Plancha 3
Arquitectura y lenguaje ensamblador de x86_64

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) Dados los siguientes números expresados en complemento a dos utilizando 8 bits:

$$N_1 = 100, \qquad N_2 = 120, \qquad N_3 = -63, \qquad N_4 = -56, \qquad N_5 = -91$$

Indicar el resultado de las siguientes operaciones y de las banderas `CF` y `OF` luego de realizarse cada una de las mismas, haciendo un código en Assembler x86-64 y utilizando GDB:

a) $S_1 = N_1 + N_2$

b) $S_2 = N_3 + N_4$

c) $S_3 = N_3 + N_5$

d) $S_4 = N_2 - N_4$

e) $S_5 = N_5 - N_2$

4) Compilar el siguiente código y ejecutar con GDB. Ir viendo cómo se modifica el contenido del registro `rax` luego de la ejecución de cada línea. ¿Qué conclusiones se pueden sacar?

```
.text
.global main
main:
    movq $-1, %rax
    movb $0, %al
    movw $0, %ax
    movl $0, %eax
    retq
```

5) Compilar el siguiente código en Assembler y observar el error que devuelve. ¿Qué conclusión se puede sacar?

```
.global main
main:
    movq $0x1122334455667788, %rax
    addq $0x1122334455667788, %rax
    ret
```

6) 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?

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 $op\_destino \leftarrow op\_origen + op\_destino + acarreo$.

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) 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
```

9) Dado el siguiente código en Assembler X86-64:

```
.data
a:      .word -1
b:      .byte 0xaa, 99
c:      .float 1.5
str:    .asciz "Hola mundo\n"


.text
.global main
main:
leaq a, %rdx
.................    # ax = 0xffff
.................    # al = 0x63
.................    # esi = "aloH"
.................    # edi = 0x3fc00000
movl %esi, (%rdx)
retq
```

a) Completar el siguiente diagrama de memoria, indicando los valores almacenados, las direcciones de memoria y los nombres de las etiquetas antes de ejecutar el código. Utilizar GDB.

Etiqueta	**Dirección de memoria**	**Valor almacenado**
...	...	...
...	...	...
...	...	...
⋮	⋮	⋮
⋮	⋮	⋮

b) Completar lo que falta en las líneas de puntos del código anterior para que cada línea de código se corresponda con el valor del registro indicado en el comentario correspondiente sin usar movimientos de valores inmediatos.

c) Repetir el diagrama de memoria luego de ejecutar la línea de código `movl %esi, (%rdx)`.

10) Dado el siguiente código en Assembler x86-64:

```
.data
a:   .long 1, 2, 3, 4
g:   .quad 0x1122334455667788
msg: .asciz "Imprime %c\n"

.text
```

```
.global main
main:
subq    $8, %rsp                    # rsp =.....................
movq    $g, %rax                    # rax =.....................
movl    g+4, %eax                   # rax =.....................
movq    $3, %rbx                    # rbx =.....................
movb    a(,%rbx,4), %al             # rax =.....................
leaq    msg, %rdi                   # rdi =.....................
movb    (%rdi, %rbx, 2), %sil       # sil =.....................
xorl    %eax, %eax                  # rax =.....................
call    printf                      # rax =.....................
addq    $8, %rsp                    # rsp =.....................
ret
```

a) Dibujar el esquema de memoria a partir de la etiqueta `a` indicando el contenido de memoria para cada byte.

b) Indicar en el esquema de memoria la dirección correspondiente a cada etiqueta. Asumir que la dirección de `a` es `0x404028`.

c) Indicar el valor de los registros indicados en cada uno de los comentarios luego de ejecutarse la línea correspondiente asumiendo que el valor inicial del registro `rsp` es `0x7fffffffebc8`.

d) Indicar lo que se imprimiría en pantalla luego de ejecutarse la instrucción `call printf`.

11) Implementar un programa en Assembler para invertir un arreglo de bytes de longitud conocida usando el *stack*.

Ayuda: Esto puede ser logrado "apilando" los valores de a uno en el *stack* y luego ir "desapilando" en sentido inverso.

12) Dadas la siguiente declaración de variables:

```
.data
list: .long 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
length: .byte 10
```

a) Implementar un programa en Assembler x86-64 para sumar el arreglo de números mostrado en la declaración de variables.

b) Actualice el programa de la pregunta anterior para encontrar el máximo, el mínimo y el promedio del arreglo de números.

c) Imprimir los resultados de los ítems anteriores por pantalla.

d) Modificar el ítem b) usando instrucciones de movimiento de datos condicionales (familia de instrucciones `CMOVcc`, ver Manual de Intel).

Recomendaciones:

- Realizar los cálculos requeridos en funciones separadas para que quede un código claro y ordenado.

13) 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
```

14) Realizar un programa en Assembler x86-64 que sume una cantidad variable de argumentos de entrada elevados al cuadrado y muestre el resultado por pantalla: $S = a_1^2 + a_2^2 + \cdots + a_n^2$. Además, debe imprimir la cantidad de argumentos ingresados (n). Por ejemplo, si se ejecuta de la siguiente manera:

```
./suma 1 2 3 4
```

debe imprimir

```
"Cantidad de argumentos ingresados: 4. La suma es 30."
```

Por el contrario, si no se ingresan argumentos debe imprimir el mensaje

```
"Ingrese al menos un argumento"
```

Ayuda: Se puede usar la función `int atoi(const char *str)` que devuelve el argumento de entrada de tipo cadena de caracteres convertido en un valor tipo entero.

15) El programa que sigue, `funcs`, implementa `void (*funcs[])() = {f1, f2, f3}`. Complételo para que la línea con el comentario corresponda a `funcs[entero]()` de manera que si se ingresa un 0 por teclado imprime un cero, si se ingresa un 1 imprime un 1 y si se ingresa un 2 imprime un dos. 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
```

16) Dado el siguiente programa en C:

```
#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 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.

17) 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`.

18) Dado el siguiente código en C:

```
#include <stdio.h>
#include <stdlib.h>

int calculo(char a, char b, char c, int d, char e, float f, double g, int h, int i, int j){
    return (a+c+d)*(i+j);
}

int main(){
    int e=5;
    return calculo(1, 2, 3, 4, 5, 1.0, 2.0, 6, 7, 8);
}
```

a) Completar el siguiente código en Assembler x86-64 de la función `calculo` de manera que resulten equivalentes:

```
calculo:
    pushq   %rbp
    movq    %rsp, %rbp
    COMPLETAR
    movq    %rbp, %rsp
    popq    %rbp
    ret
```

b) Dibujar el diagrama de pila mostrando el estado de la misma antes de ejecutarse el epílogo de la función. Asumir que `rbp=0x7fffffffebc0` y `rsp=0x7fffffffebb0` antes del llamado a `calculo` y el siguiente fragmento de código de la función llamante:

```
...
0x000000000040115c:     push   %rbp
0x000000000040115d:     mov    %rsp, %rbp
0x0000000000401160:     sub    $0x10, %rsp
...
0x00000000004011a5:     call   0x401106 <calculo>
0x00000000004011aa:     add    $0x10, %rsp
0x00000000004011ae:     leave
0x00000000004011af:     ret
```

c) Escribir un código equivalente en Assembler para la instrucción `leave`.

d) Explicar la utilidad de las instrucciones `sub $0x10, %rsp` y `add $0x10, %rsp` en el código anterior. ¿Es realmente necesaria la instrucción `add $0x10, %rsp`?

19) Dado el siguiente código en lenguaje C:

```
#include <stdio.h>
int a=0x7fffffff, b=1;
```

```
int foo1(){
    printf("Ejecutando foo1...\n");
    a=a+(b<<31);
    printf("0x%x\n", a);
    return a;
}

int foo2(){
    printf("Ejecutando foo2...\n");
    a=a+b;
    printf("0x%x\n", a);
    return a;
}

int main(){
    printf("0x%x 0x%x\n ", foo1(), foo2());
    return 0;
}
```

a) Indicar lo que se imprime en pantalla al ser ejecutado. Explicar.

b) Indicar el valor de la variable `a` y de las banderas `CF` y `OF` luego de realizarse cada una de las sumas.

c) Escribir un código equivalente en Assembler x86-64.

d) Dibujar un esquema que muestre la evolución de la pila.

Ayuda: Utilizar GDB.

20) Dado el siguiente código en C:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc < 2){
        printf("La cantidad de argumentos es insuficiente\n");
    } else {
        int suma=0;
        for (int i=1; i<argc; i++){
            suma = suma + atoi(argv[i]);
        }
        printf("Cantidad de argumentos: %d, suma de los argumentos: %d\n", argc, suma);
    }
    return 0;
}
```

Escribir un código equivalente en Assembler x86-64.

Nota: La función `atoi` de la librería `stdlib.h` convierte una cadena de caracteres a su valor numérico equivalente (entero). Toma como parámetro de entrada la cadena a convertir y devuelve el valor numérico correspondiente.

21) (Opcional) Implemente en ensamblador dos funciones que realicen lo siguiente completando el código de ejemplo:

a) Busque un caracter dentro de una cadena apuntada por `rdi`. Código de ejemplo (guardarlo en `buscachar.s`):

```
.data
cadena: .asciz "123456"

.text
.global main
main:
    movq $cadena, %rdi
    movb $'6',%sil  #Probar luego con diferentes valores (1,a,5....)

.global busca
busca:
    # completar aqui
    # si el char está en la cadena en rax debe quedar
    # la dirección en donde se encontró el char.
    # Si no, en rax debe quedar el valor -1
    ret
```

b) Comparen dos cadenas de longitud `rdx` apuntadas por `rdi` y `rsi`. Código de ejemplo (guardarlo en `comparastrings.s`):

```
.data
cadenalarga: .asciz "123456"
cadenacorta: .asciz "123" #=>1 , 124=>0, 456=>0

.text
.global main
main:
    movq $cadenalarga, %rdi
    movq $cadenacorta, %rsi
    movl $3,%edx  #debe ser la longitud de cadenacorta

.global compara
compara:
    #retorna 1 si las cadenas son iguales en los primeros %edx caracteres
    #retorna 0 si son distintas
```

Una vez implementadas las funciones anteriores y probadas con diferentes valores las utilizaremos para implementar el algoritmo de "fuerza bruta". Una versión en C de tal algoritmo es la siguiente (archivo `fuerza_bruta_c.c`):

```
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;
}
```

c) Asegúrese de entender qué hace la función anterior `fuerza_bruta`: haga un programa (guárdelo en `main.c`) que llame a dicha función y tome como argumentos las dos cadenas que llegarán a `fuerza_bruta`. Incluya al principio de `main.c` la declaración de `fuerza_bruta` para que el programa compile:

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

Puede compilar el programa completo usando:

```
gcc fuerza_bruta_c.c main.c -o main -g
```

O puede compilar los módulos en forma individual a medida que sea necesario ....:

```
gcc -g -c fuerza_bruta_c.c # esto produce fuerza_bruta_c.o
gcc -g -c main.c           # esto produce main.o
```

.... y luego compilar el programa completo usando

```
gcc main.o fuerza_bruta_c.o -o main
```

Pruebe el programa con diferentes entradas y estudie el funcionamiento de `fuerza_bruta`

d) Reescriba la función en C `fuerza_bruta` para que utilice las funciones de los apartados a) y b). Llámela `fuerza_bruta2` y guárdela en `fuerza_bruta2_c.c`. Al principio de ese archivo deberá declarar las dos funciones anteriores:

```
char *busca(const char *s1, char c, unsigned l);
int compara(const char *s1,const  char *s2, unsigned l);
```

Para compilar deberá:

- Borrar (o mejor: comentar) las referencias a `main` en los archivos "`.s`"
- Reemplazar en `main.c`: `fuerza_bruta` por `fuerza_bruta2`
- Compilar con el siguiente comando (también se puede hacer compilación de cada módulo individualmente como se hizo en el apartado anterior):

```
gcc -g main.c fuerza_bruta2_c.c buscachar.s comparastrings.s
```

Pruebe el programa con diferentes entradas.

e) Reemplace la implementación en C de `fuerza_bruta2` por una implementación en assembler de la misma función.

22) (Opcional) 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`.

23) (Opcional)

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.