# Ejemplo sobre el uso de la instrucción `lea`

Arquitectura del Computador - LCC - FCEIA-UNR

Agosto 2021

En este ejemplo se pretende mostrar el uso de la instrucción `lea`. La instrucción `lea` (*load effective address*) se usa para poner una dirección de memoria en el destino. Este ejemplo también es útil para familiarizarse con GDB.

Dado el siguiente código en Assembler X86-64 en el archivo `lea_ejemplo.s`:

```
.data

a: .quad 0x1122334455667788

.text
.global main

main:
    leaq a, %rax    # línea 1
    movq $a, %rax   # Línea 2
    retq
```

Compilar utilizando la opción `-g` para poder *debuggear* utilizando GDB:

```
$ gcc -g lea_ejemplo.s
```

Luego ejecutar utilizando GDB:

```
gdb ./a.out
```

Colocar un breakpoint en `main`:

```
(gdb) br main
Breakpoint 1 at 0x4004b6: file ejemplo_lea.s, line 9.
```

Ejecutar con el comando `run`:

```
(gdb) r
```

Ejecutar la primera línea con el comando `next`:

```
(gdb) n
```

Observar el contenido del registro `rax`:

```
(gdb) i r rax
rax            0x600880 6293632
```

Vemos que en el registro `rax` quedó almacenada la dirección de la etiqueta `a`, lo cual podemos verificar de manera adicional mediante el comando:

```
(gdb) info address a
Symbol "a" is at 0x600880 in a file compiled without debugging.
```

Ahora veamos que sucede cuando ejecutamos la siguiente línea:

```
(gdb) n
(gdb) i r rax
rax            0x600880 6293632
```

Vemos que la línea 2 es equivalente a la línea 1.

Como conclusión, hemos ejecutado dos línea de código equivalentes pero que utilizan instrucciones diferentes. En efecto, si bien las instrucciones en las líneas 1 y 2 produjeron el mismo efecto, la instrucción `lea` es diferente a la dirección `mov` y permite realizar operaciones más complejas. Recordemos que la forma general de la instrucción `lea` es

```
leaq displacement(%base, %offset, multiplier), %dest
```

la cual corresponde a `%dest = displacement + %base + %offset * multiplier` donde `displacement` es una constante entera, `multiplier` es 2, 4 u 8 y `%dest`, `%offset` y `%base` son registros. Algunos de los operandos pueden no estar.

Ejemplos:

```
movq $100, %rax
movq $4, %rbx
leaq 16(%rax, %rbx, 2), %rcx        #---> rcx = 124
leaq 16(%rax, %rbx,), %rcx          #---> rcx = 120
leaq (%rax, %rbx,), %rcx            #---> rcx = 104
leaq (%rax), %rcx                   #---> rcx = 100
```

La instrucción `lea` se puede utilizar para realizar cálculos relativamente complejos[1]. sin emabrgo, ese no es el propósito principal del la instrucción. El conjunto de instrucciones x86 fue diseñado para admitir lenguajes de alto nivel como Pascal y C, donde las matrices, especialmente las matrices de enteros, o estructuras pequeñas, son comunes. Consideremos, por ejemplo, una estructura que representa las coordenadas (x, y) de un punto:

```
struct Point
{
     int xcoord;
     int ycoord;
};
```

Ahora supongamos la siguiente declaración:

```
int y = points[i].ycoord;
```

donde `points[]` es una matriz de `Point`. Suponiendo que la base de la matriz ya está en `rbx`, la variable `i` está en `rax`, y `xcoord` e `ycoord` son cada uno de 32 bits (por lo que `ycoord` está en un desplazamiento de 4 bytes en la estructura), esta declaración se puede compilar como:

```
movq 4(%rbx, %rax, 8), %rdx
```

que cargará `y` en `rdx`. El factor de escala de 8 se debe a que cada punto tiene un tamaño de 8 bytes.

Ahora consideremos la siguiente expresión usando el operador "dirección de" &:

```
int *p = &points[i].ycoord;
```

[1]Es importante señalar que la instrucción `lea` no modifica el registro de banderas a diferencia de las instrucciones aritméticas

En este caso, no deseamos el valor de `ycoord`, sino su dirección. Aquí es donde entra la principal utilidad de la instrucción `lea` (*load effective addresss*). En lugar de un `mov`, el compilador puede generar:

```
leaq 4(%rbx, %rax, 8), %rsi
```

que cargará la dirección en `rsi`.