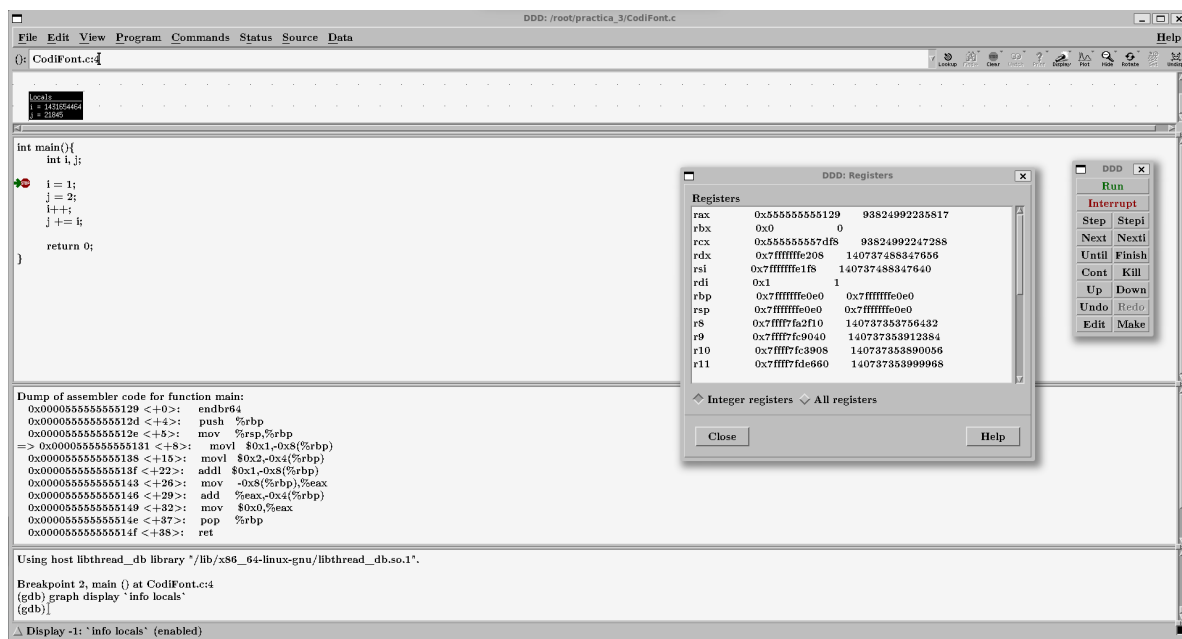


MEMÒRIA: DE C A ASSEMBLY INTRODUCCIÓ

<https://github.com/OVyla/Practica-informatica-3>

Aquest laboratori ens ha proporcionat una experiència valuosa explorant el llenguatge Assembly i entenent la relació entre instruccions en llenguatge C i instruccions Assembly. La utilització del depurador gdb a través de la interfície ddd ens ha permès visualitzar de manera efectiva els canvis en els registres del processador i en la memòria a mesura que s'executen les instruccions. Aquest coneixement ha estat essencial per comprendre millor com funciona un programa a nivell de maquinari i l'optimització el seu rendiment.

Cal mencionar que al principi vam tenir alguna dificultat amb la interfície ddd, ja que no acabàvem de comprendre del tot el seu funcionament.



Imatge: Interfície DDD

EXERCICI 1:

En el primer exercici, hem utilitzat el codi d'exemple de la pràctica:

```
int main() {
    int i, j;

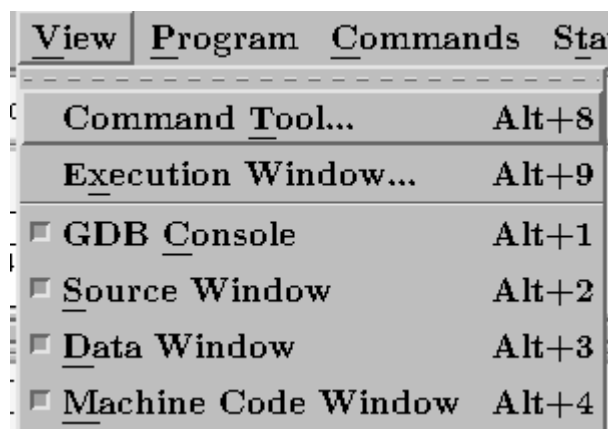
    i = 1;
    j = 2;
    i++;
    j+=1;

    return 0
}
```

Imatge: Codi del primer exercici

- a. En quantes instruccions de llenguatge Assembly es tradueix el programa escrit en C?

El programa escrit en C es tradueix en **10** instruccions de llenguatge Assembly. La primera instrucció *endbr64* no la tenim en compte ja que forma part del codi C. Per poder visualitzar-ho hem seleccionat l'opció *View-Machine Code Window*.



- b. En quina posició de memòria comença a emmagatzemar-se el programa escrit en C?

El programa escrit en C es comença a emmagatzemar a la **primera posició**. Aquesta es troba indicada per `0x0000000000001129 <+0>` (`<+0>` : representen les posicions afegides respecte la posició inicial).

```
Dump of assembler code for function main:
0x000055555555129 <+0>:  endbr64
```

- c. L'assignació del valor 1 a la variable sencera i, com es tradueix a codi Assembly?, ¿i l'assignació del valor 2 a la variable sencera j? Indica la instrucció (què fa), l'operand font i l'operand destí.

El valor i=1 i j=2 es tradueix respectivament en llenguatge Assembly com:

```
=> 0x000055555555131 <+8>:  movl  $0x1,-0x8(%rbp)
    0x000055555555138 <+15>:  movl  $0x2,-0x4(%rbp)
```

Per assignar les variables, s'utilitza la instrucció “movl”. A aquesta instrucció, se li dona un primer argument que representa l'operand font (0x1 (1) i 0x2 (2)) i un segon argument que representa l'operand destí, indicat per %rbp, (-0x8 (i) i -0x4 (j)).

- d. Quina seqüència d'instruccions és emprada per incrementar en 1 el valor de la variable i? Per a cada instrucció indica la instrucció (què fa), l'operand font i l'operand destí.

La seqüència utilitzada per incrementar en 1 el valor de la variable i és:

```
0x00005555555513f <+22>:  addl  $0x1,-0x8(%rbp)
```

Per sumar 1 a la variable i, s'utilitza la instrucció “addl”. A aquesta instrucció, se li dona un primer argument que representa l'operand font (0x1 (1)) i un segon argument que representa l'operand destí (-0x8 (i)).

- e. Quin registre, dels que apareixen a la finestra de Registres, és emprat de forma auxiliar per realitzar les operacions de suma? Quina seqüència de valors pren el registre al llarg de l'execució del programa?

El registre emprat de forma auxiliar que apareix a la finestra *Registres* que realitza les operacions de suma és %eax. Al llarg de l'execució del programa, prendrà el contingut de la posició de memòria “%rbp - 0x8”.

```
0x000055555555143 <+26>:  mov  -0x8(%rbp),%eax
```

- f. Quants bytes ocupen les variables senceres i i j en la seva representació interna?

Les variables senceres i i j ocupen 4 bytes cadascuna en la seva representació interna. Aquesta informació l'obtenim a degut a que estem treballant amb la instrucció “movl”.

```
=> 0x000055555555131 <+8>:  movl  $0x1,-0x8(%rbp)
    0x000055555555138 <+15>:  movl  $0x2,-0x4(%rbp)
```

- g. Com podem saber en quina posició de memòria es troben emmagatzemades les variables senceres i i j.**

Podem obtenir la posició de memòria comparant el llenguatge Assembly amb el codi

C. Les posicions de memòria respectives són: $i \rightarrow \%rbp - 0x8$ $j \rightarrow \%rbp - 0x4$

```
=> 0x00005555555555131 <+8>:    movl  $0x1,-0x8(%rbp)
[ 0x00005555555555138 <+15>:    movl  $0x2,-0x4(%rbp)
```

EXERCICI 2

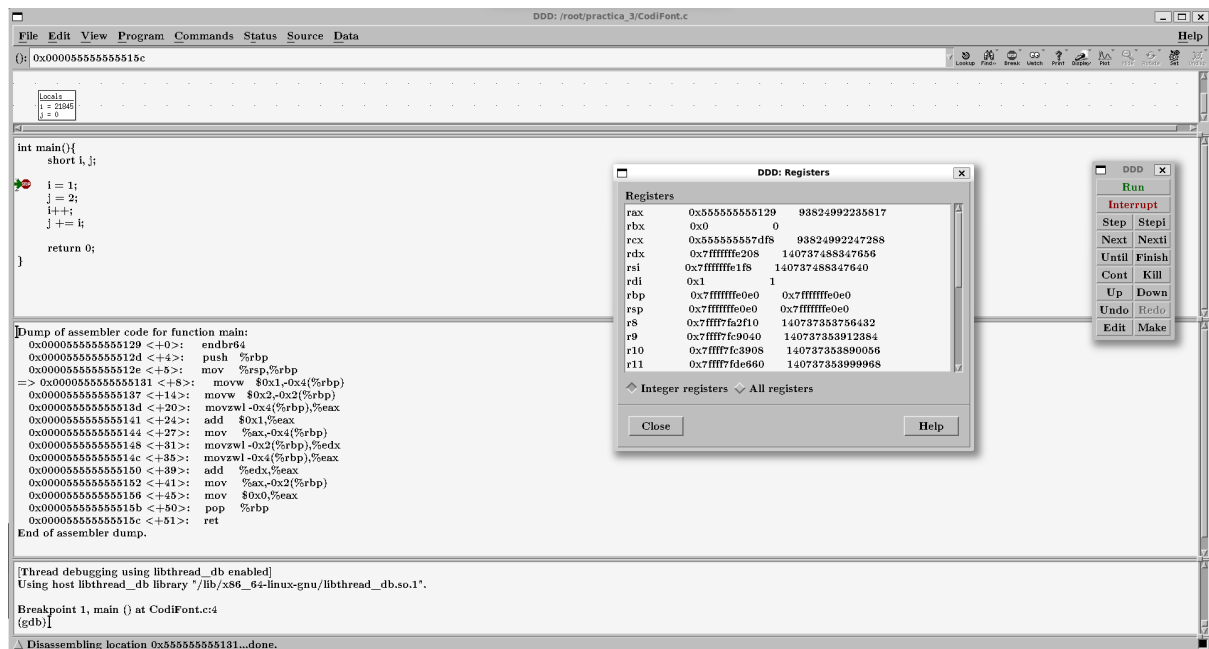
Per aquest exercici, hem treballat sobre el següent codi:

```
int main() {
    short i, j;

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

    return 0
}
```

Imatge: codi en C de l'exercici segon



Imatge: codi assembly del segon exercici

a. Quines diferències veus entre el codi Assembly generat per a aquest cas i el de l'apartat 1?

La primera diferència es troba en el nombre d'instruccions dels dos programes, el primer conté 11 línies i el segon 15.

Cambia la instrucció “*movl*” per “*movw*”, es passa a treballar amb variables de 2 bytes (utilitzem variables short en canvi de variables int).

S'utilitza “*movzwl*” per la conversió de paraules curtes (short) en llargues (long).

Per últim, es diferencia perquè es fan servir dos registres auxiliars en comptes d'un, *%eax* i *%edx*.

b. Quants bytes ocupen les variables tipus short en la seva representació interna?

En la seva representació interna les variables de tipus short ocupen 2 bytes, ho sabem a partir de les instruccions “*movw*” i “*movzwl*”.

c. En què es diferencien els registres *eax* i *ax*?

El registre *%eax* utilitza de 32 bits per realitzar operacions aritmètiques. En canvi, *%ax* és un registre inferior de *%eax* de 16 bits, on s'emmagatzemen el resultat de les variables short. Per garantir que les operacions aritmètiques es realitzen correctament s'utilitzen registres de gran capacitat, però, com en aquest segon programa estem treballant amb variables short (ocupen menys memòria que les variables int), ens hem d'assegurar que només s'utilitzen la quantitat de bits necessària. És per aquesta raó que en aquest programa estem treballant amb dos registres alhora.

d. Què fa la instrucció *movzwl*?

La instrucció “*movzwl*” converteix les paraules curtes (16 bits) en paraules llargues (32 bits). Els bits addicionals s'omplen amb un 0 al registre de destí.

EXERCICI 3:

En el programa de l'exercici 3, les variables enteres a, b, c, i d es declaren i inicialitzen amb valors específics. Després, es realitzen dues operacions diferents amb la variable a: la primera consisteix en sumar b al producte de c i d, mentre que la segona implica sumar b i c i multiplicar el resultat per d:

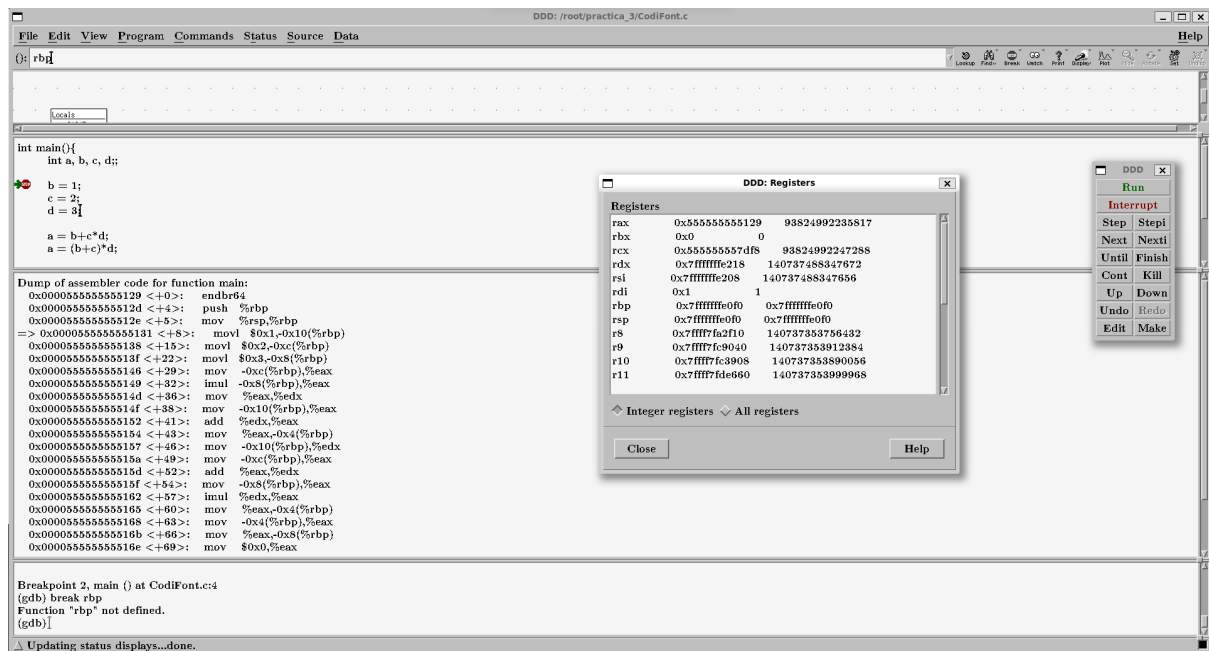
```
int main() {
    int a, b, c, d;

    b = 1;
    c = 2;
    d = 3;

    a = b*c*d;
    a = (b+c)*d;

    d = a;
    return 0
}
```

Imatge: codi en C de l'exercici tercer



Imatge: codi en Assembly del tercer exercici

- a. El codi Assembly generat per a cada expressió és el mateix? Quines diferències existeixen? Per quina raó s'ha generat un codi diferent?**

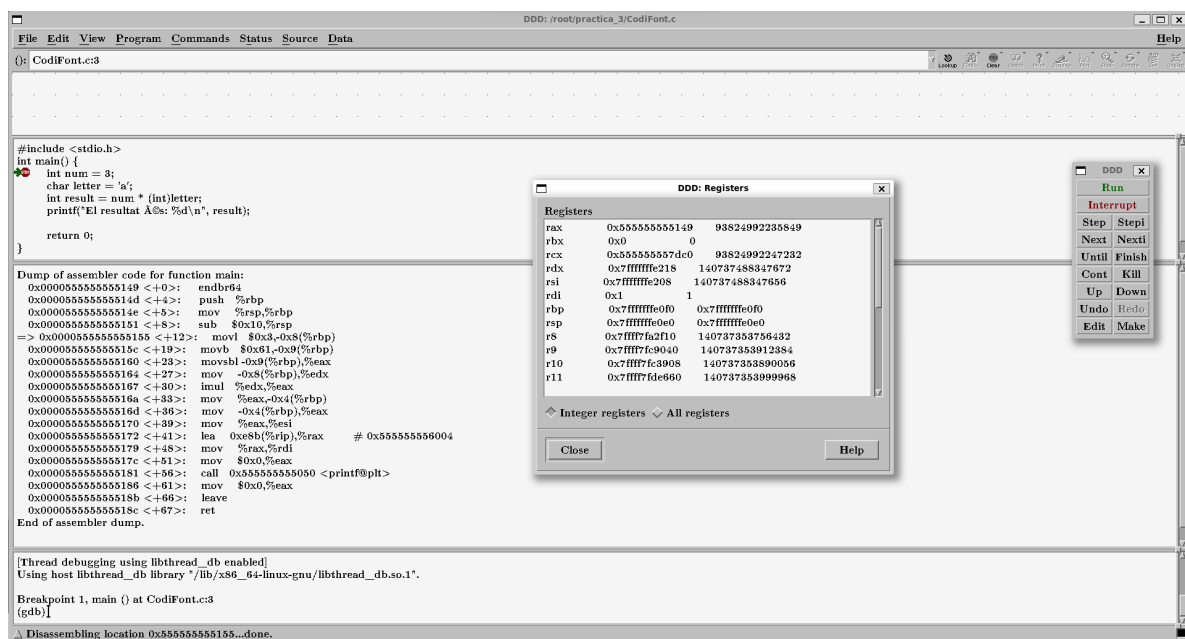
El codi generat en Assembly no és el mateix en els dos casos. Aquest fet es deu a que, en la primera expressió, primer es realitza la multiplicació $c*d$ (“imul -0x8(%rbp),%eax”). Es guarda al registre “%edx” (“mov %eax,%edx”) i després es suma b al resultat (s'emmagatzema al registre %esi). En canvi, en la segona expressió, primer es realitza la suma $b+c$ (“add %eax,%edx”) i després es multiplica pel valor d (“imul %edx,%eax”)

EXERCICI 4:

Per a l'exercici 4, el codi que hem utilitzat comença declarant i inicialitzant dues variables, 'num' com a enter amb el valor 3 i 'letter' com a caràcter amb el valor 'a'. Després, es realitza una operació en la qual es multiplica 'num' pel valor de 'letter' convertit a un enter mitjançant l'ús de la sintaxi '(int)'. Aquesta conversió força el caràcter 'a' a la seva representació numèrica en ASCII:

```
#include <stdio.h>
int main() {
    int num = 3;
    char letter = 'a';
    int result = num * (int)letter;
    printf("El resultat és: %d\n", result);
    return 0;
}
```

Imatge: Codi en C de l'exercici quart



Imatge: Codi en Assembly del quart exercici

a. Quina instrucció s'utilitza en el codi Assembly per efectuar el producte de les dues variables?

Per realitzar el producte de dues variables primerament hem d'emmagatzemar la variable 3 al registre `%eax` a partir de la instrucció `mov`.

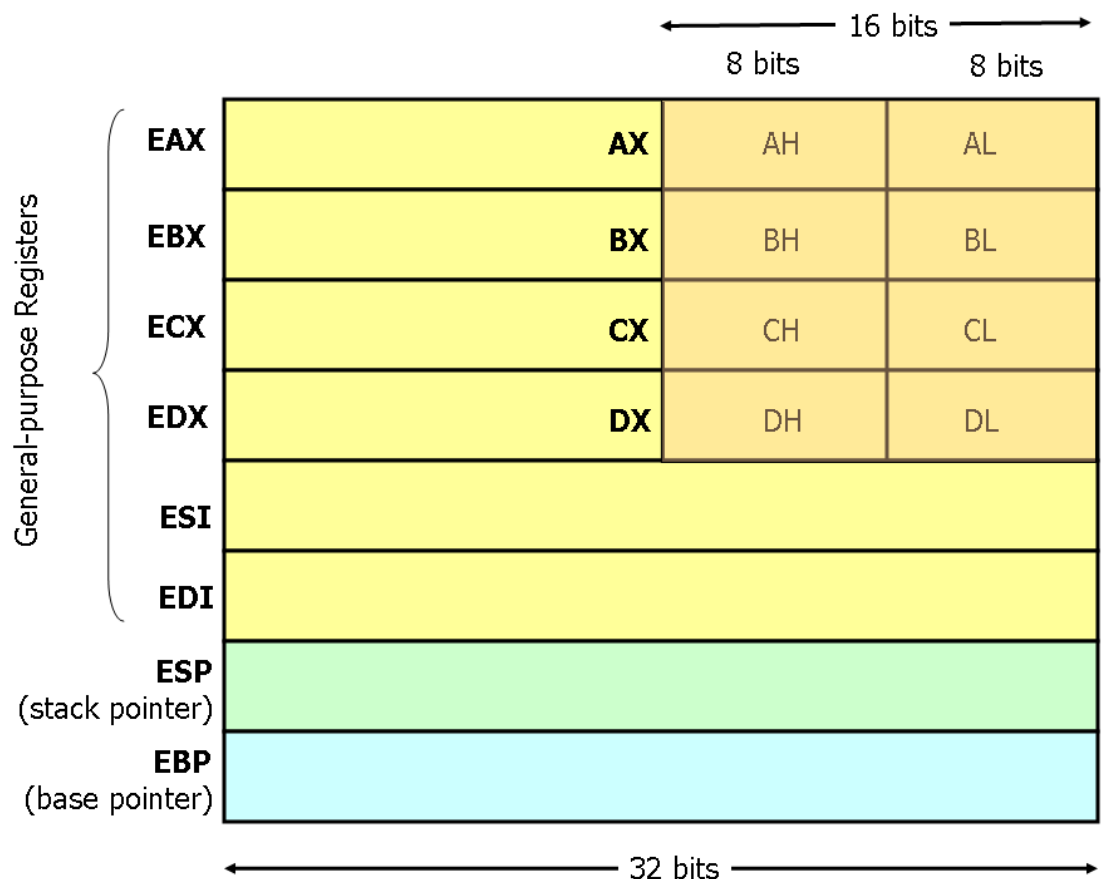
Aquest procés es repeteix però ara emmagatzemant el caràcter a al registre `%ebx`

Seguidament es realitza la multiplicació d'ambdós valors a partir de l'ordre `imul`.

L'operació i el resultat es realitzen dins del registre `%eax`.

b. De quina mida és el resultat del producte?

En l'arquitectura amb la que estem treballant (x86-64), tots els registres que comencen per la lletra "e" contenen 32 bits. Al utilitzar el registre `%eax`, sabem que estem treballant amb 32 bits = 4 bytes.



Imatge: Taula x86

- c. Canvieu el codi perquè el resultat s'emmagatzemi en una variable tipus caràcter. El valor del producte obtingut és el mateix que en el cas anterior? Per quina raó? Expliqueu els canvis que observeu en el codi Assembly i el motiu d'aquests canvis.**

El valor del producte en aquesta modificació de codi serà el mateix que el cas anterior, però en aquest cas el resultat es donarà en forma de char.

La diferència principal es troba en el printf, en el primer programa ens mostrarà per pantalla un valor i en el segon un caràcter.

EXERCICI 5

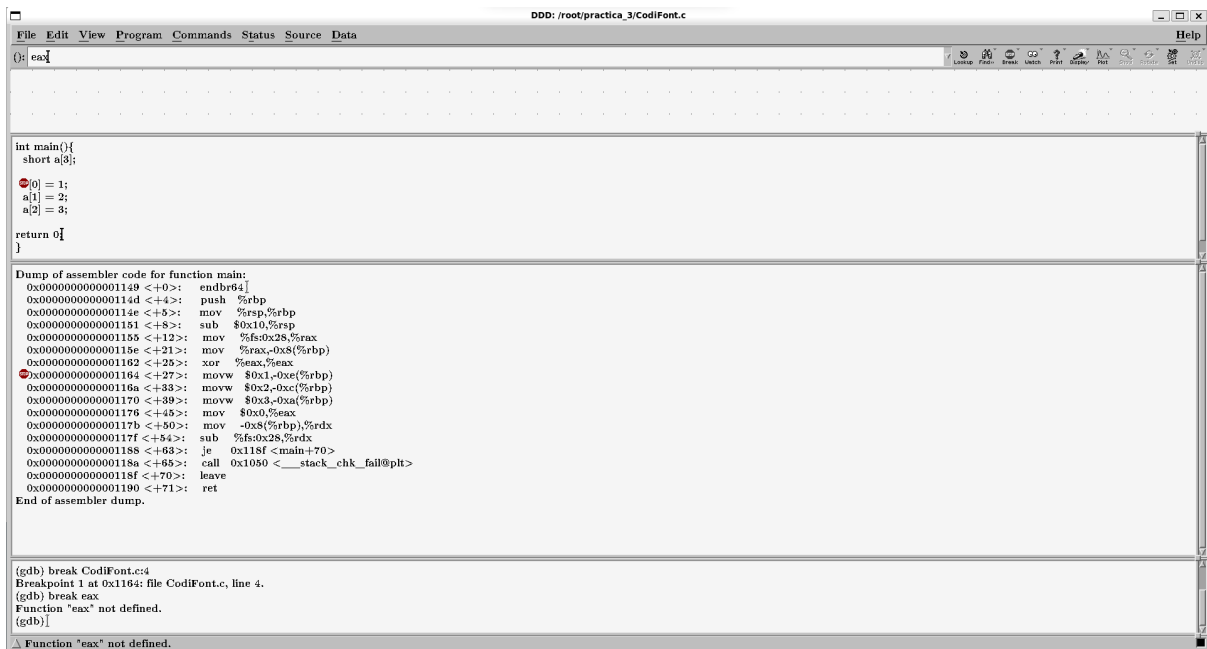
Per a l'exercici 5, el codi en c crea un array de tres elements de tipus short anomenat "a". Seguidament, assigna els valors 1, 2 i 3 als elements consecutius de l'array, respectivament ($a[0] = 1$, $a[1] = 2$, $a[2] = 3$):

```
int main() {
    short a[3];

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

    return 0;
}
```

Imatge: codi en C del cinquè exercici



Imatge: codi en Assembly de l'exercici cinquè

a. Indiqueu les adreces que ocupen ara cada element de l'array:

Al canviar el tipus d'array de short a char, els elements passen d'ocupar 2 bytes a 1 byte.

b. Què succeeix ara amb la mida total de memòria que ocupa el nou array? Explica breument el perquè.

Recordem que estem treballant amb un array de 3 elements. Al canviar el tipus d'array (de short a char), hem passat d'ocupar 6 bytes de memòria a ocupar-ne 3.

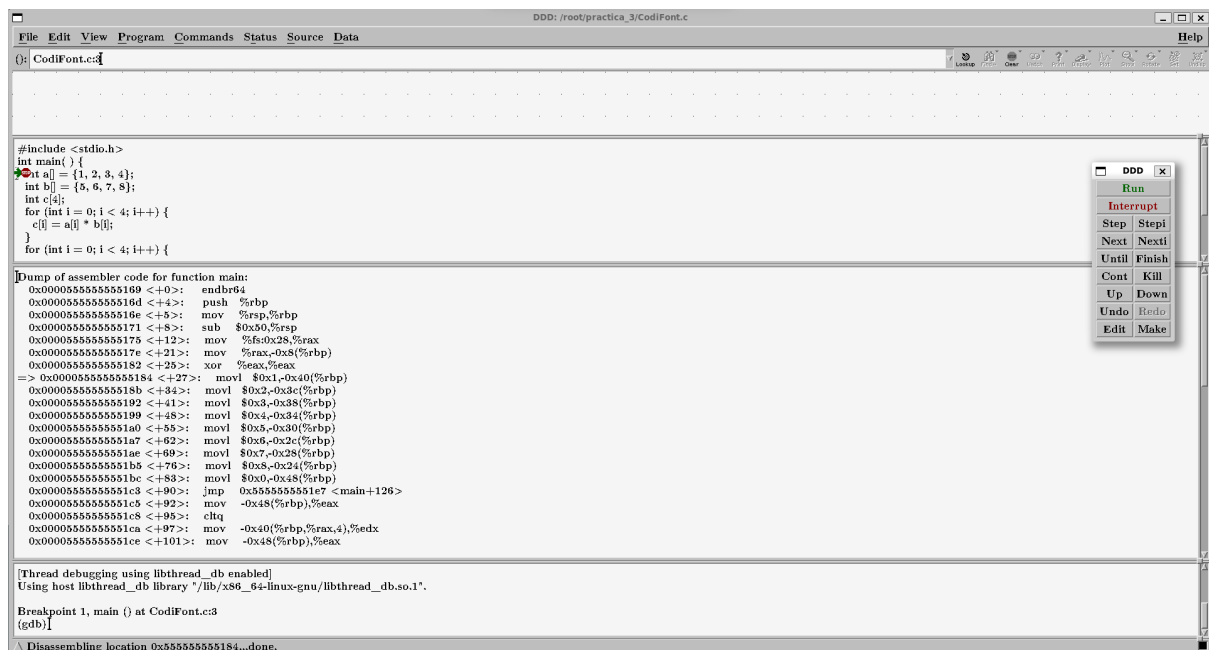
EXERCICI 6

El programa de l'exercici 6, crea dues matrius, a i b, amb valors predefinits. Després, crea una tercera matriu, c, i mitjançant un bucle for multiplica els elements corresponents de les matrius a i b i els emmagatzema a c. Finalment, utilitza un altre bucle for per imprimir els valors de la matriu c a la consola, separats per espais:

```
#include <stdio.h>
int main() {
    int a[] = {1, 2, 3, 4};
    int b[] = {5, 6, 7, 8};
    int c[4];

    for (int i = 0; i < 4; i++) {
        c[i] = a[i] * b[i];
    }
    for (int i = 0; i < 4; i++) {
        printf("%d ", c[i]);
    }
    return 0;
}
```

Imatge: Codi en C del sisè exercici




Imatge: Codi en Assembly de l'exercici sisè

a. Quina és la mida total de memòria, en bytes, de l'array c?

Sabem que l'array és de tipus *int* ja que se'ns indica a l'enunciat de l'exercici. Per tant, estem treballant amb elements de 4 bytes. Com l'array conté 4 posicions, la mida total de memòria és de 16 bytes (4 elements x 4 bytes cadascun).

b. En quina posició de memòria comença l'array b? Quina és l'adreça de memòria de la tercera posició d'aquest array?

L'array b comença en la posició 0x00000000000011a0. La seva adreça de memòria és 0x00000000000011a0.

 0x00000000000011a0 <+55>: movl \$0x5,-0x30(%rbp)

c. La memòria reservada per a l'array b, ¿Pot començar 12 posicions de memòria després de la primera posició de memòria de l'array a? Expliqueu breument el perquè.

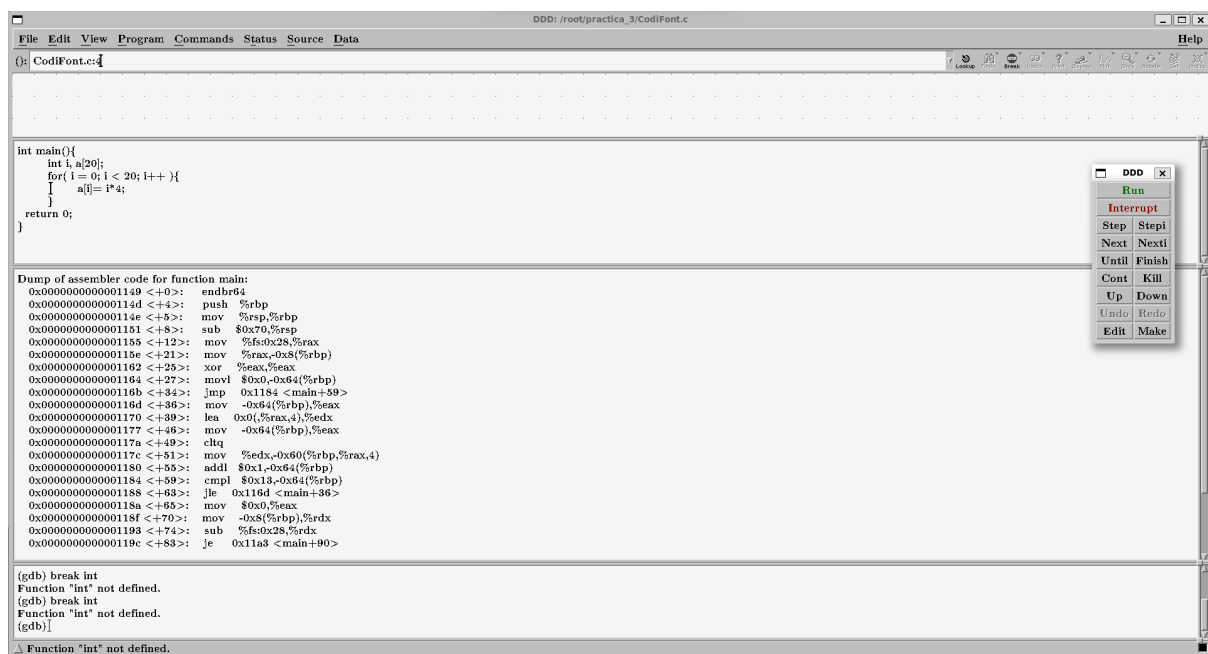
L'array a ocupa 16 posicions de memòria i la primera posició d'aquest ocupa 4 (mencionat a l'apartat a). Per tant, la memòria reservada per l'array b sí que podrà començar 12 posicions després de la primera.

EXERCICI 7

En el codi de l'exercici 7, s'inicialitza un array d'enters amb 20 elements, on cada element és el resultat de la multiplicació de la seva posició per 4. Es comença declarant una variable d'iteració i un array d'enters a amb 20 posicions. Després, s'executa un bucle for que recorre les posicions de l'array mitjançant la variable d'iteració i assigna a cada element el valor de la seva posició multiplicat per 4:

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

Imatge: codi en C de l'exercici setè



Imatge: Codi en Assembly de l'exercici setè

a. Quina posició de memòria ocupa la variable i?

La variable 'i' es troba en la posició 0x0000000000001164 <+27>: movl \$0x0,-0x64(%rbp).

```
0x0000000000001164 <+27>: movl $0x0,-0x64(%rbp)
```

b. A partir de quina posició de memòria trobem l'array a?

Comença en la posició -0x64(%rbp). Per trobar-la, hem buscat la primera instrucció que fa referència a l'array 'a' (movl \$0x0,-0x64(%rbp)). Tot el que es troba després d'aquesta posició de memòria està reservat per a l'array 'a'.

c. En quin registre s'emmagatzema el resultat de l'operació i*8?

El resultat de la multiplicació s'emmagatzema al registre %edx

```
0x000000000000116d <+36>: mov -0x64(%rbp),%eax
0x0000000000001170 <+39>: lea 0x0(,%rax,4),%edx
```

La primera instrucció destina el valor de 'i' al registre %eax, mentre que la segona realitza l'operació i guarda al resultat a %edx.

El resultat i*4 equival a i*8 quan estem treballant amb el registre %edx.

d. La forma en què el compilador ha traduït l'estructura iterativa for és una mica diferent a la que vam veure a classe de teoria. Podeu indicar les diferències?

Les diferències destacades són:

1. L'ús de la instrucció 'lea' per incrementar el valor de 'i' en lloc d'un 'inc' o un 'add'.
2. L'ús de la instrucció 'cltq', encarregada de gestionar els registres %eax (paraula) i rax (paraula llarga)