ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

1A GPSE – Année 2022-2023

ARCHITECTURE DES ORDINATEURS

Nom	Prénom	Table
Corrigé		42

Examen sur table

Durée : 1h30

Ressources autorisées :
- Une feuille A4 manuscrite, recto-verso
- Calculatrice interdite

Répondre directement sur le sujet

CONSIGNES

Ce sujet d'examen est décomposé en trois parties indépendantes, à traiter dans l'ordre qui vous convient :

1. QUESTIONS RAPIDES – 5 points, approx 20 min : Questions de culture générale pouvant traiter de tout point abordé en séance de cours ou présent dans le support de travail.

2. QUESTIONS LONGUES – 5 points, approx 20 min : Idem, mais cette fois-ci les réponses doivent être détaillées.

3. ANALYSE D'ASSEMBLEUR x86-64 – 10 points, approx 50 min : Exercice d'analyse d'un fichier en langage d'assemblage (ISA x86-64) et de traduction en un programme C. Le but est de vérifier votre compréhension de l'interaction entre l'exécution d'un programme et l'utilisation de la stack

1. QUESTIONS RAPIDES

10 questions x 0.5 point

Répondez brièvement, sans justification. Par contre, restez précis dans le choix de vos mots.

Ces questions ont été rédigées par ChatGPT.

1. **CPU.** Quelle est la signification de l'acronyme "CPU" ? Et l'acronyme "GPP" ?

Central Processing Unit

General Purpose Proc

2. **CPU.** Quelles sont les quatre étapes du cycle d'exécution (workflow) d'une instruction ?

Fetch Decode Execute Store

3. **CPU.** Qu'est-ce que la cache (ou mémoire cache) ?

Composant materiel d'un processeur

copie locale de la mémoire

4. **Asm.** Quelle est la signification de l'acronyme "x86" dans l'assembleur x86-64 ?

Heritage du 8086

ISA x86 version 64 bit

5. **Asm.** Quelle est la signification de l'acronyme "IP" dans l'assembleur x86 ?

Instruction

Pointer

6. **Toolchain.** Quel est le rôle d'une chaîne de compilation C (*C toolchain*) ?

C → binaire
portable spécifique à l'archi

7. **Toolchain.** Proposez une commande permettant de compiler un programme à partir d'un fichier source C.

gcc fichier.c -o fichier

8. **Stack.** Que contient le segment de stack (pile) ?

→ var locales non-static
→ sauvegarde de contexte

9. **Stack.** Quelle instruction permet d'empiler des données sur la pile ? Laquelle permet de dépiler des données ?

empiler: push
dépiler: pop

10. **Mémoire.** Quel segment mémoire est utilisé pour stocker les données (et non les variables) allouées dynamiquement ?

Heap
Tas

2. QUESTIONS LONGUES

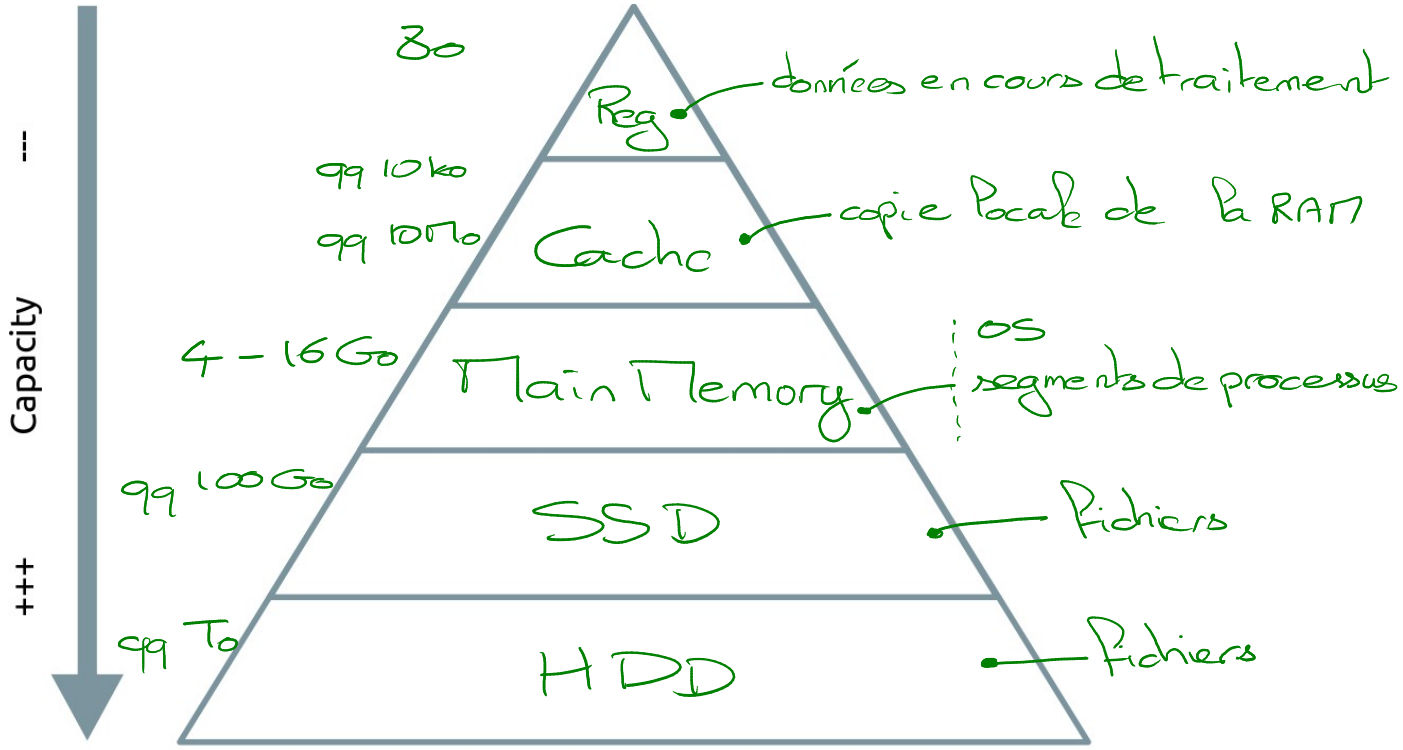
11. **Hiérarchie mémoire (2 pt)**

Considérons tous les composants mémoire d'un ordinateur, allant de l'extérieur à la carte mère, jusqu'à l'intérieur du CPU. On précise que l'on désigne ici un ordinateur personnel (simple ordinateur de bureau, ordinateur portable ou encore PC gaming) mais qu'il n'est pas question de serveur de calcul par exemple.

a) Dans la pyramide, classez ces composants mémoire en fonction de leur capacité de stockage respective.

b) À gauche de la pyramide, complétez le schéma en donnant un ordre de grandeur (ou une fourchette) des valeurs usuelles de capacité ces composants.

c) À droite de la pyramide (ou en dessous selon la place), indiquez les informations contenues dans chacun des composants mémoire (autrement dit : que trouve-t-on dedans ?).



12. **Toolchain (3 pt)**

Tracez un **schéma** présentant les étages de la chaîne de compilation et les fichiers manipulés.

Expliquez avec des **phrases** le rôle de chaque étage.

3. ANALYSE D'ASSEMBLEUR X86-64

Le fichier assembleur `ex3.s` , présenté en annexe, est issu de la compilation (partielle) d'un fichier mystère `ex3.c` . Il a été obtenu grâce à la commande : `gcc -S ex3.c -o ex3.s` .

L'objectif des questions 13 à 17 est d'analyser le code assembleur. Ces questions sont fortement liées mais peuvent être traitées dans le désordre.

Les questions 18 et 19 demandent d'analyser l'évolution de la pile au fil de l'exécution du programme.

L'objectif de la question 20 est de proposer un code C correspondant au fichier assembleur.

13. **(1 pt)** Pour les deux fonctions (`main` et `operation`) du fichier `ex3.s`, on retrouve les deux mêmes premières instructions (lignes 6-7, 21-22).

    Quelle est la **finalité** de ces deux instructions (`pushq %rbp` et `movq %rsp, %rbp`) ?

    Soyez le plus explicite possible, ne pas seulement répondre « poser sur la pile » et « déplacer » ou « affecter ».

    Sauvegarder le Base Pointer de la fct appelante

    Deplacer BP pour indiquer le debut de la zone de la fct appelée

14. **(1 pt)** Observons la troisième instruction de la fonction `main` (ligne 23).

    Quelle est la **finalité** de l'instruction « `subq $16, %rsp` » ?

    Ne pas seulement répondre « soustraire 16 ».

    Creer un ecart de 16o entre SP et BP

    ⇒ Allouer 16o de mémoire pour les var locales du main

15. **(1 pt)** Juste après le traitement étudié question précédente, on aperçoit les instructions « `movl $15, -4(%rbp)` » et « `movl $7, -8(%rbp)` » (respectivement lignes 24 et 25).

Pourquoi des valeurs sont-elles placées dans la pile ? Quelle pourrait-être les instructions en langage C nécessitant ce traitement ?

Ecrit la valeur 15 (7) dans la zone de pile de la fct main()

⇒ pour les stocker dans les variables locales du main()

long a = 15;

long int b = 7;

int

16. **(1 pt)** Juste avant le « `call operation` » (l. 30), on remarque que les registres EDI et ESI sont affectés (l. 28-29). Or quasiment dès le début de la fonction `operation`, ces mêmes registres sont cette fois-ci lus (l. 8-9).

À quoi servent les registres EDI et ESI dans ce cas (au moment d'un appel de fonction) ?

Passage de paramètres

(b) 7 → ESI

(a) 15 → EDI

main()

ESI (=7) → ope_b

EDI (=15) → ope_a

operation()

17. **(1 pt)** La fonction `operation` doit réaliser un certain traitement, pour ensuite retourner son résultat à la fonction `main`.

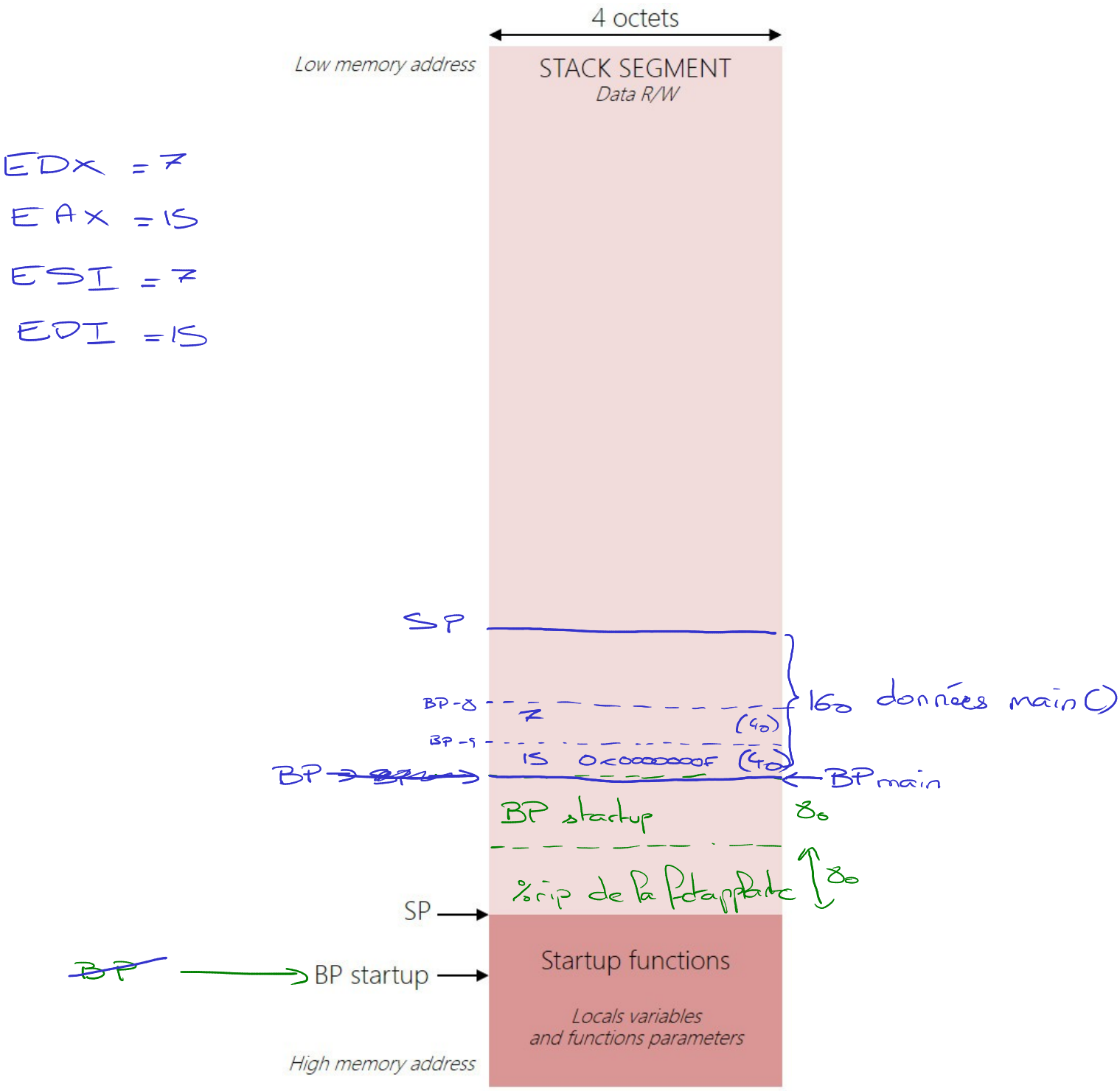
Comment la valeur de retour est-elle passée de la fonction `operation` à la fonction `main` ?

Par le registre eax

18. **(1.5 pt)** Ci-dessous se trouve l'état du segment de pile au point d'entrée du programme. Pour rappel, le point d'entrée du programme est la fonction **`main`** (ici ligne 20).

    En suivant le code assembleur du fichier **`ex3.s`**, complétez la figure pour indiquer l'état de la pile **juste avant exécution de la ligne 30** (« **`call operation`** ») (donc l'instruction ligne 29 vient juste de se terminer).

    Soyez précis en indiquant un maximum d'information (taille des éléments manipulés, valeur des pointeurs SP, BP, …).



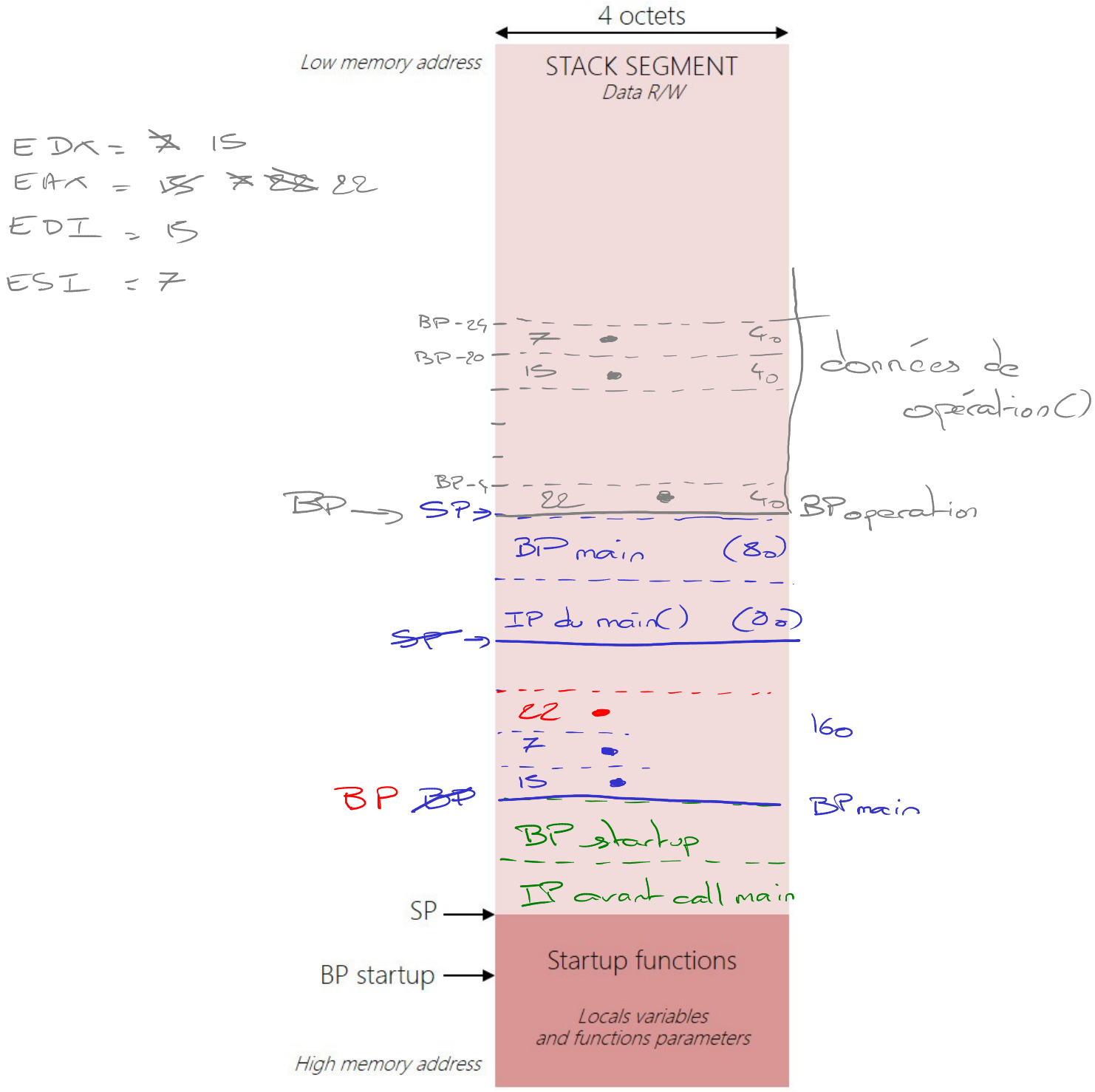
19. **(1.5 pt)** En poursuivant le code assembleur du fichier `ex3.s`, continuez l'exécution mentale du code à partir de l'instruction ligne 30 (« `call operation` ») et indiquez sur la figure ci-dessous l'état de la pile **juste avant l'exécution de la ligne 15 (« `popq %rbp` »)**.

Soyez le plus exhaustif possible.

Indiquez également la valeur contenue dans les registres EAX et EDX à cet instant.

EAX = 22

EDX = 15



20. **(2 pt)** À partir du fichier **ex3.s**, (et éventuellement des réponses aux questions précédentes) proposez un code C qui pourrait correspondre au code assembleur donné.

Conseil : il n'y a pas de solution unique.

```
int operation(int ope_a, int ope_b) {
    int ope_c;
    ope_c = ope_a + ope_b;

    return ope_c;
}

int main(void) {
    int a = 15;
    int b = 7;
    int c;
    c = operation(a, b);

    return 0;
}
```

ANNEXE EXERCICE 3

Fichier ex3.s à analyser

```
File: ex3.s
```

```
 1      .file   "ex3.c"
 2      .text
 3      .globl  operation
 4      .type   operation, @function
 5  operation:                          réception d'arguments
 6      pushq   %rbp
 7      movq    %rsp, %rbp
 8      movl    %edi, -20(%rbp)         var locale
 9      movl    %esi, -24(%rbp)         var locale
10      movl    -20(%rbp), %edx
11      movl    -24(%rbp), %eax
12      addl    %edx, %eax
13      movl    %eax, -4(%rbp)          var locale
14      movl    -4(%rbp), %eax          paramètre de retour
15      popq    %rbp
16      ret
17      .size   operation, .-operation
18      .globl  main
19      .type   main, @function
20  main:
21      pushq   %rbp
22      movq    %rsp, %rbp
23      subq    $16, %rsp
24      movl    $15, -4(%rbp)           var locale
25      movl    $7, -8(%rbp)            var locale
26      movl    -8(%rbp), %edx
27      movl    -4(%rbp), %eax          passage d'arguments
28      movl    %edx, %esi
29      movl    %eax, %edi
30      call    operation
31      movl    %eax, -12(%rbp)         var locale   récupération de la valeur de retour de fonction
32      movl    $0, %eax
33      leave                           ; Voir la note en fin de cette page
34      ret
35      .size   main, .-main
36      .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
37      .section    .note.GNU-stack,"",@progbits
```

Note : l'instruction ***leave*** *réalise en réalité deux opérations :*

```
leave  <=>    movq %rbp, %rsp
              popq %rbp
```