Martina Vitagliano i

NIU:1667937 - 1671634

DE C A ASSEMBLY

Link GitHub:
La pràctica consisteix en realitzar una sèrie de programes en C, compilar-los i obrir-los amb l'ensamblador ddd. L'objectiu és respondre unes preguntes, a partir del codi assembly que surt.

Per resoldre aquesta pràctica hem obert un editor amb la comanda nano i hem escrit el nostre programa en llenguatge de programació C. Després l'hem compilat amb l'instrucció "gcc -c -o [el nom del programa compilat] [el nom del programa en C]". Un cop compilat i comprovat que el nostre programa no dona cap tipus d'error, hem executat la comanda "ddd ./ [el nom del programa compilat]" per obrir el debugger i a partir d'aquest pas podrem observar el codi assembly del nostre programa.

Exercici 1

a)¿En cuántas instrucciones de lenguaje Assembly se traduce el programa escrito en C?

Tot i que hi ha 11 línies, la primera de totes, *endbr64* no es considera una instrucció ja que significa els bit que utilitzarem per treballar, en aquest cas 64 bits. Per tant, el nostre programa té **10 instruccions** en llenguatge assembly.

```
#include <stdio.h>

int main(){

        int i,j;

        i = 1;
        j = 2;
        i++;
        j += i;

        return 0;
}
```

```
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0x1,-0x8(%rbp)
   0x0000555555555138 <+15>:    movl   $0x2,-0x4(%rbp)
   0x000055555555513f <+22>:    addl   $0x1,-0x8(%rbp)
   0x0000555555555143 <+26>:    mov    -0x8(%rbp),%eax
   0x0000555555555146 <+29>:    add    %eax,-0x4(%rbp)
=> 0x0000555555555149 <+32>:    mov    $0x0,%eax
   0x000055555555514e <+37>:    pop    %rbp
   0x000055555555514f <+38>:    retq
End of assembler dump.
```

b)¿En qué posición de memoria comienza a almacenarse el programa escrito en C?

En aquest cas les nostres instruccions comencen a emmagatzemar-se a la posició de memoria **0x0000555555555129** degut a que és la nostra primera instrucció del programa.

c)La asignación del valor 1 a la variable entera i, ¿cómo se traduce a código Assembly?, ¿y la asignación del valor 2 a la variable entera j? Indica la instrucción (qué hace), el operando fuente y el operando destino.

L'instrucció d'assignació d'1 a la variable i es: **0x0000555555555131 <+8>: movl $0x1,-0x8(%rbp)**
El $0x1 és l'operand font i igual a 1, ho movem a l'operand destí (-0x8(%rbp)), això el que fa és la direcció base de %rbp + el desplaçament de -8

L'instrucció d'assignació d'1 a la variable i es: **0x0000555555555138 <+15>: movl $0x2,-0x4(%rbp)**
El $0x2 és l'operand font i igual a 2, ho movem a l'operand destí (-0x4(%rbp)), això el que fa és la direcció base de %rbp + el desplaçament de -4

```
0x0000555555555131 <+8>:     movl   $0x1,-0x8(%rbp)
0x0000555555555138 <+15>:    movl   $0x2,-0x4(%rbp)
```

d)¿Qué secuencia de instrucciones es empleada para incrementar en 1 el valor de la variable i? Para cada instrucción indica la instrucción (qué hace), el operando fuente y el operando destino.

0x000055555555513f <+22>: addl $0x1,-0x8(%rbp)

Aquí el que està fent és la suma d'un enter $0x1[1] = 1 a la variable *i*, que es troba en %rbp - 8. És a dir, 8 posicions menys que la direcció de memòria on es troba el registre rbp.

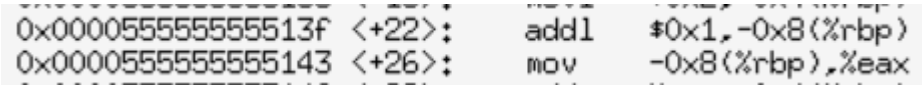
- Operand font: $0x1

- Operand destí: -0x8(%rbp)

El resultat d'aquesta operació el guarda en un registre, en aquest cas: %eax. Per fer aquest pas l'assembler realitza el pas **0x0000555555555143 <+26>: mov -0x8(%rbp),%eax.**
En aquesta instrucció,

- **Operand fon:** -0x8(%rbp)

- **Operand destí:** %eax



e)¿Qué registro, de los que aparecen en la ventana de *Registros*, es empleado de forma auxiliar para realizar las operaciones de suma? ¿Qué secuencia de valores toma el registro a lo largo de la ejecución del programa?

El registre que s'utilitza com a auxiliar per fer la suma és el **%eax (4 Bytes)**, els valors d'aquest registre van variant al llarg del programa. El registre %eax apareix quan es desa el resultat de la suma i++. Afegeix el valor de *i*, guardat al registre, a la variable *j* que es troba a la posició %rbp -4. Per acabar, guarda 0 al registre %eax per fer el return del programa.

f)¿Cuántos bytes ocupan las variables enteras *i* y *j* en su representación interna?

Com les nostres variables son dos enteres *int* i la instrucció *mov* porta una *l* al final (movl), això en està indicant que les variables *i*, *j* equivalen a **32 bits.**

g)¿Cómo podemos saber en qué posición de memoria se encuentran almacenadas las variables enteras *i* y *j*?

Per saber la posició de memoria a la que es troben emmagatzemades les nostres variables *i* i *j* hem de veure les instruccions que ha fet l'assembler al guardar-les.

Quan fa el *mov*, observem que aquest pas es realitza a les adreces de memoria:

- Adreça de memòria variable *i*: 0x0000555555555131 <+8>: movl $0x1,-0x8(%rbp)

[1] 0x* - Significa que és amb base hexadecimal.

- Adreça de memòria variable *j*: 0x0000555555555138 <+15> : movl $0x2,-0x4(%rbp)

Si ens fixem l'adreçament que està utilitzant és l'adreçament relatiu a base. És a dir, a partir de l'adreça de memòria del registre %rbp, trobem les adreces dels altres valors/variables.

Si ens fixem en aquest cas són menys 8 posicions que el rbp per la variable *i*, i menys 4 posicions que la posició del registre %rbp per la variable *j*.

Per tant, hem de fer la resta del l'adreça de memòria de %rbp menys -8 posicions per a *i*, i -4 per a la variable *j*.

Adreça de memòria %rbp: 7fffffffe60

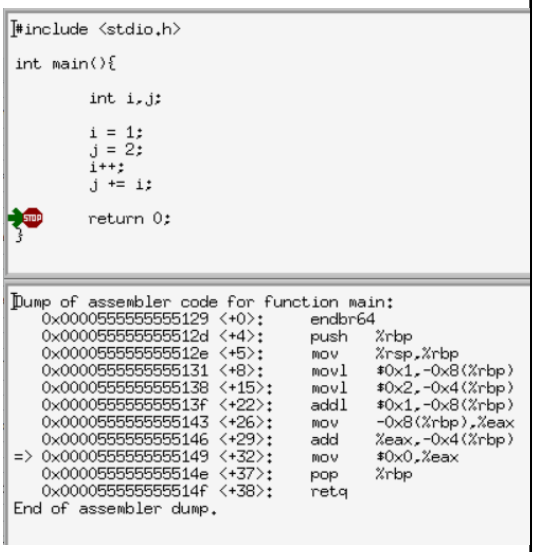
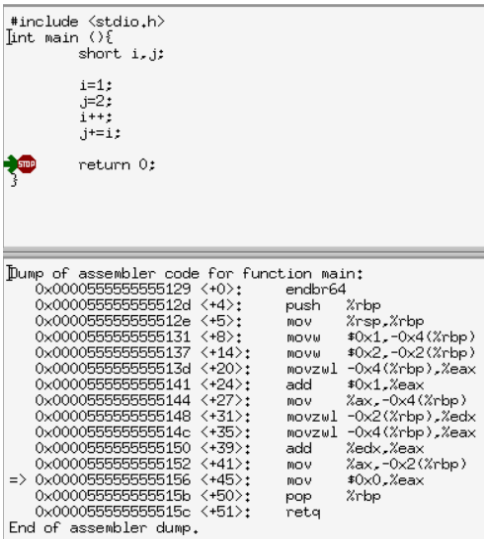
Adreça de memòria variable *i*: 7fffffffe60 - 8 = **7fffffffe58**

Adreça de memòria variable *j*: 7fffffffe60 - 8 = **7fffffffe58**

```
0x0000555555555131 <+8>:     movl   $0x1,-0x8(%rbp)
0x0000555555555138 <+15>:    movl   $0x2,-0x4(%rbp)
```

Exercici 2

a)¿Qué diferencias ves entre el c. Assembly generado para este caso y el del apartado 1?

Programa 1	Programa 2
	

```
#include <stdio.h>

int main(){

        int i,j;

        i = 1;
        j = 2;
        i++;
        j += i;

        return 0;
}
```

```
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0x1,-0x8(%rbp)
   0x0000555555555138 <+15>:    movl   $0x2,-0x4(%rbp)
   0x000055555555513f <+22>:    addl   $0x1,-0x8(%rbp)
   0x0000555555555143 <+26>:    mov    -0x8(%rbp),%eax
   0x0000555555555146 <+29>:    add    %eax,-0x4(%rbp)
=> 0x0000555555555149 <+32>:    mov    $0x0,%eax
   0x000055555555514e <+37>:    pop    %rbp
   0x000055555555514f <+38>:    retq
End of assembler dump.
```

```
#include <stdio.h>
int main (){
        short i,j;

        i=1;
        j=2;
        i++;
        j+=i;

        return 0;
}
```

```
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movw   $0x1,-0x4(%rbp)
   0x0000555555555137 <+14>:    movw   $0x2,-0x2(%rbp)
   0x000055555555513d <+20>:    movzwl -0x4(%rbp),%eax
   0x0000555555555141 <+24>:    add    $0x1,%eax
   0x0000555555555144 <+27>:    mov    %ax,-0x4(%rbp)
   0x0000555555555148 <+31>:    movzwl -0x2(%rbp),%edx
   0x000055555555514c <+35>:    movzwl -0x4(%rbp),%eax
   0x0000555555555150 <+39>:    add    %edx,%eax
   0x0000555555555152 <+41>:    mov    %ax,-0x2(%rbp)
=> 0x0000555555555156 <+45>:    mov    $0x0,%eax
   0x000055555555515b <+50>:    pop    %rbp
   0x000055555555515c <+51>:    retq
End of assembler dump.
```

Les principals diferències entre el codi assembly del segon programa i el del primer són:

Per començar, en el segon programa veiem que les variables *i*, i *j* estan declarades com a *short*, i es generen més instruccions.

Una variable short és de 16 bits, en canvi, una variable *int* és de 32 bits. Per tant, el canvi es troba en el nombre de bits i el tipus d'instruccions que fa.

Les primeres quatre instruccions són les mateixes amb posicions diferents (en programa 2: i=pos 4, j=pos 2 en programa 1: i=pos 8, j=pos 4), el que fa es assignar un valor a les variables *i*, *j*. A partir d'aquests passos comencen a aparèixer els canvis.

En el programa 1 passa directament a fer la suma de la variable *i*. En canvi, en el segon programa fa un *mov* del que hi ha a la posició 4 (la variable i=1) a un registre %eax, i un cop fet aquest pas, fa la suma a la *i*.

La diferència també es troba en el programa 2, guarda el resultat a l'adreçament -0x4(%rbp) i en el programa 1 el desa en un registre %eax.

Una altre és que al programa 1 fa directament la suma d'*i* i *j* amb un add amb un registre i una posició. Al programa 2 guarda el que i ha a les posicions -0x2(%rbp) en el registre %edx i, -0x4(%rbp)

en el registre %eax, i després fa la suma *add* amb aquests.

A partir d'això els dos programes fan el mateix, guarden els resultats a diferents posicions.

b)¿Cuántos bytes ocupan las variables tipo *short* en su representación interna?

Les variables *short* ocupen un espai de 16 bits, és a dir, **2 bytes**

c) ¿En qué se diferencian los registros %eax y %ax?

El registre **%eax** ocupa un espai de **32 bits** i el de **%ax** ocupa un de **16 bits** que són els primers bits del registre %eax.

d) ¿Qué hace la instrucción *movzwl*?

Aquesta instrucció el que realitza **és una còpia dels bits del registre %ax a un altre registre, %edx, omplint els 16 que falten a %edx amb zeros.** El mateix fa per el registre %eax.

Exercici 3

a)¿El código Assembly generado para cada expresión es el mismo? ¿Qué diferencias existen?

INT	SHORT
Optimitza moviments	Utilitza registres
Primer fa la suma, guarda resultat en un registre	Per operar primer guardar els valors en un registre
32 bits	16 bits

```
#include <stdio.h>

int main(){
        int a, b, c, d;
        a=1;
        b=2;
        c=3;
        d=4;
        a=b+c*d;
        a=(b+c)*d;
        d=a;
}
```

```
Dump of assembler code for function main:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0x1,-0x10(%rbp)
   0x0000555555555138 <+15>:    movl   $0x2,-0xc(%rbp)
   0x000055555555513f <+22>:    movl   $0x3,-0x8(%rbp)
   0x0000555555555146 <+29>:    movl   $0x4,-0x4(%rbp)
   0x000055555555514d <+36>:    mov    -0x8(%rbp),%eax
   0x0000555555555150 <+39>:    imul   -0x4(%rbp),%eax
   0x0000555555555154 <+43>:    mov    %eax,%edx
   0x0000555555555156 <+45>:    mov    -0xc(%rbp),%eax
   0x0000555555555159 <+48>:    add    %edx,%eax
   0x000055555555515b <+50>:    mov    %eax,-0x10(%rbp)
   0x000055555555515e <+53>:    mov    -0xc(%rbp),%edx
   0x0000555555555161 <+56>:    mov    -0x8(%rbp),%eax
   0x0000555555555164 <+59>:    add    %eax,%edx
   0x0000555555555166 <+61>:    mov    -0x4(%rbp),%eax
   0x0000555555555169 <+64>:    imul   %edx,%eax
   0x000055555555516c <+67>:    mov    %eax,-0x10(%rbp)
=> 0x000055555555516f <+70>:    mov    -0x10(%rbp),%eax
   0x0000555555555172 <+73>:    mov    %eax,-0x4(%rbp)
   0x0000555555555175 <+76>:    mov    $0x0,%eax
   0x000055555555517a <+81>:    pop    %rbp
   0x000055555555517b <+82>:    retq
End of assembler dump.
```

El programa de l'apartat tres d'aquesta pràctica es pot veure a la dreta, en aquest cas l'enunciat ens demana saber si les dues operacions aritmètiques que s'han fet tenen les mateixes instruccions. Per resoldre aquest apartat hem analitzat les instruccions de l'assembly. La resposta és que **no** son les mateixes, perquè per l'operació sense parèntesis el que fa primer és assignar els valors a les seves respectives posicions i després amb la instrucció imul multiplica la c i la d, desa el resultat a un registre i més tard calcula la suma d'aquest amb la b.

```
0x0000555555555131 <+8>:     movl   $0x1,-0x10(%rbp
0x0000555555555138 <+15>:    movl   $0x2,-0xc(%rbp)
0x000055555555513f <+22>:    movl   $0x3,-0x8(%rbp)
0x0000555555555146 <+29>:    movl   $0x4,-0x4(%rbp)
0x000055555555514d <+36>:    mov    -0x8(%rbp),%eax
0x0000555555555150 <+39>:    imul   -0x4(%rbp),%eax
```

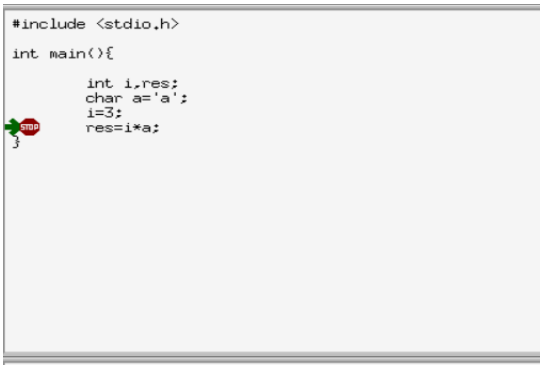
En canvi, per l'operació amb parèntesis, primer mou la b i c, a dos registres (%edx i %eax). Un cop fet aquest pas realitza un add (suma) del que hi ha als registres, i per acabar calcula el producte del resultat de la suma amb la d i ho torna a desar a la posició 10 i d'aquest a un registre (%eax).

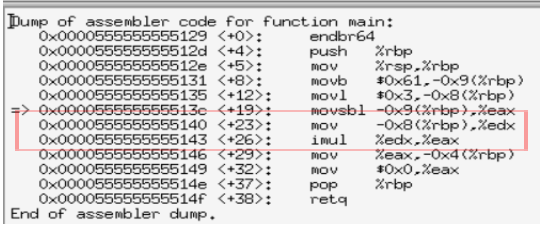
```
0x0000555555555154 <+43>:    mov    %eax,%edx
0x0000555555555156 <+45>:    mov    -0xc(%rbp),%eax
0x0000555555555159 <+48>:    add    %edx,%eax
0x000055555555515b <+50>:    mov    %eax,-0x10(%rbp)
0x000055555555515e <+53>:    mov    -0xc(%rbp),%edx
0x0000555555555161 <+56>:    mov    -0x8(%rbp),%eax
0x0000555555555164 <+59>:    add    %eax,%edx
0x0000555555555166 <+61>:    mov    -0x4(%rbp),%eax
0x0000555555555169 <+64>:    imul   %edx,%eax
0x000055555555516c <+67>:    mov    %eax,-0x10(%rbp)
0x000055555555516f <+70>:    mov    -0x10(%rbp),%eax
0x0000555555555172 <+73>:    mov    %eax,-0x4(%rbp)
```

Martina Vitagliano i

Exercici 4

a)¿Qué instrucción se genera en el código Assembly para efectuar el producto de las dos variables?



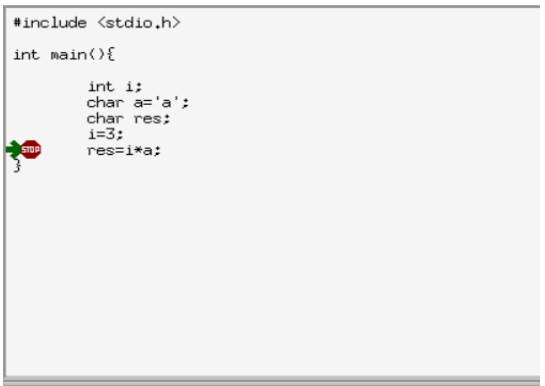


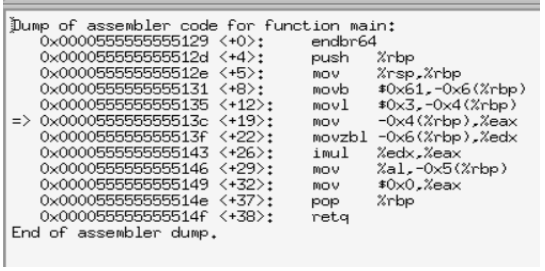
Hem observat el codi assembly del nostre programa i la instrucció que efectua la multiplicació és la de **0x0000555555555143 <+26>: imul %edx,%eax**, en aquest cas els valors a multiplicar son guardats en dos registres (%edx, %eax).

b)¿De qué tamaño es el resultado del producto?

El nostre resultat s'ha desat al registre %eax, que té un tamany de **32 bits**.

c)Cambiad el código para que el resultado se almacene en una variable tipo char. ¿El valor del producto obtenido es el mismo que en el caso anterior? Explicad los cambios que observes en el código Assembly y el motivo de dichos cambios.





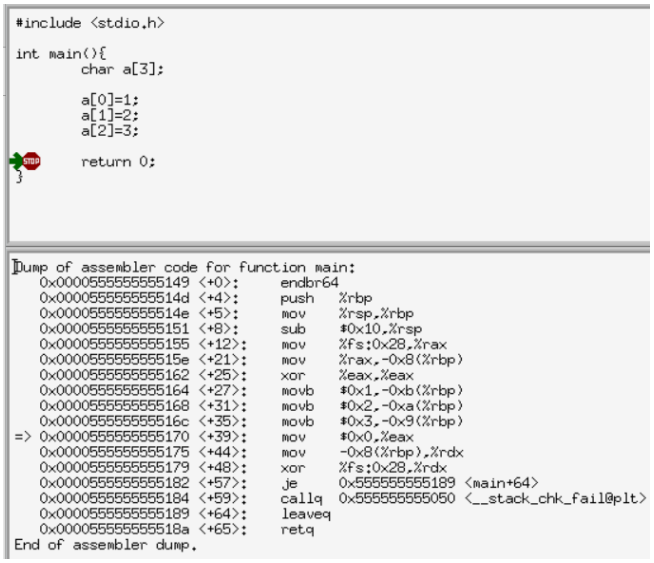
No, ens apareix el que entenem que són els últims 8 bits d'eax. La variable a en codi ascii és 97. El que fa aquest programa és multiplicar el valor de i per 97 (97*3=291). En binari aquest nombre és: 100100011, llavors com és un char que és un byte, és a dir, són 8 bits, com que el resultat es guarda en una variable declarada char, el resultat ha de contenir el seu tamany i per aquesta raó agafem del registre resultant %eax, només els últims 8 bits, amb el registre %al ja que agafa els últims 8 bits.

Per tant, només agafa 00100011 que traduït és 35. Concloure que no guarda el valor correcte.

Martina Vitagliano i

Exercici 5

a)Indique las direcciones que ocupan ahora cada elemento del array.



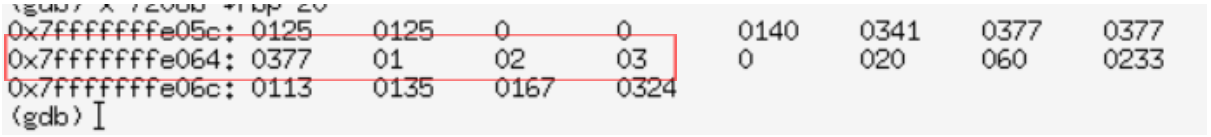
```
#include <stdio.h>

int main(){
        char a[3];

        a[0]=1;
        a[1]=2;
        a[2]=3;

        return 0;
}
```

```
Dump of assembler code for function main:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x10,%rsp
   0x0000555555555155 <+12>:    mov    %fs:0x28,%rax
   0x000055555555515e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555162 <+25>:    xor    %eax,%eax
   0x0000555555555164 <+27>:    movb   $0x1,-0xb(%rbp)
   0x0000555555555168 <+31>:    movb   $0x2,-0xa(%rbp)
   0x000055555555516c <+35>:    movb   $0x3,-0x9(%rbp)
=> 0x0000555555555170 <+39>:    mov    $0x0,%eax
   0x0000555555555175 <+44>:    mov    -0x8(%rbp),%rdx
   0x0000555555555179 <+48>:    xor    %fs:0x28,%rdx
   0x0000555555555182 <+57>:    je     0x555555555189 <main+64>
   0x0000555555555184 <+59>:    callq  0x555555555050 <__stack_chk_fail@plt>
   0x0000555555555189 <+64>:    leaveq
   0x000055555555518a <+65>:    retq
End of assembler dump.
```

Per veure les posicions de cada array hem obert al data l'examine memory. Observem que el nombre 0x7fffffffe064 és la direcció que ocupa l'element 0377. Per tant, com l'array està declarat com un char (1byte), les direccions van de 1 en 1 perquè és l'espai que ocupa cada element. En el cas de que fos un int, sumariem de 4 en 4, ja que cada element enter ocupa per si sol 4 bytes, per aquesta raó van de 4 en 4.

Arribem doncs a la conclusió de que les direccions dels elements de l'array són:

- **Valor 1: 0x7fffffffe065**
- **Valor 2: 0x7fffffffe066**
- **Valor 3: 0x7fffffffe067**



```
0x7fffffffe05c: 0125    0125    0       0       0140    0341    0377    0377
0x7fffffffe064: 0377    01      02      03      0       020     060     0233
0x7fffffffe06c: 0113    0135    0167    0324
(gdb)
```

b)¿Qué sucede ahora con el tamaño total de memoria que ocupa el nuevo array? Explica brevemente el porqué.

El tamany que ocupa el nostre array **no varia**, és el mateix. Simplement, l'array busca posicions de memòria consequtives que estiguin buides per tal de poder substituri-les i guardar els valors de l'array. El que varia son les posicions a les que guarda els valors de l'array.

Exercici 6

a)¿Cuál es el tamaño total de memoria, en bytes, del array c?

El tamany de la memòria és el mateix que a i b ja que l'array c és un array enter igual que a i b per tant ocupa **4 bytes(32 bits)*n**

n = nombre d'elements que té l'array

En aquest cas com n = 4:

4*4 = 16 bytes

```
int main(){
        int i;
        int  a[4]= {1, 2, 3, 4};
        int  b[4]= {5, 6, 7, 8};
        int c[4];
        while (i<4){
                c[i]= a[i] * b[i];
                i++;
                }
```

```
Dump of assembler code for function main:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x50,%rsp
   0x0000555555555155 <+12>:    mov    %fs:0x28,%rax
   0x000055555555515e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555162 <+25>:    xor    %eax,%eax
   0x0000555555555164 <+27>:    movl   $0x1,-0x40(%rbp)
   0x000055555555516b <+34>:    movl   $0x2,-0x3c(%rbp)
   0x0000555555555172 <+41>:    movl   $0x3,-0x38(%rbp)
   0x0000555555555179 <+48>:    movl   $0x4,-0x34(%rbp)
   0x0000555555555180 <+55>:    movl   $0x5,-0x30(%rbp)
   0x0000555555555187 <+62>:    movl   $0x6,-0x2c(%rbp)
   0x000055555555518e <+69>:    movl   $0x7,-0x28(%rbp)
   0x0000555555555195 <+76>:    movl   $0x8,-0x24(%rbp)
=> 0x000055555555519c <+83>:    jmp    0x5555555551c0 <main+119>
   0x000055555555519e <+85>:    mov    -0x44(%rbp),%eax
   0x00005555555551a1 <+88>:    cltq
   0x00005555555551a3 <+90>:    mov    -0x40(%rbp,%rax,4),%edx
   0x00005555555551a7 <+94>:    mov    -0x44(%rbp),%eax
   0x00005555555551aa <+97>:    cltq
   0x00005555555551ac <+99>:    mov    -0x30(%rbp,%rax,4),%eax
   0x00005555555551b0 <+103>:   imul   %eax,%edx
   0x00005555555551b3 <+106>:   mov    -0x44(%rbp),%eax
   0x00005555555551b6 <+109>:   cltq
   0x00005555555551b8 <+111>:   mov    %edx,-0x20(%rbp,%rax,4)
   0x00005555555551bc <+115>:   addl   $0x1,-0x44(%rbp)
   0x00005555555551c0 <+119>:   cmpl   $0x3,-0x44(%rbp)
   0x00005555555551c4 <+123>:   jle    0x55555555519e <main+85>
   0x00005555555551c6 <+125>:   mov    $0x0,%eax
   0x00005555555551cb <+130>:   mov    -0x8(%rbp),%rcx
   0x00005555555551cf <+134>:   xor    %fs:0x28,%rcx
   0x00005555555551d8 <+143>:   je     0x5555555551df <main+150>
   0x00005555555551da <+145>:   callq  0x555555555050 <__stack_chk_fail@plt>
   0x00005555555551df <+150>:   leaveq
   0x00005555555551e0 <+151>:   retq
```

b)¿En qué posición de memoria empieza el array b?

Hem cercat la instrucció on comença a emmagatzemar-se l'array b, **0x0000555555555180 <+55>: movl $0x5,-0x30(%rbp),** per tant la posició de memòria és:

Adreça de memòria rbp = 7fffffffe080

Per tant: 7fffffffe080 - 30 = 7fffffffe050

¿Cuál es la dirección de memoria de la tercera posición de este array?

Si l'array b té quatre posicions, la tercera és on es sitúa el número 7. Per tant, la instrucció que fa el *mov* del valor 7 a un lloc del registre és **0x000000000000118e <+69>: movl $0x7,-0x28(%rbp),** i la posició de memòria és:

Adreça de memòria rbp = 7fffffffe080

Per tant: 7fffffffe080 - 28 = 7fffffffe058

c)La memoria reservada para el array b, ¿Puede empezar 12 posiciones de memoria después de la primera posición de memoria del array a? Explicad brevemente el porqué.

Sí, un enter ocupa 32 bits, és a dir 4 bytes, i l'array està composat de 4 enters, per tant si ens trobem en la posició 0 del nostre array a, ens queden tres enters per començar l'array b, llavors son tres posicions per la quantitat d'espai que ocupa un enter (4 bytes), l'operació seria 3*4, que ens dona 12, per això, son 12 posicions de memòria les que falten per començar l'array b.

Martina Vitagliano i

Exercici 7

a)¿Qué posición de memoria ocupa la variable i?

Hem buscat la posició de memoria del **%rbp** que era 0x7fffffff070 i la de memoria de **i** que era -64(%rbp). Per trobar la posició de memòria de i, hem restat aquests valors que ens donen: **800000007C4C.** Per tant, aquesta és la posició de memòria.

```
#include <stdio.h>

int main(){
        int i, a[20];
        for(i=0;i<20;i++){
                a[i]=i*4;
        }
        return 0;
}
```

```
Dump of assembler code for function main:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
   0x0000555555555151 <+8>:     sub    $0x70,%rsp
   0x0000555555555155 <+12>:    mov    %fs:0x28,%rax
   0x000055555555515e <+21>:    mov    %rax,-0x8(%rbp)
   0x0000555555555162 <+25>:    xor    %eax,%eax
   0x0000555555555164 <+27>:    movl   $0x0,-0x64(%rbp)
   0x000055555555516b <+34>:    jmp    0x555555555184 <main+59>
   0x000055555555516d <+36>:    mov    -0x64(%rbp),%eax
   0x0000555555555170 <+39>:    lea    0x0(,%rax,4),%edx
   0x0000555555555177 <+46>:    mov    -0x64(%rbp),%eax
   0x000055555555517a <+49>:    cltq
   0x000055555555517c <+51>:    mov    %edx,-0x60(%rbp,%rax,4)
   0x0000555555555180 <+55>:    addl   $0x1,-0x64(%rbp)
   0x0000555555555184 <+59>:    cmpl   $0x13,-0x64(%rbp)
   0x0000555555555188 <+63>:    jle    0x55555555516d <main+36>
=> 0x000055555555518a <+65>:    mov    $0x0,%eax
   0x000055555555518f <+70>:    mov    -0x8(%rbp),%rcx
   0x0000555555555193 <+74>:    xor    %fs:0x28,%rcx
   0x000055555555519c <+83>:    je     0x5555555551a3 <main+90>
   0x000055555555519e <+85>:    callq  0x555555555050 <__stack_chk_fail@plt>
   0x00005555555551a3 <+90>:    leaveq
   0x00005555555551a4 <+91>:    retq
End of assembler dump.
```

b)¿A partir de qué posición de memoria encontramos el array a?

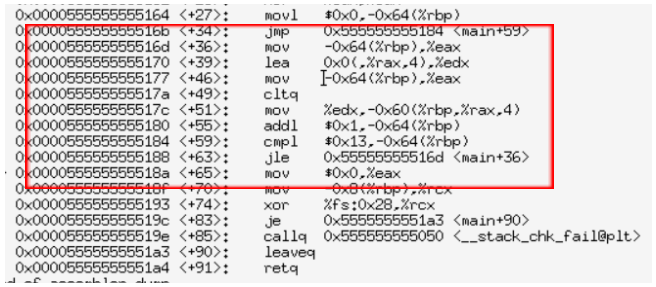
Adreça de memòria de rbp és: 7fffffffe080

Per tant, 7fffffffe080 - 0x64 = **7fffffffe01c**

c)¿En cuál registro se almacena el resultado de la operación i*4?

Com es es pot veure a les instruccions del nostre assembly, el resultat de fer i*4 es **guarda en el registre %edx,** ja que la instrucció que realitza aquest pas és **0x000055555555517c <+51>: mov %edx,-0x60(%rbp,%rax,4)**

d)La forma en la que el compilador ha traducido la estructura iterativa for es un poco diferente a la que vimos en clase de teoría. ¿Podéis indicar las diferencias?



Hi ha més instruccions. Primer fa una igualació de la i a 0 ja que és l'inici de la condició del for. Després realitza un salt a la instrucció de comparació (la 59) amb un jump. En aquesta instrucció compara la i amb 20 per veure si es menor. A partir d'això, si la i es menor a 20, amb un altre jump salta a una instrucció anterior (la 36), on comença fer-se l'operació de la multiplicació de i por 4, i es suma 1 a la variable i. Un cop acabat el bucle, salta a la instrucció 90 per sortir. En els bucles for fets a la teoria de clase, no haviem fet un de la instrucció lea, els jumps els fèiem abans i a més a més, tampoc feiem ús de la comanda cltq.