuerza bruta":

```
int fuerza_bruta(const char *S, const char *s, unsigned lS, unsigned ls)
{
    unsigned 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;
}
```

Escriba una función `main` en C que llame a la función `fuerza_bruta`. Deberá mantener dos archivos separados.

Este ejercicio se debe realizar sin guardar variables locales en la pila.

9) 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

        .global main
main:
        pushq %rbp
        movq %rsp, %rbp

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

        xorq %rax, %rax

        # COMPLETE CON DOS INSTRUCCIONES.
        jmp *%rdx
fin:
        movq %rbp, %rsp
        popq %rbp
```

```
        ret
```

10) Las funciones `setjmp` y `longjmp` permiten hacer saltos no locales. Sus signaturas en C se encuentran en la cabecera `setjmp.h` y son las siguientes:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

`setjmp` "guarda" el estado de la computadora en el argumento `env` y luego `longjmp` lo restaura. Reimplemente `setjmp` y `longjmp`; llámelas `setjmp2` y `longjmp2`.

11) Dado el siguiente programa en C (`código/memoria/disposición_pila.c` en Subversion):

```
#include <stdio.h>

int f(char a, int b, char c, long d,
      char e, short f, int g, int h)
{
    printf("a: %p\n", &a);
    printf("b: %p\n", &b);
    printf("c: %p\n", &c);
    printf("d: %p\n", &d);
    printf("e: %p\n", &e);
    printf("f: %p\n", &f);
    printf("g: %p\n", &g);
    printf("h: %p\n", &h);
    return 0;
}

int main(void)
{
    return f('1', 2, '3', 4, '5', 6, 7, 8);
}
```

a) Realice un diagrama de la pila de cuando se está ejecutando `f`. Indique en el diagrama la ubicación y el espacio utilizado por cada argumento.

b) Indique la dirección dentro de la pila en donde está almacenada la dirección de retorno de `f` y si es posible verifique con GDB. Sugerencia: utilice el comando `si`, para avanzar de a una instrucción.

12) Corrutinas:

a) Compile y ejecute el código de corrutinas cooperativas:

```
$ cd código/corrutinas_cooperativas
$ gcc corrutinas.c guindows.c -o corrutinas
$ ./corrutinas
```

b) Agregue una nueva corrutina:

```
task t3;
```

```
void ft3(void)
{
    for (unsigned i = 0; i < 5000; i++) {
        printf("t3: i=%u\n", i);
        TRANSFER(t3, t1);
    }
    TRANSFER(t3, taskmain);
}
```

Nota: se debe reservar espacio en la pila (`stack` también para `ft3`). Haga que `ft3` realice una iteración luego de que `ft2` lo haya hecho.

c) Modifique las tres corrutinas para que impriman la dirección de una variable local antes de comenzar a iterar. Compare las direcciones mostradas.