

## 5.1 Sumar N enteros sin signo de 32 bits sobre dos registros de 32 bits usando uno de ellos como acumulador de acarreo (N≈16)

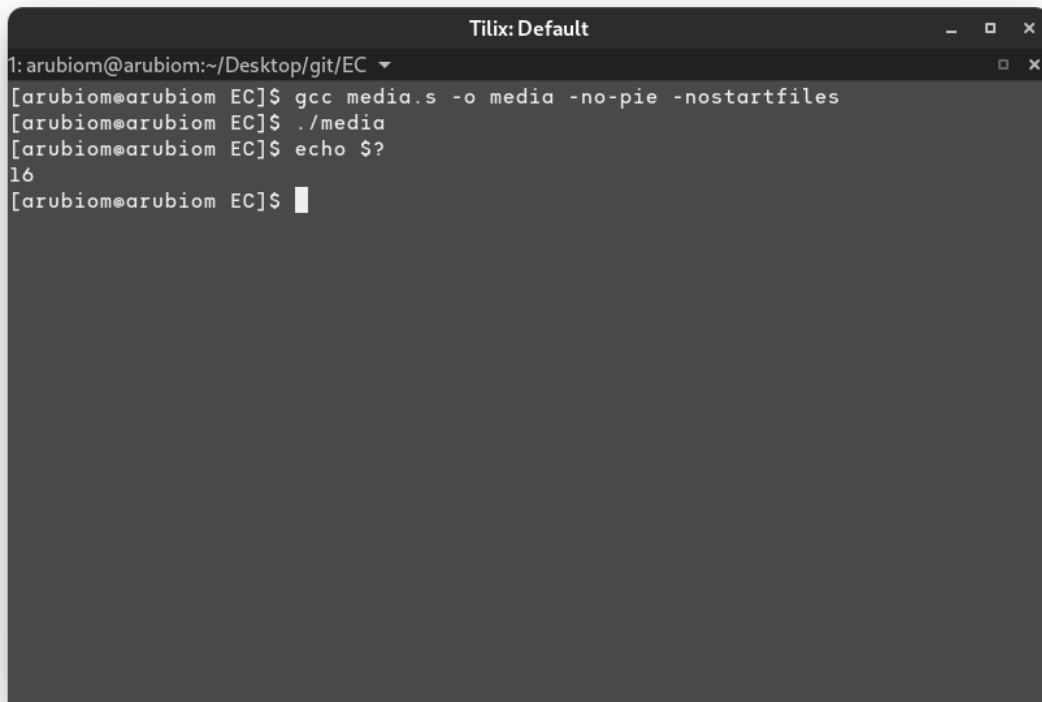
En primer lugar vamos a cambiar el tamaño de la lista por una de 16 elementos, y que repita 16 veces el número 1. Para ello nos vamos a suma.s y editamos lo siguiente para el archivo media.s:

```
.section .data
lista:      .int 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
longlista:  .int  (.-lista)/4
```

Ahora tenemos una lista de 16 veces el entero 1. Veamos que obtenemos al ejecutarlo. Para ello vamos a crear el binario ejecutable con gcc usándola orden:

```
gcc media.s -o media -no-pie -nostartfiles
```

Al realizar esto y ejecutar obtenemos el siguiente resultado:



```
Tilix: Default
1:arubiom@arubiom:~/Desktop/git/EC
[arubiom@arubiom EC]$ gcc media.s -o media -no-pie -nostartfiles
[arubiom@arubiom EC]$ ./media
[arubiom@arubiom EC]$ echo $?
16
[arubiom@arubiom EC]$
```

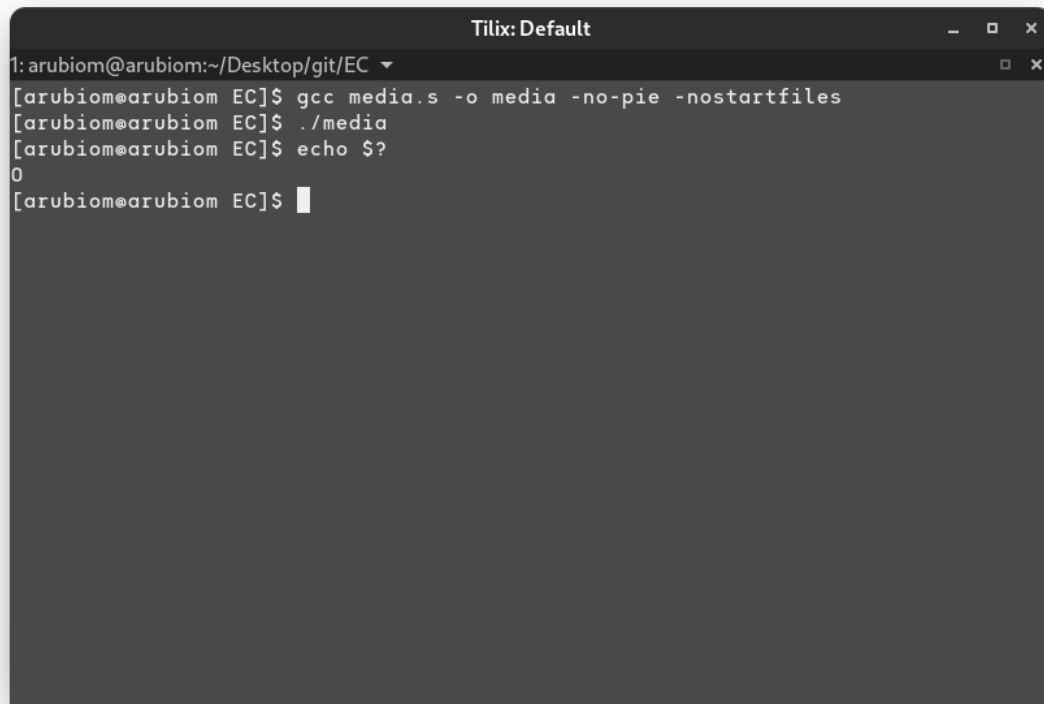
Donde podemos observar como obtenemos el resultado esperado.

Tenemos que el valor mínimo para que no se produzca acarreo al sumar 16 veces es el 0x1000 0000. Esto es fácil de ver sumando mentalmente pues en las primeras 15 sumas el resultado que tendremos será 0xf000 0000, el cual todavía está permitido, pero si volvemos a sumar 0x1000 0000 el resultado que tenemos sería 0x0001 0000 0000, que ya son más de 32 bits, y entonces truncaría en los 32 primeros bits, quedando de resultado el 0x0000 0000. Podemos ver como esto es cierto experimentalmente haciendo lo siguiente:

Primero cambiamos los valores que queremos sumar en media.s:

```
.section .data
lista:      .int 0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000,
0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000,
0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000
longlista:  .int  (.-lista)/4
```

Ejecutamos como antes:



```
Tilix: Default
1:arubiom@arubiom:~/Desktop/git/EC
[arubiom@arubiom EC]$ gcc media.s -o media -no-pie -nostartfiles
[arubiom@arubiom EC]$ ./media
[arubiom@arubiom EC]$ echo $?
0
[arubiom@arubiom EC]$
```

Donde vemos que efectivamente el resultado es el esperado.

Sin embargo si el valor que usamos fuera 0x0fff ffff no se produciría acarreo, y esto también es fácil de ver mentalmente pues si sumamos las primeras 15 tenemos el valor 0xefff fff1, que si le sumamos el dato una vez más obtenemos 0xffff fff0, el cual es un valor que no produce acarreo. Veámoslo experimentalmente:

```
.section .data
lista:      .int 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff,
0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff,
0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff
longlista:  .int  (.-lista)/4
```

```
Tilix: Default
1:arubiom@arubiom:~/Desktop/git/EC
[arubiom@arubiom EC]$ gcc media.s -o media -no-pie -nostartfiles
[arubiom@arubiom EC]$ ./media
[arubiom@arubiom EC]$ echo $?
240
[arubiom@arubiom EC]$
```

Sin embargo, si pasamos 0xffff fff0 a binario sería 0b1111 1111 1111 1111 1111 1111 1111 0000, que si ya finalmente lo pasamos a decimal sería 4294967280, y sin embargo vemos como se devuelve el valor 240. Vamos a depurarlo paso a paso para ver que está ocurriendo. Para ello primero compilamos indicando al compilador que no optimice el binario para que este y nuestro código sean iguales. Hacemos:

```
gcc -g media.s -o media -no-pie -nostartfiles
```

Ahora lanzamos con gdb:

```
gdb media
```

Y ya estamos en la interfaz de gdb. Primero colocamos un breakpoint donde nos interese, en mi caso en la línea 28, que es la que coincide con la subrutina bucle. Para ello escribimos break 28 y seguimos ejecutando línea por línea. Al final vemos que todo va como queremos hasta llamar a la función exit. Para evitar este fallo vamos a utilizar printf() de libC y para ello realizamos los siguientes cambios en media.s:

```
.section .text
main: .global main

    call trabajar    # subrutina de usuario
    call imprim_C    # printf() de libC
    call acabar_C    # exit() de libC
    ret
```

Y además implementamos las dos funciones imprim\_C y acabar\_C:

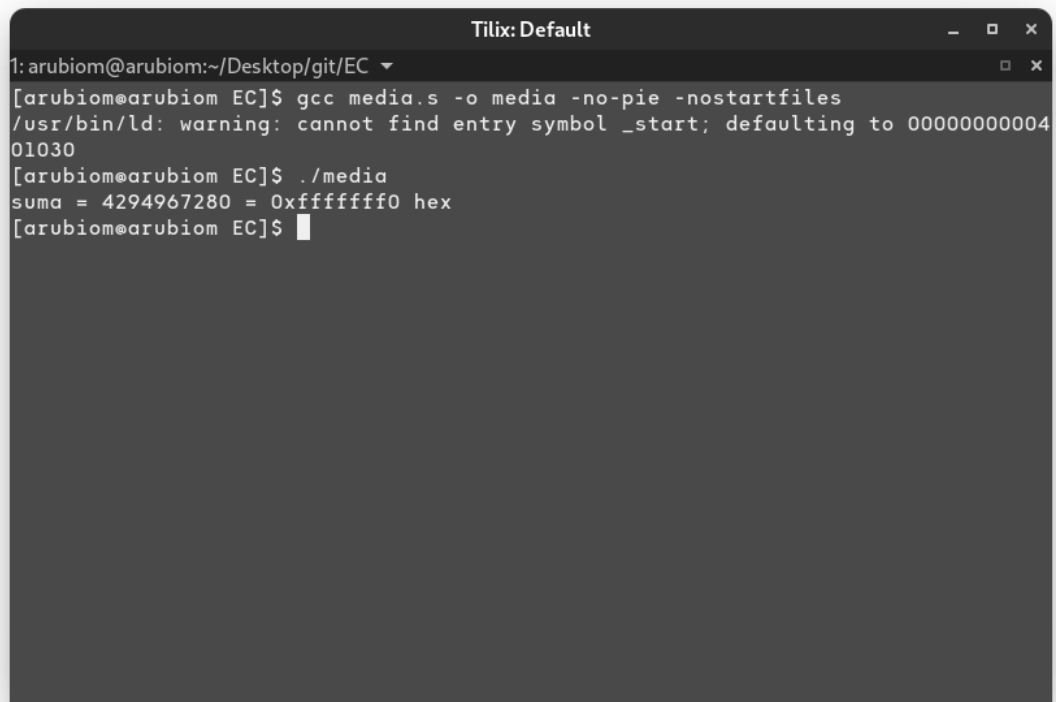
```

imprim_C:          # requiere libc
    mov    $formato, %rdi
    mov    resultado,%esi
    mov    resultado,%edx
    mov     $0,%eax    # varargin sin xmm
    call   printf      # == printf(formato, res, res);

acabar_C:          # requiere libc
    mov    resultado, %edi
    call   exit        # ==  exit(resultado)

```

Con esto al ejecutar obtenemos:



```

Tilix: Default
1: arubiom@arubiom: ~/Desktop/git/EC
[arubiom@arubiom EC]$ gcc media.s -o media -no-pie -nostartfiles
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000401030
[arubiom@arubiom EC]$ ./media
suma = 4294967280 = 0xffffffff0 hex
[arubiom@arubiom EC]$

```

Que es el resultado que esperábamos obtener. Ya podemos comprobar que nuestro programa es correcto sin necesidad de usar gdb. Veamos ahora que pasa cuando sumamos 16 veces el dato 0x1000 0000:

```
Tilix: Default
1:arubiom@arubiom:~/Desktop/git/EC
[arubiom@arubiom EC]$ gcc media.s -o media -no-pie -nostartfiles
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000401030
[arubiom@arubiom EC]$ ./media
suma = 0 = 0x0 hex
[arubiom@arubiom EC]$
```

Seguimos viendo como se pierde el bit de acarreo en este caso. Para solucionarlo lo que vamos a hacer es guardar el acarreo cada vez que ocurra en otro registro y luego los concatenamos y almacenamos en un registro de 64 bits. Para ello necesitamos usar la orden JNC para saber cuando hay que incrementar el acarreo, y para ello vamos a necesitar una nueva etiqueta:

```
suma:
    mov $0, %eax
    mov $0, %edx
    mov $0, %rsi
bucle:
    add (%rbx,%rsi,4), %eax
    jnc incrementos
    inc %edx
incrementos:
    inc %rsi
    cmp %rsi,%rcx
    jne bucle

ret
```

También vamos a tener que cambiar el tipo de resultado a .quad para que ocupe 8 bytes y decirle al formato que va a tener ese tamaño. Para ello modificamos los datos:

```
.section .data
lista: .int 0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000,
0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000,
0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000
longlista: .int (.-lista)/4
resultado: .quad 0
formato: .asciz "suma = %lu = 0x%lx hex\n"
```

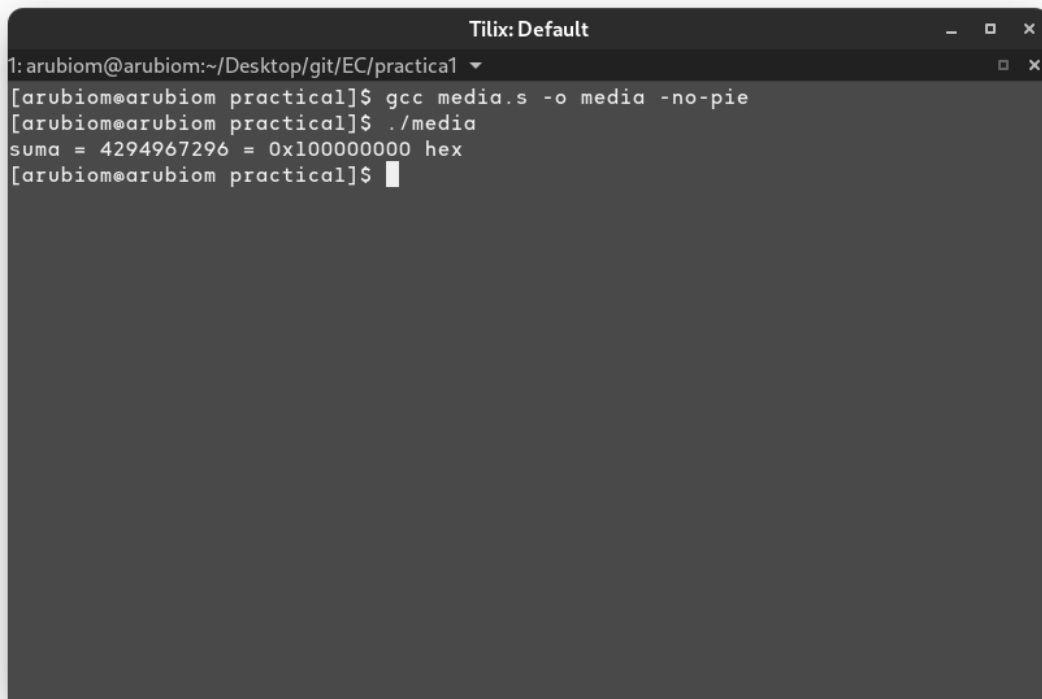
Ahora para concatenar los registros EDX:EAX tan solo hacemos movs pero moviendo el offset:

```
mov  %eax, resultado
mov  %edx, resultado+4
```

Finalmente ya para que nos muestre por pantalla el resultado en 64 bits cambiamos los registros de imprim\_C:

```
imprim_C:          # requiere libc
    mov    $formato, %rdi
    mov    resultado,%rsi
    mov    resultado,%rdx
    mov     $0,%eax    # varargin sin xmm
    call   printf      # == printf(formato, res, res);
    ret
```

Ya por último se ha podido observar como el compilador muestra un warning "cannot find entry symbol \_start" y esto se debe a que -nostartfiles no es necesario ponerlo cuando usamos gcc y definimos el main. Al final probamos el programa:



The screenshot shows a terminal window titled "Tilix: Default". The user is in a directory ~/Desktop/git/EC/practica1. They compile a file media.s into media using gcc with the -no-pie flag. Then they run ./media, which outputs "suma = 4294967296 = 0x100000000 hex".

```
Tilix: Default
1:arubiom@arubiom:~/Desktop/git/EC/practica1
[arubiom@arubiom practical]$ gcc media.s -o media -no-pie
[arubiom@arubiom practical]$ ./media
suma = 4294967296 = 0x100000000 hex
[arubiom@arubiom practical]$
```

Efectivamente obtenemos el resultado que queríamos por lo cual ya sabemos que nuestro programa suma con acarreo y sin signo. Ya solo nos falta ver que pasaría para sumar 16 veces el número más grande posible, e 0xffff ffff. Para ello cambiamos los datos:

```
.section .data
lista:      .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
longlista:  .int  (.-lista)/4
```

Ahora tan solo probamos el programa:

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practical1
[arubiom@arubiom practical]$ gcc media.s -o media -no-pie
[arubiom@arubiom practical]$ ./media
suma = 68719476720 = 0xffffffff0 hex
[arubiom@arubiom practical]$
```

Efectivamente nuestro programa hace lo esperado.

## 5.2. Sumar N enteros sin signo de 32 bits sobre dos registros de 32 bits mediante extensión con ceros (N≈16)

Ahora tenemos una solución al problema de la suma con acarreo, pero esto no significa que sea la mejor forma posible. Vamos a hacer ahora uso de la orden ADC para sumar por separado las partes más y menos significativas de los números. Para ello podemos imaginar un número sin signo de 32 bits como uno de 64 bits en el que los 32 de la izquierda son ceros, que será la parte a la que se le suma el acarreo si lo hubiera. Para esto hacemos los siguientes cambios:

```
suma:
    mov    $0, %eax
    mov    $0, %edx
    mov    $0, %rsi

bucle:
    add    (%rbx,%rsi,4), %eax
    adc    $0, %edx
    inc    %rsi
    cmp    %rsi,%rcx
    jne    bucle

    ret
```

Comprobamos ahora si funciona para todos los ejemplos realizados anteriormente:

[illegible]

```
Tilix: Default
1: arubiom@arubiom: ~/Desktop/git/EC/practica1
[arubiom@arubiom practical]$ gcc unsignedsum2.s -o unsigned -no-pie
[arubiom@arubiom practical]$ ./unsigned
suma = 16 = 0x10 hex
[arubiom@arubiom practical]$
```

Para el primer ejemplo (no tiene acarreo) funciona. Veamos el siguiente:

```
.section .data
lista:      .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
longlista:  .int  (.-lista)/4
```

```
Tilix: Default
1: arubiom@arubiom: ~/Desktop/git/EC/practica1
[arubiom@arubiom practical]$ gcc unsignedsum2.s -o unsigned -no-pie
[arubiom@arubiom practical]$ ./unsigned
suma = 4294967280 = 0xffffffff0 hex
[arubiom@arubiom practical]$
```

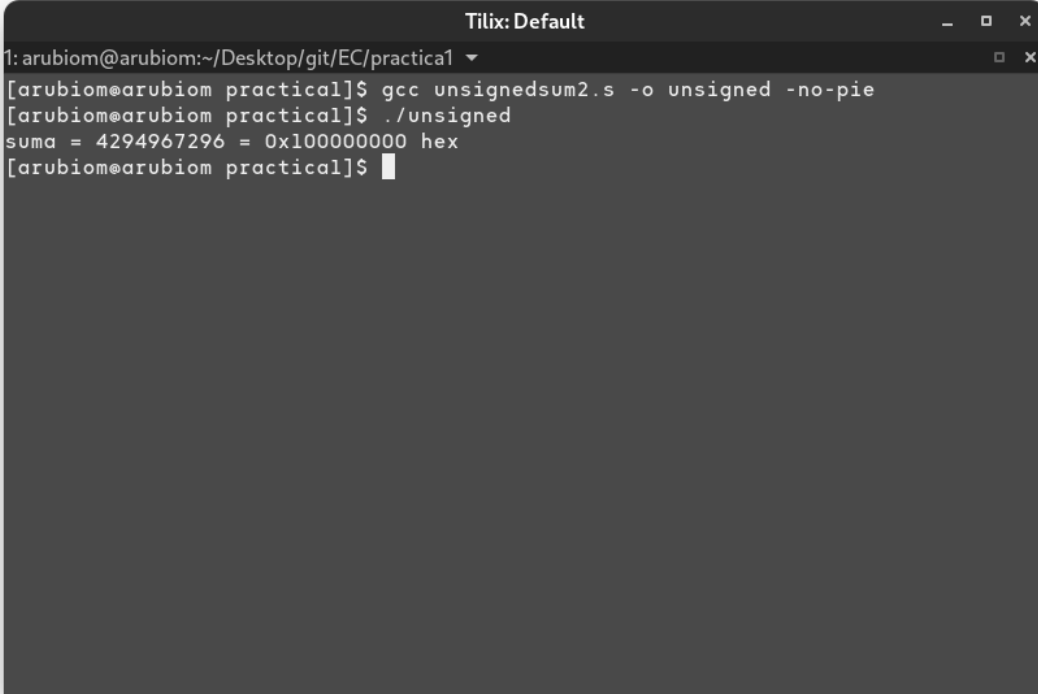
También es correcto. Vamos con el siguiente ejemplo:



```

.section .data
lista:      .int 0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000,
0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000,
0x10000000, 0x10000000, 0x10000000, 0x10000000, 0x10000000
longlista:  .int  (.-lista)/4

```



```

Tilix: Default
1:arubiom@arubiom:~/Desktop/git/EC/practical1 ▾
[arubiom@arubiom practical1]$ gcc unsignedsum2.s -o unsigned -no-pie
[arubiom@arubiom practical1]$ ./unsigned
suma = 4294967296 = 0x100000000 hex
[arubiom@arubiom practical1]$

```

Podemos ver como este método funciona también para sumas que tengan acarreo. Veamos ya por último el ejemplo del máximo número que podemos representar con 32 bits. Este es:

```

.section .data
lista:      .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
longlista:  .int  (.-lista)/4

```

```
Tilix: Default
1: arubiom@arubiom: ~/Desktop/git/EC/practica1
[arubiom@arubiom practical]$ gcc unsignedsum2.s -o unsigned -no-pie
[arubiom@arubiom practical]$ ./unsigned
suma = 68719476720 = 0xffffffff0 hex
[arubiom@arubiom practical]$
```

Efectivamente esto es correcto para todos los ejemplos.

Como se puede observar, es bastante engorroso pasarle los tests al programa uno a uno. Para ello buscamos una solución, como por ejemplo, dejar comentado los tests y solo descomentar el que queramos pasar. Algo así, haciendo uso de las órdenes `.macro` y `.irpc`:

```
.section .data
.macro linea
    .int 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
    # .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
    # .int 5000000000, 5000000000, 5000000000, 5000000000
.endm
lista: .irpc i,1234
    linea
.endr
```

En este caso estamos usando los datos `.int 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1` y el resultado no cambiará al de antes:

```
Tilix: Default
1: arubiom@arubiom: ~/Desktop/git/EC/practica1
[arubiom@arubiom practical]$ gcc unsignedsum2.s -o unsigned -no-pie
[arubiom@arubiom practical]$ ./unsigned
suma = 16 = 0x10 hex
[arubiom@arubiom practical]$
```

Pero aún así podríamos mejorar la comodidad a la hora de pasar los tests aún más usando el compilación condicional con cpp. Para esto vamos a diseñar de nuevo los datos, concretamente los valores de la lista:

```
.section .data
#ifdef TEST
#define TEST 9
#endif

.macro linea
#if TEST==1
.int 1,1,1,1
#elif TEST==2
.int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
...
#elif TEST==8
.int 5000000000, 5000000000, 5000000000, 5000000000
#else
.error "Definir TEST entre 1..8"
#endif
.endm
```

Donde se entiende que "..." son otros tests distintos. También aprovechamos para cambiar el formato del printf:

```
formato:
.ascii "resultado \t =  %18lu (uns)\n"
.ascii  "\t\t = 0x%18lx (hex)\n"
.asciz "\t\t = 0x %08x %08x\n"
```

Así obtendremos más información de cada tests. En concreto los tests que vamos a pasar son los siguientes:

```

#if TEST==1
    .int 1,1,1,1
#elif TEST==2
    .int 0x0fffffff, 0x0fffffff, 0x0fffffff, 0x0fffffff
#elif TEST==3
    .int 0x10000000, 0x10000000, 0x10000000, 0x10000000
#elif TEST==4
    .int 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
#elif TEST==5
    .int -1, -1, -1, -1
#elif TEST==6
    .int 200000000, 200000000, 200000000, 200000000
#elif TEST==7
    .int 300000000, 300000000, 300000000, 300000000
#elif TEST==8
    .int 500000000, 500000000, 500000000, 500000000

```

Pero para ejecutarlos paso a paso necesitamos enviarlos uno a uno al compilador. Para ello utilizamos el siguiente script de bash:

```

for i in $(seq 1 9); do
    rm unsigned
    gcc -x assembler-with-cpp -D TEST=$i -no-pie unsignedsum2.s -o unsigned
    printf "__TEST%02d__%35s\n" $i "" | tr " " "-"; ./unsigned
done

```

Ahora solo lo ejecutamos y vemos los resultados:

```

Tilix: Default
1: arubiom@arubiom: ~/Desktop/git/EC/practica1
__TEST01__-----
resultado      =          16 (uns)
               = 0x          10 (hex)
               = 0x 00000010 00000000
__TEST02__-----
resultado      =    4294967280 (uns)
               = 0x          ffffffff (hex)
               = 0x 00000010 00000000
__TEST03__-----
resultado      =    4294967296 (uns)
               = 0x          100000000 (hex)
               = 0x 00000010 00000000
__TEST04__-----
resultado      =    68719476720 (uns)
               = 0x          ffffffff0 (hex)
               = 0x 00000010 00000000
__TEST05__-----
resultado      =    68719476720 (uns)
               = 0x          ffffffff0 (hex)
               = 0x 00000010 00000000
__TEST06__-----
resultado      =   32000000000 (uns)
               = 0x          bebc2000 (hex)
               = 0x 00000010 00000000

```

Vemos que los tests del 1 al 6 no hay problema, a excepción del 5, que como era esperado no es el resultado correcto, pues nuestro programa no trabaja con signos.

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
__TEST02__-----
resultado      =          4294967280 (uns)
               = 0x          ffffffff (hex)
               = 0x 00000010 00000000
__TEST03__-----
resultado      =          4294967296 (uns)
               = 0x          100000000 (hex)
               = 0x 00000010 00000000
__TEST04__-----
resultado      =          68719476720 (uns)
               = 0x          ffffffff0 (hex)
               = 0x 00000010 00000000
__TEST05__-----
resultado      =          68719476720 (uns)
               = 0x          ffffffff0 (hex)
               = 0x 00000010 00000000
__TEST06__-----
resultado      =          3200000000 (uns)
               = 0x          bebc2000 (hex)
               = 0x 00000010 00000000
__TEST07__-----
resultado      =          4800000000 (uns)
               = 0x          11e1a3000 (hex)
               = 0x 00000010 00000000
```

El test 7 también pasa correctamente. Veamos sin embargo que pasa ahora con el 8:

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
      = 0x 00000010 00000000
unsignedsum2.s: Assembler messages:
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
__TEST08__-----
resultado      =          11280523264 (uns)
               = 0x          2a05f2000 (hex)
               = 0x 00000010 00000000
unsignedsum2.s: Assembler messages:
unsignedsum2.s:28: Error: Definir TEST entre 1..8
```

Para empezar el test 8 nos dice que está truncando los números, pues habíamos puesto un número que no cabía en 32 bits. Luego el resultado está calculado para este resultado.

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
unsignedsum2.s:28: Warning: value 0x12a05f200 truncated to 0x2a05f200
__TEST08__-----
resultado      =          11280523264 (uns)
               = 0x          2a05f2000 (hex)
               = 0x 00000010 00000000
unsignedsum2.s: Assembler messages:
unsignedsum2.s:28: Error: Definir TEST entre 1..8
unsignedsum2.s:28: Error: Definir TEST entre 1..8
unsignedsum2.s:28: Error: Definir TEST entre 1..8
unsignedsum2.s:28: Error: Definir TEST entre 1..8
__TEST09__-----
./passtest.sh: line 6: ./unsigned: No such file or directory
[arubiom@arubiom practica1]$
```

Vemos como el test 9 no existe y así se indica. Con esto nos quedaría ya listo el programa para sumar dos enteros de 32 bits sin signo. Ahora veamos lo que tenemos que hacer para sumar dos enteros también de 32 bits pero con signo.

### 5.3. Sumar N enteros con signo de 32 bits sobre dos registros de 32 bits (mediante extensión de signo, naturalmente) (N≈16)

Ahora nuestro problema radica en que nuestro programa no tiene en cuenta el signo del elemento de la lista. Para evitarlo lo que buscamos hacer es primero extender el signo del número en cuestión y luego sumarlo. Para ello antes de sumarlo usamos la orden `cld` en nuestro caso que es la que nos sirve para los registros `EDX:EAX`. También vamos a necesitar nuevos registros para acumular, y para ello usamos `EDI` y `EBP`, aunque luego el resultado tiene que seguir quedando en `EDX:EAX`:

```
suma:
    mov $0, %edi
    mov $0, %ebp
    mov $0, %rsi
bucle:
    mov (%rbx,%rsi,4), %eax
    cld
    add %eax, %edi
    adc %edx, %ebp
    inc %rsi
    cmp %rsi,%rcx
    jne bucle

    mov %edi, %eax
    mov %ebp, %edx

    ret
```

Ahora solo tenemos que especificar en el formato que vamos a trabajar con signos, esto es:

```
.ascii "resultado \t = %18ld (sgn)\n"
```

Y ya tan solo le pasamos los tests para ver si funciona correctamente. Los tests que usamos esta vez son:

```
#if TEST==1
    .int -1 , -1 , -1, -1
#elif TEST==2
    .int 0x04000000, 0x04000000, 0x04000000, 0x04000000
#elif TEST==3
    .int 0x08000000, 0x08000000, 0x08000000, 0x08000000
#elif TEST==4
    .int 0x10000000, 0x10000000, 0x10000000, 0x10000000
#elif TEST==5
    .int 0x7fffffff, 0x7fffffff, 0x7fffffff, 0x7fffffff
#elif TEST==6
    .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
#elif TEST==7
    .int 0xF0000000, 0xF0000000, 0xF0000000, 0xF0000000
#elif TEST==8
    .int 0xF8000000, 0xF8000000, 0xF8000000, 0xF8000000
#elif TEST==9
    .int 0xF7FFFFFF, 0xF7FFFFFF, 0xF7FFFFFF, 0xF7FFFFFF
#elif TEST==10
    .int 100000000, 100000000, 100000000, 100000000
#elif TEST==11
    .int 200000000, 200000000, 200000000, 200000000
#elif TEST==12
    .int 300000000, 300000000, 300000000, 300000000
#elif TEST==13
    .int 2000000000, 2000000000, 2000000000, 2000000000
#elif TEST==14
    .int 3000000000, 3000000000, 3000000000, 3000000000
#elif TEST==15
    .int -100000000, -100000000, -100000000, -100000000
#elif TEST==16
    .int -200000000, -200000000, -200000000, -200000000
#elif TEST==17
    .int -300000000, -200000000, -200000000, -200000000
#elif TEST==18
    .int -2000000000, -2000000000, -2000000000, -2000000000
#elif TEST==19
    .int -3000000000, -3000000000, -3000000000, -3000000000
```

Realizando pequeñas modificaciones en el script de ejecutar los tests:

```
for i in $(seq 1 19); do
    rm signed;
    gcc -x assembler-with-cpp -D TEST=$i signedsum.s -no-pie -o signed;
    printf "__TEST%02d__%35s\n" $i "" | tr " " "-"; ./signed;
done
```

Ejecutamos e interpretemos los resultados:

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
__TEST01__-----
resultado      =          -16 (sgn)
               = 0x  ffffffff00000000 (hex)
               = 0x  00000010 00000000
__TEST02__-----
resultado      =       1073741824 (sgn)
               = 0x      40000000 (hex)
               = 0x  00000010 00000000
__TEST03__-----
resultado      =       2147483648 (sgn)
               = 0x      80000000 (hex)
               = 0x  00000010 00000000
__TEST04__-----
resultado      =       4294967296 (sgn)
               = 0x     100000000 (hex)
               = 0x  00000010 00000000
__TEST05__-----
resultado      =       3435973832 (sgn)
               = 0x      7fffffff (hex)
               = 0x  00000010 00000000
__TEST06__-----
resultado      =      -34359738368 (sgn)
               = 0x  ffffffff80000000 (hex)
               = 0x  00000010 00000000
```

Vemos como los tests del 1 al 6 el resultado es correcto.

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
__TEST07__-----
resultado      =      -4294967296 (sgn)
               = 0x  ffffffff00000000 (hex)
               = 0x  00000010 00000000
__TEST08__-----
resultado      =      -2147483648 (sgn)
               = 0x  ffffffff80000000 (hex)
               = 0x  00000010 00000000
__TEST09__-----
resultado      =      -2147483664 (sgn)
               = 0x  ffffffff7fffffff (hex)
               = 0x  00000010 00000000
__TEST10__-----
resultado      =       1600000000 (sgn)
               = 0x      5f5e1000 (hex)
               = 0x  00000010 00000000
__TEST11__-----
resultado      =       3200000000 (sgn)
               = 0x      bebc2000 (hex)
               = 0x  00000010 00000000
__TEST12__-----
resultado      =       4800000000 (sgn)
               = 0x      11e1a3000 (hex)
               = 0x  00000010 00000000
```

De igual manera los tests 7 a 12 están bien.





#### 5.4. Media y resto de N enteros con signo de 32 bits calculada usando registros de 32 bits (N≈16)

Ahora lo que buscamos no es obtener el resultado de la suma de la lista, sino su media y su resto al dividir por la longitud. Para ello usamos la orden IDIV, que divide el contenido de los registros EDX:EAX entre el número que elijamos, y almacena el cociente en EAX y el resto en EDX. Así tenemos un dividendo de 64 bits y un resto y cociente de 32 bits.

Dejamos de la siguiente forma la subrutina suma:

```
suma:
    mov $0, %edi
    mov $0, %ebp
    mov $0, %rsi
bucle:
    mov (%rbx,%rsi,4), %eax
    cld
    add %eax, %edi
    adc %edx, %ebp
    inc %rsi
    cmp %rsi,%rcx
    jne bucle

    mov %edi, %eax
    mov %ebp, %edx

    idiv %ecx

    ret
```

Y claro está ahora hay que tener cuidado en donde guardamos los resultados entonces declaramos dos variables nuevas resto y media:

```
media: .int 0
resto: .int 0
```

Y cambiamos el formato para que salga por pantalla como queremos:

```
formato: .asciz "Media = %d = 0x%x hex\n Resto = %d = 0x%x hex\n"
```

Además tendremos que cambiar el imprimir para que los registros que se muestren por pantalla sean los adecuados:

```
imprim_C:          # requiere libC
    mov $formato, %rdi # argumentos
    mov media, %esi # argumentos
    mov media, %edx # argumentos
    mov resto, %ecx
    mov resto, %r8
    mov $0, %eax
    call printf
    ret

acabar_C:          # requiere libC
    mov media, %edi
```

```
call exit      # == exit(resultado)
ret
```

Así que así ya tenemos el programa funcional que calcula la media y el resto. Ahora comprobemos que funciona pasándole los siguientes tests:

```
#if TEST==1
    .int 1,2,1,2
#elif TEST==2
    .int -1,-2,-1,-2
#elif TEST==3
    .int 0x7fffffff,0x7fffffff,0x7fffffff,0x7fffffff
#elif TEST==4
    .int 0x80000000,0x80000000,0x80000000,0x80000000
#elif TEST==5
    .int 0xffffffff,0xffffffff,0xffffffff,0xffffffff
#elif TEST==6
    .int 2000000000,2000000000,2000000000,2000000000
#elif TEST==7
    .int 3000000000,3000000000,3000000000,3000000000
#elif TEST==8
    .int -2000000000,-2000000000,-2000000000,-2000000000
#elif TEST==9
    .int -3000000000,-3000000000,-3000000000,-3000000000
#elif TEST>=10 && TEST <=14
    .int 1,1,1,1
#elif TEST >= 15 && TEST<=19
    .int -1,-1,-1,-1
#else
    .error "Definir test"
#endif
.endm

.macro linea0
#if TEST>=1 && TEST<=9
    linea
#elif TEST==10
    .int 0,2,1,1
#elif TEST==11
    .int 1,2,1,1
#elif TEST==12
    .int 8,2,1,1
#elif TEST==13
    .int 15,2,1,1
#elif TEST==14
    .int 16,2,1,1
#elif TEST==15
    .int 0,-2,-1,-1
#elif TEST==16
    .int -1,-2,-1,-1
#elif TEST==17
    .int -8,-2,-1,-1
#elif TEST==18
    .int -15,-2,-1,-1
#elif TEST==19
    .int -16,-2,-1,-1
```

```

#else
    .error "Definir test"
#endif
.endm

lista:    linea0
    .irpc i,123
        linea
    .endr

```

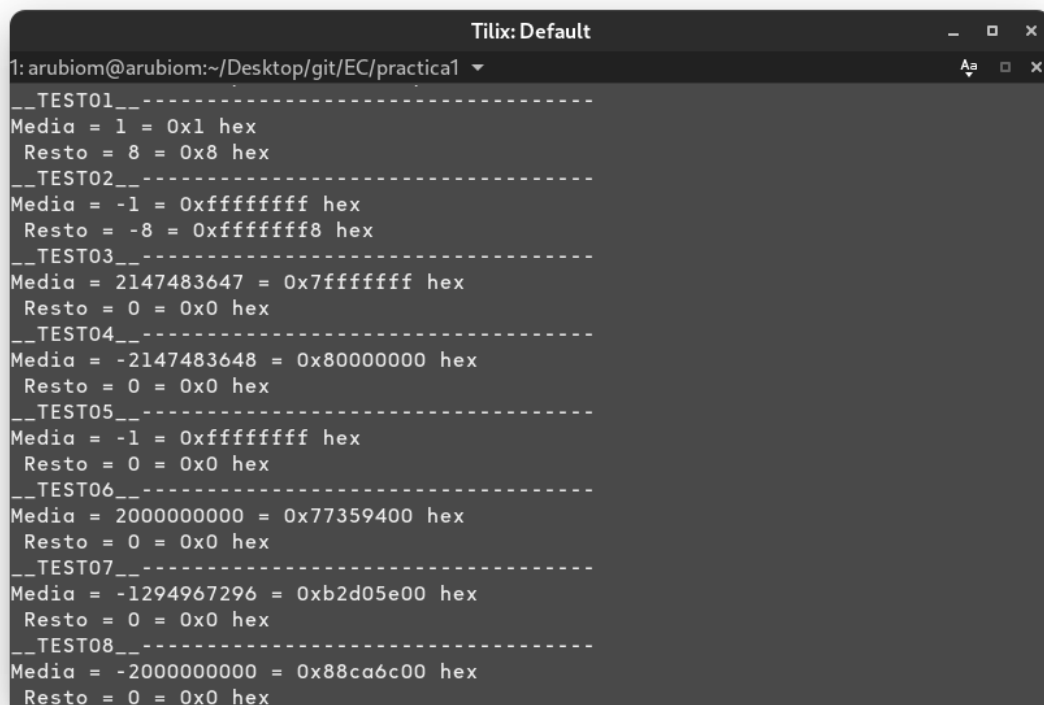
Utilizando dos macros en este caso para definir los tests de la forma más cómoda posible. Vamos a ejecutar el programa con la script modificada siguiente:

```

for i in $(seq 1 19); do
    rm media;
    gcc -x assembler-with-cpp -D TEST=$i media.s -no-pie -o media;
    printf "__TEST%02d__%35s\n" $i "" | tr " " "-"; ./media;
done

```

Veamos los resultados ahora de los tests:



```

Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
__TEST01__-----
Media = 1 = 0x1 hex
Resto = 8 = 0x8 hex
__TEST02__-----
Media = -1 = 0xffffffff hex
Resto = -8 = 0xffffffff8 hex
__TEST03__-----
Media = 2147483647 = 0x7fffffff hex
Resto = 0 = 0x0 hex
__TEST04__-----
Media = -2147483648 = 0x80000000 hex
Resto = 0 = 0x0 hex
__TEST05__-----
Media = -1 = 0xffffffff hex
Resto = 0 = 0x0 hex
__TEST06__-----
Media = 2000000000 = 0x77359400 hex
Resto = 0 = 0x0 hex
__TEST07__-----
Media = -1294967296 = 0xb2d05e00 hex
Resto = 0 = 0x0 hex
__TEST08__-----
Media = -2000000000 = 0x88ca6c00 hex
Resto = 0 = 0x0 hex

```

Aquí vemos como todos los resultados de los tests son lo que esperábamos menos el test 7, esto debido a que ha habido desbordamiento al sumar, puesto que los valores introducidos son muy grandes. A ver ahora si los siguientes tests también son correctos:

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
Resto = 0 = 0x0 hex
media.s: Assembler messages:
media.s:64: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:64: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:64: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:64: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
media.s:67: Warning: value 0xffffffff4d2fa200 truncated to 0x4d2fa200
__TEST9__-----
Media = 1294967296 = 0x4d2fa200 hex
Resto = 0 = 0x0 hex
__TEST10__-----
Media = 1 = 0x1 hex
Resto = 0 = 0x0 hex
```

Ahora vemos para empezar que los valores introducidos en el test 9 son demasiado grandes para guardarlos en 32 bits así que el programa los trunca a enteros positivos a pesar de ser negativos, lo cual es el motivo por el que obtenemos esa media. Veamos lo siguiente tests que en principio no debería haber ningún problema:

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
__TEST11__-----
Media = 1 = 0x1 hex
Resto = 1 = 0x1 hex
__TEST12__-----
Media = 1 = 0x1 hex
Resto = 8 = 0x8 hex
__TEST13__-----
Media = 1 = 0x1 hex
Resto = 15 = 0xf hex
__TEST14__-----
Media = 2 = 0x2 hex
Resto = 0 = 0x0 hex
__TEST15__-----
Media = -1 = 0xffffffff hex
Resto = 0 = 0x0 hex
__TEST16__-----
Media = -1 = 0xffffffff hex
Resto = -1 = 0xffffffff hex
__TEST17__-----
Media = -1 = 0xffffffff hex
Resto = -8 = 0xffffffff8 hex
__TEST18__-----
Media = -1 = 0xffffffff hex
Resto = -15 = 0xffffffffl hex
```

```
Tilix: Default
1: arubiom@arubiom:~/Desktop/git/EC/practica1
__TEST19__-----
Media = -2 = 0xfffffffffe hex
Resto = 0 = 0x0 hex
[arubiom@arubiom practical]$ ^C
```

Vemos como el resto de tests son correctos por lo que podemos asumir que nuestro programa es correcto.

Ahora vamos a discutir el signo del módulo (resto). Si buscamos en wikipedia la definición de división truncada encontramos lo siguiente:

"Many implementations use *truncated division*, where the quotient is defined by truncation  $q = \text{trunc}(a/n)$  and thus according to equation the remainder would have *same sign as the dividend*. The quotient is rounded towards zero: equal to the first integer in the direction of zero from the exact rational quotient.

$$r = a - n \cdot \text{trunc}(a/n)$$

De donde podemos extraer básicamente que el signo del resto coincidirá con el del dividendo. Esto es, EDX conservará los bits más significativos si son 0 ó f.