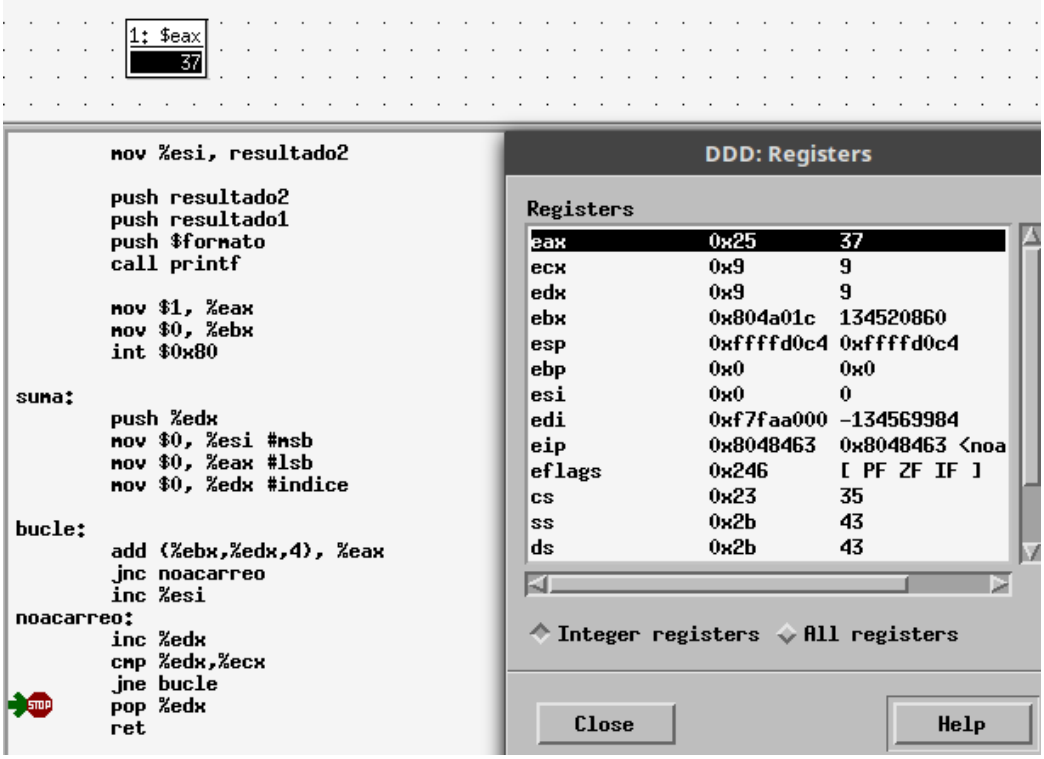
Sesión de depuración suma.s

1. ¿Cual es el contenido de EAX justo antes de ejecutar la instruccion RET, para esos componentes de lista concretos? Razonar la respuesta, incluyendo cuánto valen 0b10, 0x10, y (.lista)/4



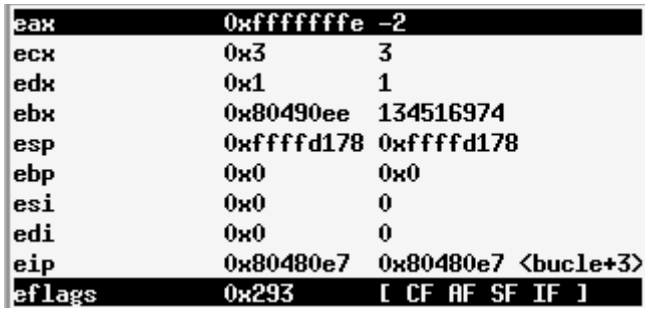
El contenido en EAX justo antes de la instrucción RET es 37 en decimal, que se corresponde con 0x25 en hexadecimal
0b10 → Esta en binario y su valor es 2 en decimal
0x10 → Esta en hexadecimal y su valor es 2 en decimal
(.-lista)/4 → Es 9 debido a que tenemos 9 elementos en la lista de 32 bits cada uno = 4 Bytes cada uno, por lo que la operación '.' (desde aquí) '-lista' obtiene 36 B y dividido entre 4 obtenemos el numero de elementos de la lista.

2.¿ ¿Qué valor en hexadecimal se obtiene en resultado si se usa la lista de 3 elementos: .int 0xffffffff, 0xffffffff,0xffffffff? ¿Por qué es diferente del que se obtiene haciendo la suma a mano? NOTA: Indicar qué valores va tomando EAX en cada iteración del bucle, como los muestra la ventana Status->Registers, en hexadecimal y decimal (con signo). Fijarse también en si se van activando los flags CF y OF o no tras cada suma. Indicar también qué valor muestra resultado si se vuelca con Data->Memory como decimal (con signo) o unsigned (sin signo).

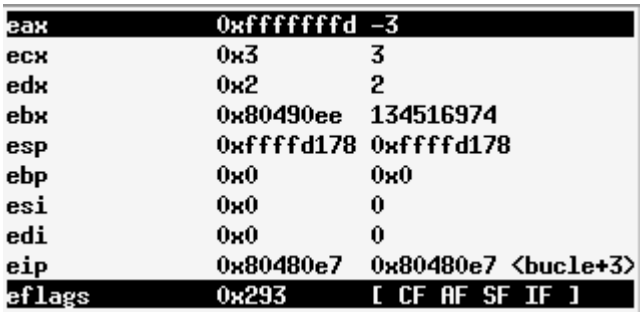
El resultado calculado a mano sería 0x2FFFFFFFD pero como no estamos teniendo el cuanta el bit de acarreo el resultado obtenido depurando el programa es el siguiente:

1ª iteración →

2ª iteración →



3ª iteración →



Como decimal con signo el valor de resultado en decima es -3, y sin simbolo 4294967293 (2^32-3).

```
(gdb) x /1dw &resultado
0x80490fe:      -3
(gdb) x /1dg &resultado
0x80490fe:      4294967293
```

3.¿Que direccion se le ha asignado a la etiqueta suma? ¿Y a bucle? ¿Como se ha obtenido esa informacion?

```
Dump of assembler code for function _start:
   0x080480b8 <+0>:     mov    $0x80490ee,%ebx
   0x080480bd <+5>:     mov    0x80490fa,%ecx
   0x080480c3 <+11>:    call   0x80480d9 <suma>
   0x080480c8 <+16>:    mov    %eax,0x80490fe
```

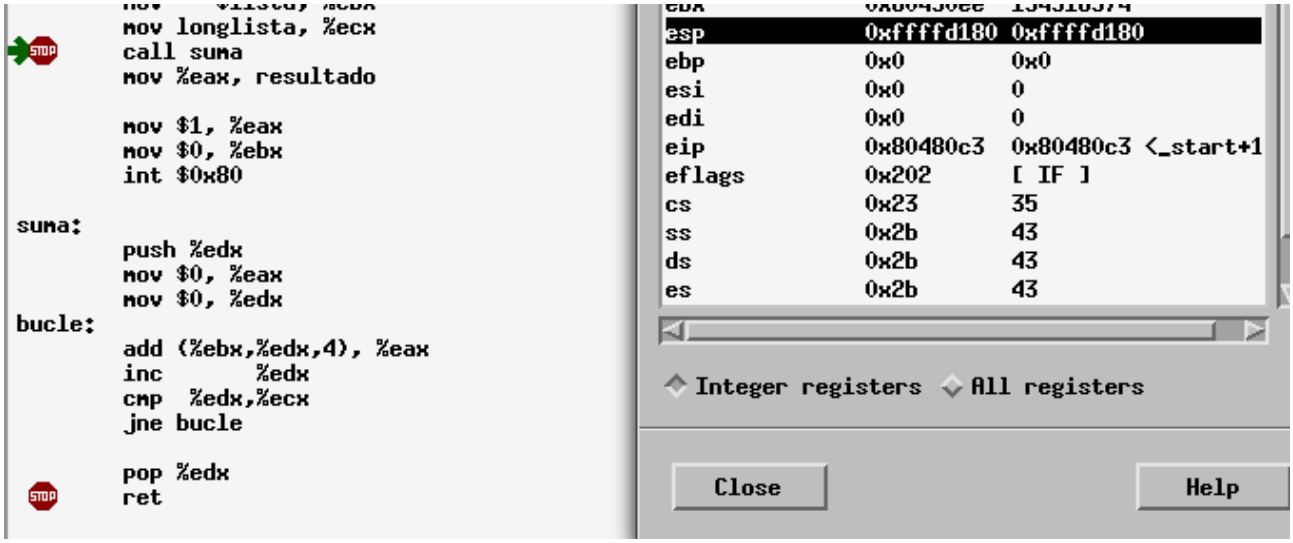
La etiqueta suma tiene la dirección de memoría 0x80480d9 y la de bucle es 0x804080e4. Esta informacion se puede obtener desde la vista de codigo maquina si se realiza un CALL a la etiqueta o posicionando el raton sobre la etiqueta.

4.¿Para que usa el procesador los registros EIP y ESP?

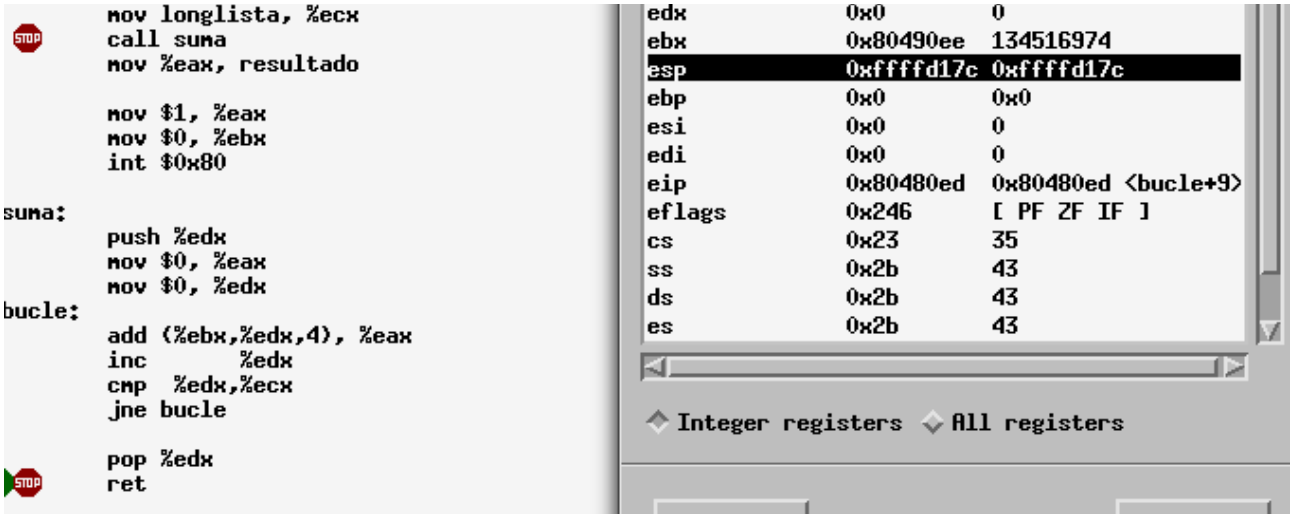
EIP contiene la dirección de memoría de la próxima instrucción a ejecutar y ESP el tope de la pila.

5.¿Cual es el valor de ESP antes de ejecutar CALL, y cual antes de ejecutar RET? ¿En cuanto se diferencian ambos valores? ¿Por qué? ¿Cual de los dos valores de ESP apunta a algún dato de interés para nosotros? ¿Cual es ese dato?

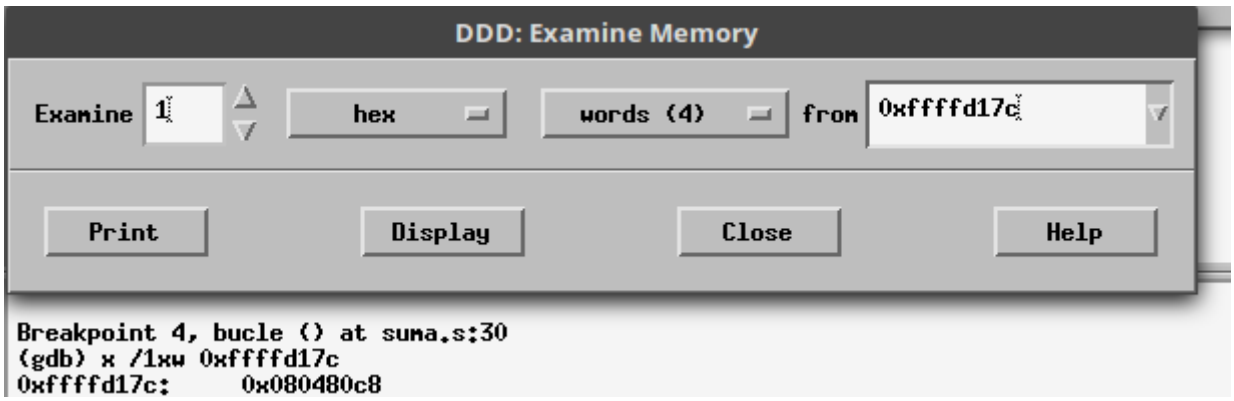
El valor de ESP antes de ejecutar CALL es 0xffffd180.



Y antes de ejecutar RET es 0xffffd17c.



Antes de ejecutar el CALL, ESP contiene una dirección que no nos interesa, pero al llamar al colar en el tope de la pila se guarda la direccion de memoría que contiene la siguiente instrucción, que en este caso es 0xffffd17c, cuyo valor en hexadecimal es el 0x080480c8, que es la dirección de la proxima instrucción '*mov %eax, resultado*'.



6.¿Que registros modifica la instruccion CALL? Explicar por qué necesita CALL modificar esos registro.

ANTES DE CALL

```
eax            0x0        0
ecx            0x3        3
edx            0x0        0
ebx            0x80490ee  134516974
esp            0xffffd180 0xffffd180
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x80480c3  0x80480c3 <_start+1
eflags         0x202      [ IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
```

DESPUES DE CALL

```
Registers
eax            0x0        0
ecx            0x3        3
edx            0x0        0
ebx            0x80490ee  134516974
esp            0xffffd17c 0xffffd17c
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x80480d9  0x80480d9 <suma>
eflags         0x202      [ IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
```

Modifica los registros ESP y EIP. Necesita modificar dichos registros para saber volver saber continuar por la nueva siguiente instrucción (EIP) y saber volver a la instrucción siguiente después del CALL (ESP).

7.¿Que registros modifica la instruccion RET? Explicar por qué necesita RET modificar esos registro.

ANTES DE RET

```
eax            0xfffffffd -3
ecx            0x3        3
edx            0x0        0
ebx            0x80490ee  134516974
esp            0xffffd17c 0xffffd17c
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x80480ed  0x80480ed <bucle+9>
eflags         0x246      [ PF ZF IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
```

DEPUES DE RET

```
eax            0xfffffffd -3
ecx            0x3        3
edx            0x0        0
ebx            0x80490ee  134516974
esp            0xffffd180 0xffffd180
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x80480c8  0x80480c8 <_start+1
eflags         0x246      [ PF ZF IF ]
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
```

Modifica los registros ESP y EIP. Igual que lo mencionado en el ejercicio anterior.

9.¿Cuantas posiciones de memoria ocupa la instroccion mov $0, %edx? ¿Y la instruccion inc %edx? ¿Cuales son sus respectivos codigos maquina? Indicar como se han obtenido. NOTA: en los volcados Data->Memory se puede usar una dirección hexadecimal 0x..... para indicar la direccion del volcado. Recordar la ventana View->Machine Code Window. Recordar tambien la herramienta objdump

La instrucción ‘mov $0, %edx’ ocupa 5 bits.

```
        mov $0, %eax
80480da:        b8 00 00 00 00          mov    $0x0,%eax
        mov $0, %edx
80480df:        ba 00 00 00 00          mov    $0x0,%edx
```

La instrucción ‘inc %edx’ ocupa 3 bits.

```
       add (%ebx,%edx,4), %eax
80480e4:       03 04 93                add    (%ebx,%edx,4),%eax
       inc       %edx
80480e7:       42                      inc    %edx
```

En la venta de código máquina podemos comprobar lo anterior.

```
Dump of assembler code for function suma:
=> 0x080480d9 <+0>:     push   %edx
   0x080480da <+1>:     mov    $0x0,%eax
   0x080480df <+6>:     mov    $0x0,%edx
```

```
Dump of assembler code for function bucle:
=> 0x080480e4 <+0>:     add    (%ebx,%edx,4),%eax
   0x080480e7 <+3>:     inc    %edx
   0x080480e8 <+4>:     cmp    %edx,%ecx
   0x080480ea <+6>:     jne    0x80480e4 <bucle>
   0x080480ec <+8>:     pop    %edx
   0x080480ed <+9>:     ret
```

10.¿Que ocurriría si se eliminara la instrucción RET? Razonar la respuesta. Comprobarlo usando ddd

Lo que ocurriría sería que el programa no encontraría la instrucción de retorno una vez finalizada la función suma, con lo que cuando terminara la ejecución de suma acabaría el programa consigo devolviendo un ‘Illegal instruction’ debido a que no podemos realizar la llamada al sistema para salir (sys_exit).

```
juanka1995@juanka-laptop ~/practicas/ec $ ./suma
Illegal instruction
```

```
        jne bucle

        pop %edx
        #ret
```

```
Dump of assembler code from 0x80480ed to 0x80481ed:
=> 0x080480ed:  (bad)
   0x080480ee:  (bad)
   0x080480ef:  (bad)
   0x080480f0:  (bad)
   0x080480f1:  (bad)
   0x080480f2:  (bad)
   0x080480f3:  (bad)
   0x080480f4:  (bad)
```