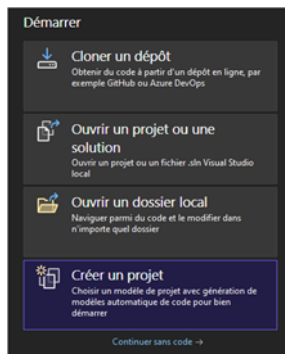


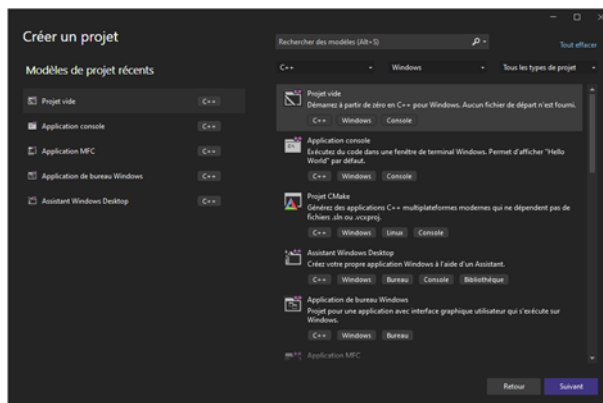
## 1. Travail préparatoire

Dans ce TP nous allons écrire des programmes C++ qui vont appeler des fonctions écrites en ASMx64. Nous allons pour cela utiliser l'environnement Microsoft Visual Studio sous Windows qui intègre aussi bien un compilateur C++ que le compilateur assembleur de Microsoft MASM. Les étapes décrites ci-dessous devront être répétées chaque fois que vous voudrez créer un programme C++ qui exploite de l'assembleur sous l'environnement Visual.

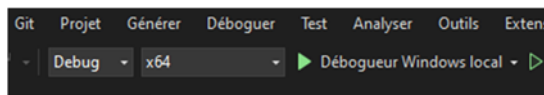
1. Lancez Visual Studio 2022
2. Choisir « Créer un projet » pour créer un nouveau projet.



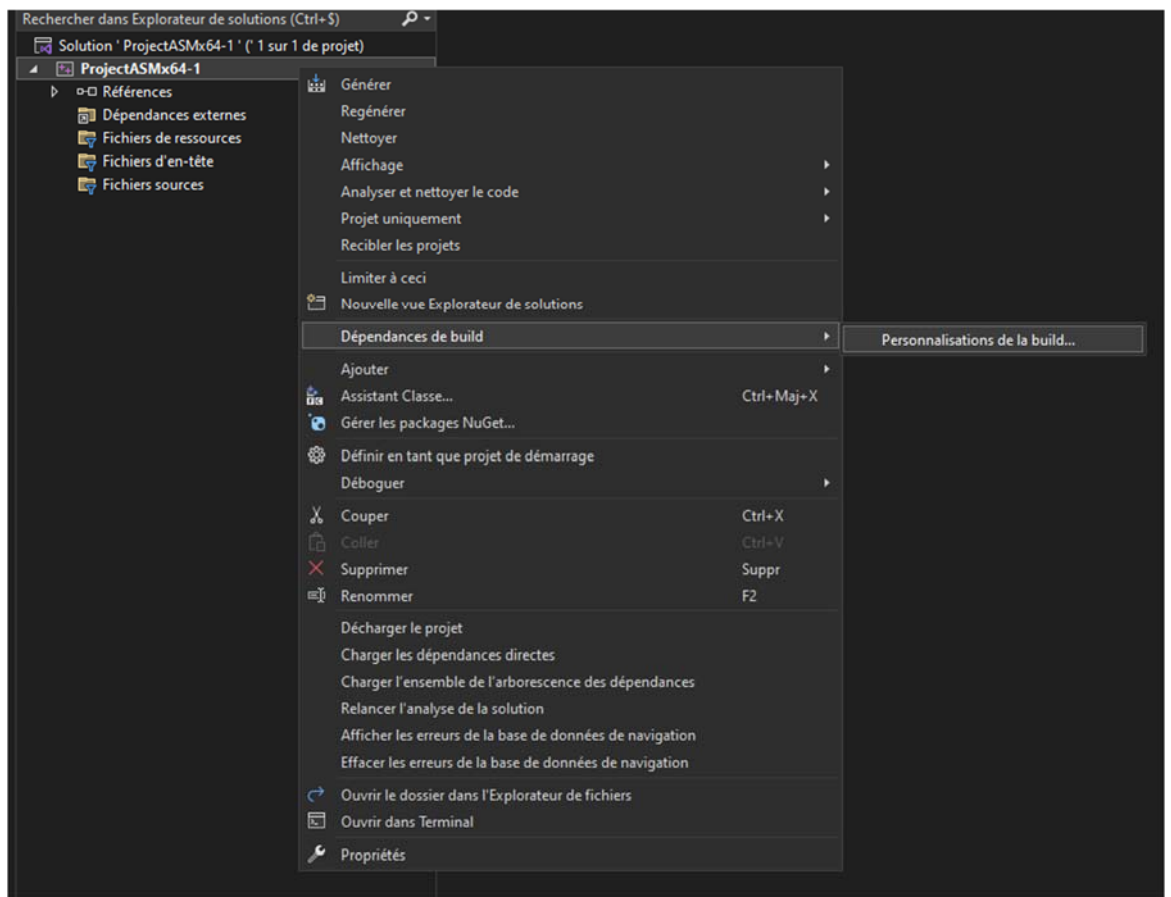
3. Choisir la première option à droite « Projet vide » (C++).



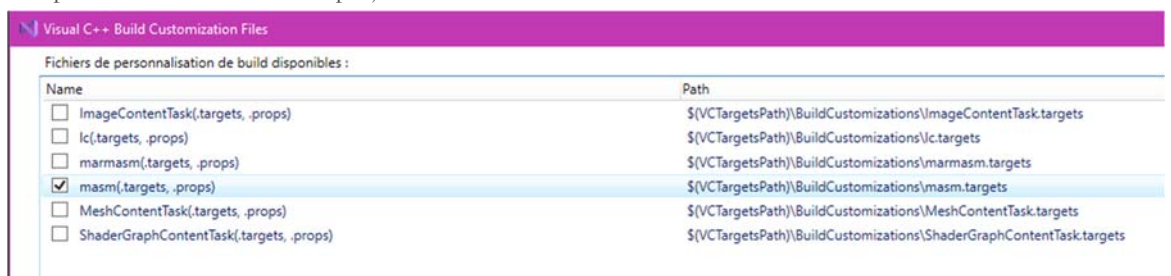
4. Dans la fenêtre suivante, choisissez un nom pour votre projet (par exemple ProjectASMx64-1) ainsi que l'emplacement où vous voulez mettre votre projet. Cochez la case « Placer la solution et le projet dans le même répertoire » afin de ne pas multiplier les dossiers. Enfin appuyez sur le bouton « Créer ».
5. Une fois que la fenêtre principale du projet s'ouvre, vérifiez dans la barre d'outils que votre projet est bien en mode Debug x64, sinon modifiez dans la barre d'outils pour passer dans ce mode. Ceci est nécessaire car nous voulons utiliser l'ASM x64 ainsi que le debugger qui va nous permettre d'exécuter les instructions une par une.



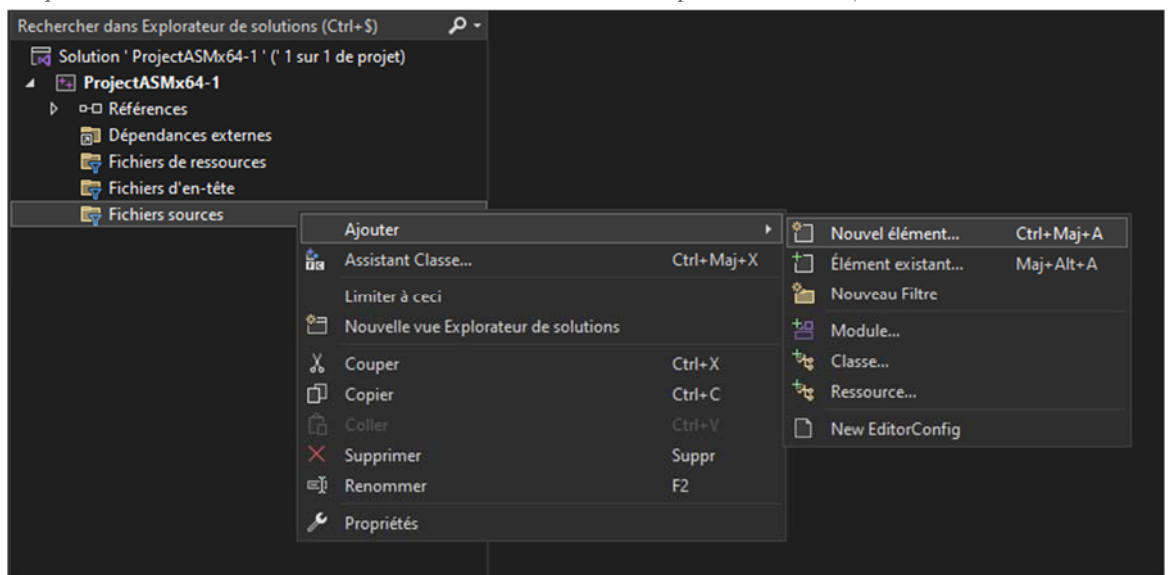
6. Cliquez avec le bouton **droit** de la souris sur votre projet (pas sur la solution) puis choisissez dans le menu qui apparaît « Dépendances de build » et « Personnalisations de la build... »



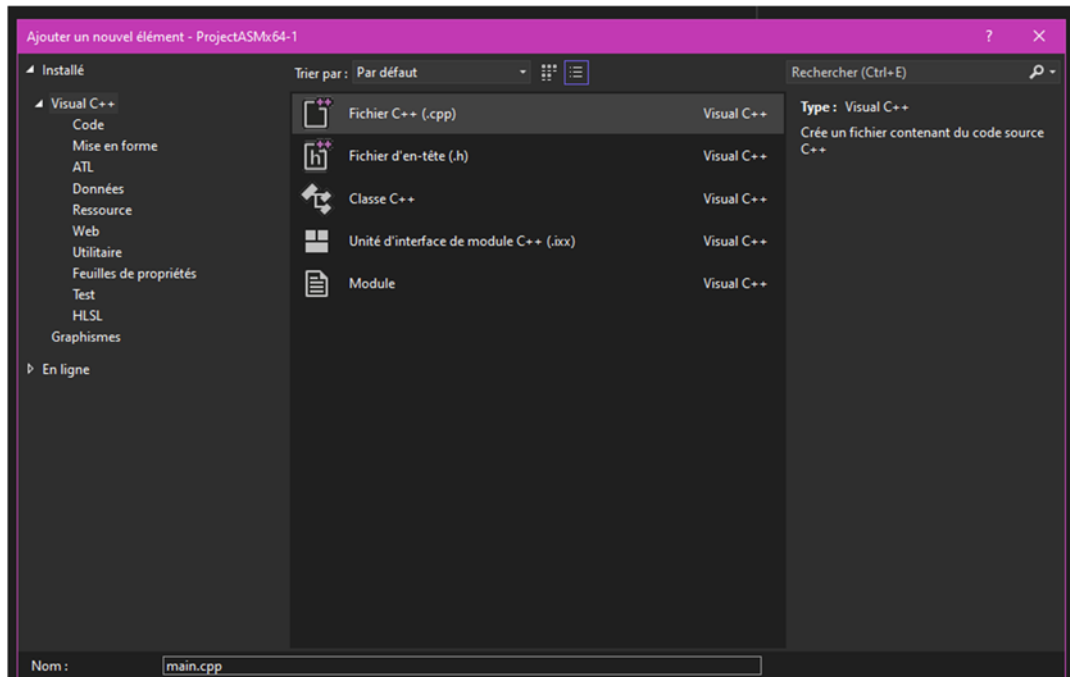
7. Dans la fenêtre qui apparaît cochez la case `masm(.targets,.props)`. Ceci va nous permettre d'exploiter le compilateur MASM dans notre projet C++. Validez avec OK.



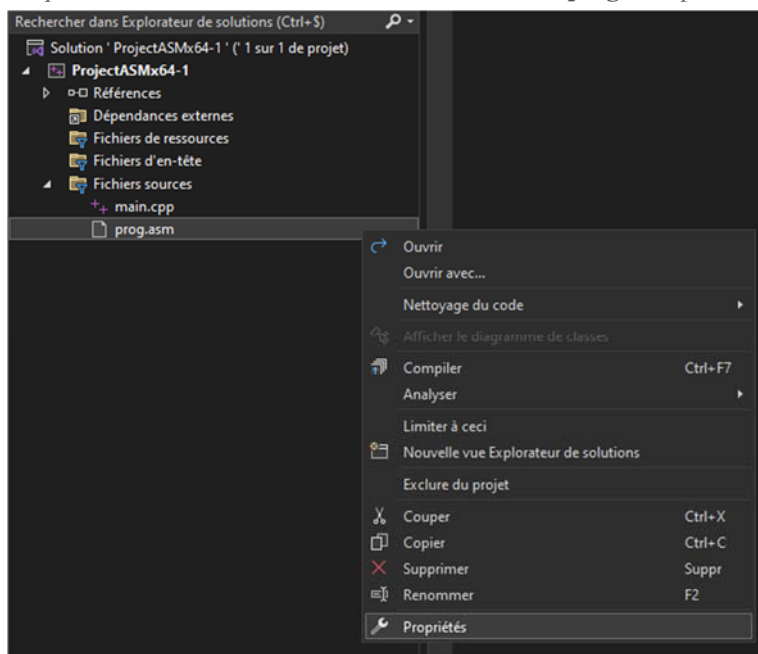
8. Cliquez avec le bouton **droit** de la souris sur « Fichiers sources » puis choisissez Ajouter et Nouvel élément...



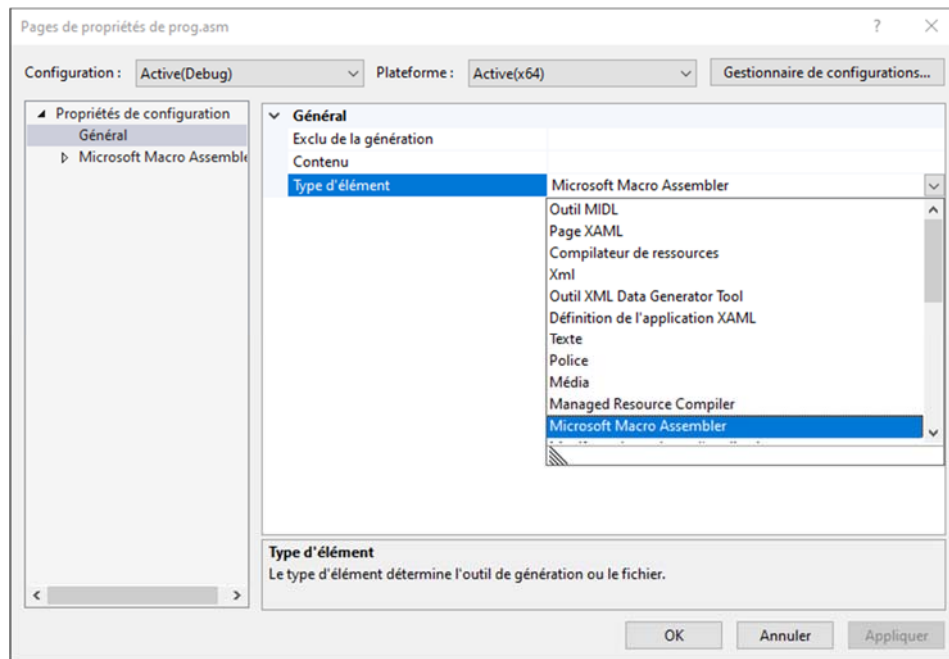
9. Dans la fenêtre qui apparaît, cliquez sur Fichier C++ (.cpp) et choisissez **obligatoirement** comme nom de fichier **main.cpp**. Ce fichier contiendra notre programme principal en C++.



10. Cliquez de nouveau avec le bouton **droit** de la souris sur « Fichiers sources » puis choisissez Ajouter et Nouvel élément...
11. Dans la fenêtre qui apparaît, cliquez sur Fichier C++ (.cpp) et choisissez comme nom de fichier **prog.asm**. **Attention** : l'extension **.asm** est importante. Ce fichier contiendra la partie de notre programme en ASM x64.
12. Cliquez avec le bouton **droit** de la souris sur le fichier **prog.asm** puis choisissez **Propriétés**



13. Dans la fenêtre qui apparaît, vérifiez que 'Type d'élément' est bien sur Microsoft Macro Assembler, sinon sélectionnez Type d'élément puis le choisir dans la liste déroulante.



14. La création et la configuration de notre projet sont à présent terminées et nous allons pouvoir commencer à remplir le fichier source **main.cpp** qui contiendra la partie C++ de notre programme et le fichier source **prog.asm** qui contiendra la partie ASM x64 de notre programme.

## 2. Premier programme exploitant l'ASM x64

1. Tapez le code ci-dessous dans le fichier **main.cpp**

```
#include <iostream>
using namespace std;

extern "C" int somme(int a, int b);

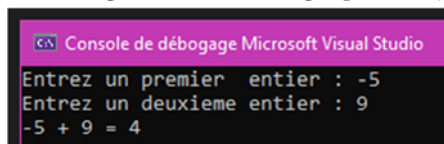
int main()
{
    int a, b;

    cout << "Entrez un premier entier : "; cin >> a;
    cout << "Entrez un deuxieme entier : "; cin >> b;
    cout << a << " + " << b << " = " << somme(a,b) << endl;
    return 0;
}
```

2. Tapez le code ci-dessous dans le fichier **prog.asm**

```
.CODE
    somme PROC
        MOV EAX,ECX
        ADD EAX,EDX
        RET
    somme ENDP
END
```

3. Compilez et exécutez le programme (sans débbuger) en cliquant sur **CTRL-F5**



```
Console de débbugage Microsoft Visual Studio
Entrez un premier entier : -5
Entrez un deuxieme entier : 9
-5 + 9 = 4
```

### Explication du programme

Le fichier **main.cpp** contient un programme principal C++ classique qui saisit 2 entiers puis appelle une fonction **somme** qui va retourner comme résultat la somme de ces 2 entiers. La particularité de ce programme est que la fonction **somme** n'est pas écrite dans le fichier **main.cpp** mais dans un autre fichier et dans un autre langage (ici ASM x64) qui utilise une convention d'appel spécifique. C'est pourquoi il faut ajouter le qualificatif **extern "C"** lors de la déclaration de la fonction.

Le fichier **prog.asm** contient lui la définition de la fonction **somme** en ASM x64. Lorsque le programme principal C++ va appeler la fonction **somme**, il va placer le premier paramètre (l'entier **a**) dans le registre **ECX** et le deuxième paramètre (l'entier **b**) dans le registre **EDX** et il récupérera la valeur de retour à partir du registre **EAX**. C'est pourquoi la fonction **somme** copie d'abord **ECX** dans **EAX**, puis ajoute le contenu de **EDX** à **EAX**.

### Rappel

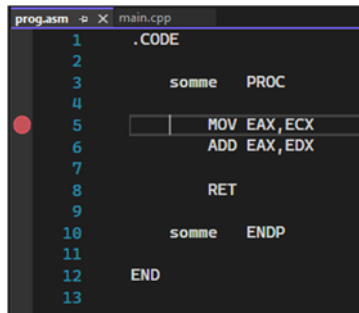
Le registre utilisé dans Visual, pour passer un paramètre donné, dépend de la position du paramètre et de son type. Dans Visual Studio, int est équivalent à long int.

Taille Position	char (8 bits)	short int (16 bits)	int ou long int (32 bits)	long long int (64 bits)
1	CL	CX	<b>ECX</b>	RCX
2	DL	DX	<b>EDX</b>	RDX
3	R8B	R8W	R8D	R8
4	R9B	R9W	R9D	R9
>4	Pile	Pile	Pile	Pile
Valeur de retour	AL	AX	<b>EAX</b>	RAX

### 3. Utilisation du débogueur

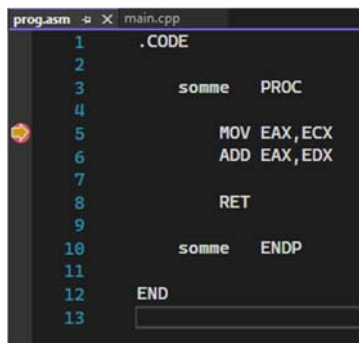
L'écriture d'un programme en assembleur peut parfois s'avérer complexe. Lorsque le programme ne donne pas le résultat souhaité il peut être fastidieux d'arriver à trouver le bug. Heureusement nous pouvons utiliser le débogueur intégré à Visual Studio pour exécuter les instructions du programme une par une et observer après chaque instruction les valeurs des registres et des variables. Cela pourra aider à identifier précisément l'origine d'un éventuel bug.

1. Avant de démarrer l'exécution du programme en mode debug, nous allons placer un point d'arrêt sur une instruction de notre programme. Ainsi quand nous démarrerons l'exécution du programme celle-ci s'effectuera normalement jusqu'à atteindre l'instruction choisie et quand celle-ci sera atteinte, l'exécution se mettra en pause afin de nous permettre d'observer les valeurs des registres et/ou des variables.
2. Placer le curseur sur la ligne `MOV EAX, ECX` du fichier **prog.asm** puis appuyez sur la touche **F9**. Un cercle rouge apparaît alors à gauche de la ligne indiquant qu'un point d'arrêt a été défini sur cette ligne.



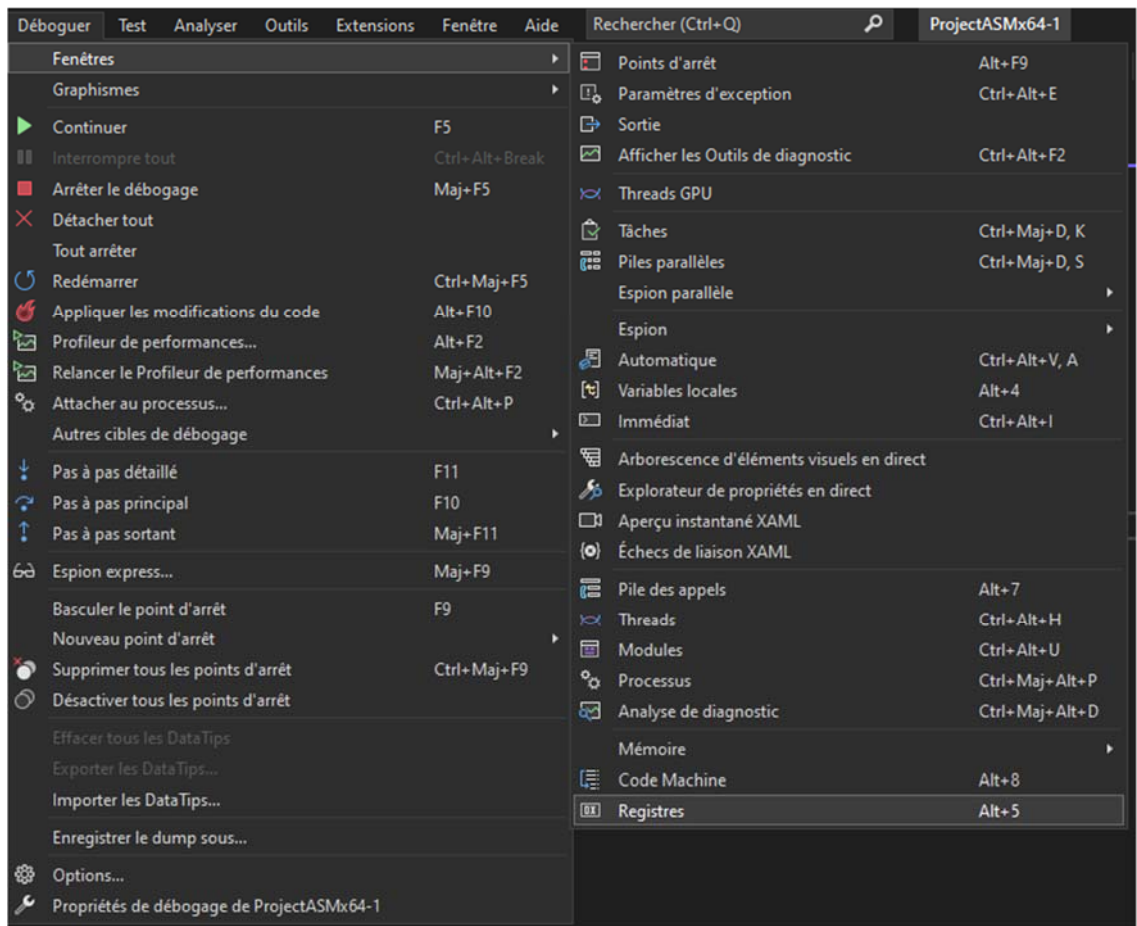
```
1 .CODE
2
3     somme PROC
4
5     MOV EAX, ECX
6     ADD EAX, EDX
7
8     RET
9
10    somme ENDP
11
12    END
13
```

3. Démarrer maintenant l'exécution du programme en mode debug en appuyant sur **F5**.
4. Saisissez comme **valeurs 5 et 7**.
5. Vous constatez que l'exécution du programme s'est mise en pause. Revenez à la fenêtre de l'éditeur. Remarquez maintenant la petite flèche à l'intérieur du cercle rouge, indiquant que l'exécution est en pause. L'instruction `MOV EAX, ECX` n'a pas encore été exécutée.

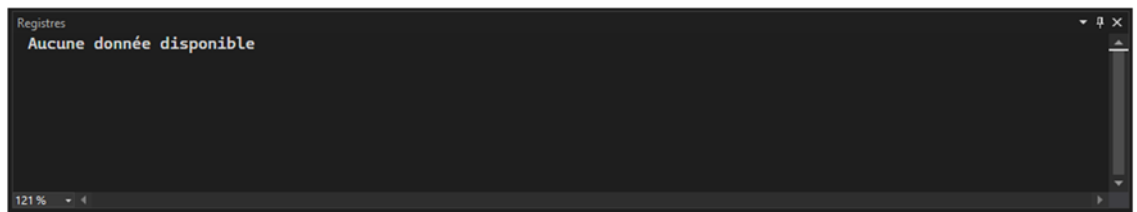


```
1 .CODE
2
3     somme PROC
4
5     MOV EAX, ECX
6     ADD EAX, EDX
7
8     RET
9
10    somme ENDP
11
12    END
13
```

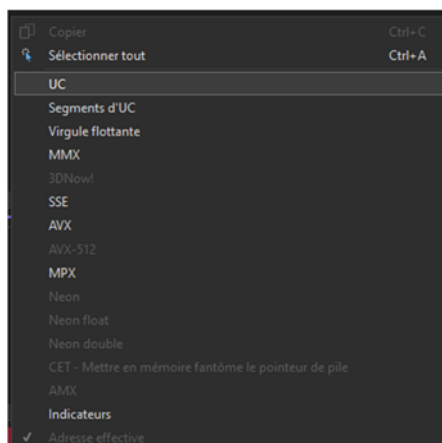
6. Nous allons maintenant demander l'affichage des valeurs actuelles des registres du processeur. Pour cela déroulez le menu **Déboguer** et choisissez **Fenêtres** puis **Registres** tout en bas (ou appuyez sur ALT+5).



7. Une nouvelle fenêtre appelée **Registres** apparaît en bas à gauche mais aucune donnée n'est affichée dans cette fenêtre.

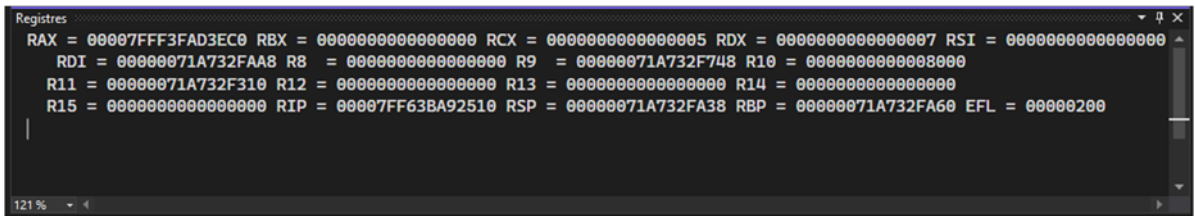


8. Nous allons maintenant choisir quelle catégorie de registres nous désirons afficher dans cette fenêtre. Cliquez dans cette fenêtre avec le bouton droit de la souris puis choisissez UC.

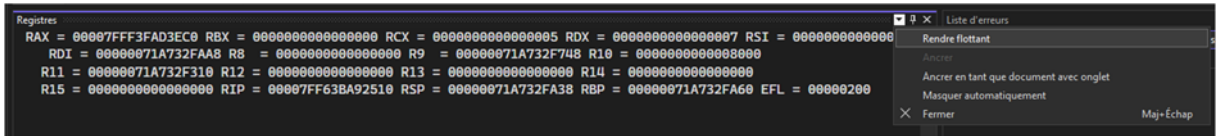


9. Les registres de l'UC apparaissent maintenant.

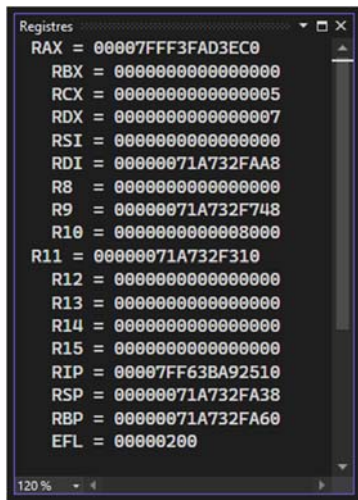




10. Pour une meilleure lisibilité nous allons rendre cette fenêtre flottante afin de pouvoir la redimensionner. Pour cela cliquez sur le petit triangle en haut à droite et choisir rendre flottant.

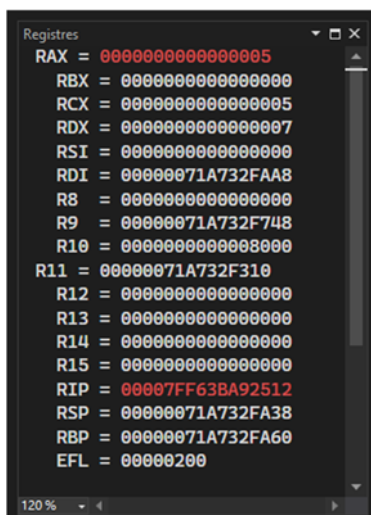


11. Redimensionnez ensuite la fenêtre pour obtenir un affichage proche de celui-ci :



Vous constatez qu'il n'y a pas de registre qui s'appelle EAX, ECX ou EDX. En effet, le registre EAX (32 bits) n'est que la partie basse du registre RAX (64 bits). De ce fait le registre EAX correspond tout simplement au 8 chiffres hexa de droite du registre RAX (ici 3FAD3EC0). De même le registre ECX correspond à la partie basse du registre RCX (ici 00000005 qui correspond à la valeur du premier **paramètre a**). Enfin le registre EDX correspond à la partie basse du registre RDX (ici 00000007 qui correspond à la valeur du deuxième **paramètre b**).

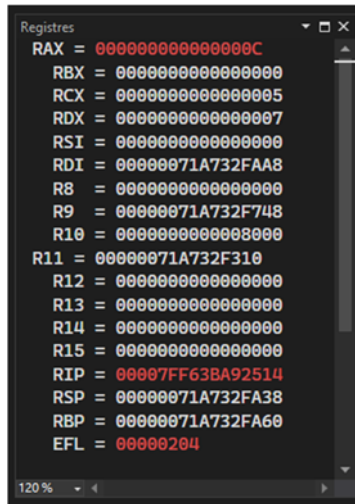
12. Nous sommes maintenant prêts à exécuter l'instruction MOV EAX, ECX. Pour cela appuyez dans la fenêtre de l'éditeur sur la **touche F11**.



Après l'exécution de cette instruction, apparaissent en rouge les registres modifiés. On voit que le registre EAX a bien pris la valeur du registre ECX (00000005) et que le registre (RIP, pointeur d'instruction) contient maintenant l'adresse de l'instruction suivante ADD EAX, EDX. Appuyez de nouveau sur **F11** pour exécuter cette instruction.



13. Maintenant le registre ECX contient la valeur 0000000C ( $12_{10}$ ) qui correspond à la somme de l'ancienne valeur du registre EAX ( $5_{10}$ ) et du registre EDX ( $7_{10}$ ). Cette fois le registre EFL (registre des indicateurs) est aussi modifié car il est mis à jour après chaque exécution d'une instruction arithmétique ou logique.



14. Appuyez maintenant sur **F5** pour reprendre l'exécution normale sans pause jusqu'à la fin ou bien **MAJ+F5** pour arrêter le débogage.
15. Pour supprimer le point d'arrêt, placez de nouveau le curseur sur la ligne MOV EAX, ECX et appuyez sur **F9** (ou CTRL-MAJ-F9 pour supprimer tous les points d'arrêt même s'il n'y en a ici qu'un seul).

## 4. Programme avec variables

Sauvez le projet précédent (**Fichier**→**Enregistrer Tout**) puis fermez le (**Fichier**→**Fermer la solution**).

En reprenant les étapes de la section 1 (**Travail préparatoire**) créer un nouveau projet que vous appellerez **Polynome2int** puis tapez le code ci-dessous dans le fichier **prog.asm** :

```
.DATA

a      QWORD ?
b      QWORD ?
c      QWORD ?
n      QWORD ?

tmp    QWORD ?

.CODE

polynome2int PROC

    ; la fonction appelante passe a dans RCX, b dans RDX,
    ; c dans R8 et n dans R9

    MOV     a,RCX
    MOV     b,RDX
    MOV     c,R8
    MOV     n,R9

    ; ici IMUL fait une multiplication entière signée de RAX
    ; (64 bits) par un autre registre ou une autre variable
    ; entière de 64 bits et stocke le résultat sur 128 bits dans
    ; RDX:RAX mais on supposera ici que le résultat tient sur
    ; 64 bits seulement (soit RAX)

    MOV     RAX,n
    IMUL     n           ; calcul de n^2
    IMUL     a           ; calcul de a*n^2
    MOV     tmp,RAX      ; stockage du résultat temporaire

    MOV     RAX,n
    IMUL     b           ; calcul de b*n

    ADD     RAX,tmp      ; calcul de a*n^2 + b*n
    ADD     RAX,c        ; calcul de a*n^2 + b*n + c

    RET                     ; le résultat à retourner est dans RAX

polynome2int ENDP

END
```

Tapez ensuite le code ci-dessous dans le fichier **main.cpp**

```
#include <iostream>
using namespace std;

extern "C" long long polynome2int(long long a, long long b, long long c, long long n);

int main()
{
    long long a,b,c,n;

    cout << endl << "Ce programme sert a calculer la valeur d'un polynome entier de
second degre a*n^2 + b*n + c" << endl << endl;

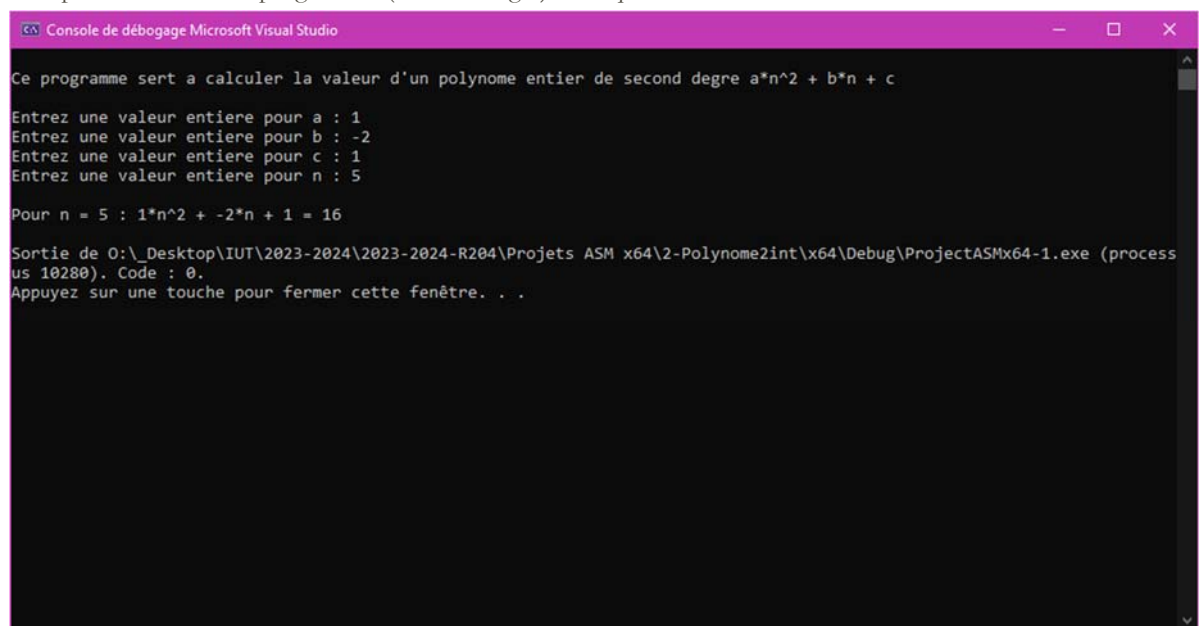
    cout << "Entrez une valeur entiere pour a : "; cin >> a;
    cout << "Entrez une valeur entiere pour b : "; cin >> b;
    cout << "Entrez une valeur entiere pour c : "; cin >> c;

    cout << "Entrez une valeur entiere pour n : "; cin >> n;

    cout << endl << "Pour n = " << n << " : " << a << " * n^2 + " << b << " * n + " << c
<< " = " << polynome2int(a, b, c, n) << endl;

    return 0;
}
```

Compilez et exécutez le programme (sans débbuger) en cliquant sur **CTRL-F5**



```
Console de débbugage Microsoft Visual Studio

Ce programme sert a calculer la valeur d'un polynome entier de second degre a*n^2 + b*n + c

Entrez une valeur entiere pour a : 1
Entrez une valeur entiere pour b : -2
Entrez une valeur entiere pour c : 1
Entrez une valeur entiere pour n : 5

Pour n = 5 : 1*n^2 + -2*n + 1 = 16

Sortie de O:\_Desktop\IUT\2023-2024\2023-2024-R204\Projets ASM x64\2-Polynome2int\x64\Debug\ProjectASMx64-1.exe (process
us 10280). Code : 0.
Appuyez sur une touche pour fermer cette fenetre. . .
```

### Explication du programme

Ce programme permet de calculer la valeur d'un polynôme de second degré  $a*n^2+b*n+c$  pour des valeurs entières données de a, b, c et n.

Le programme C++ principal s'occupe de saisir les valeurs de a, b, c et n puis appelle la fonction **polynome2int** (écrite en ASM x64) qui calcule la valeur du polynôme et retourne le résultat. Le programme principal affiche alors ce résultat.

Comme on peut le voir ici le programme principal a besoin de passer 4 paramètres à la fonction assembleur. Lors de l'appel, les paramètres a, b, c et n sont passés automatiquement et respectivement à travers les registres RCX, RDX, R8 et R9. Le résultat est récupéré à travers le registre RAX. En effet le programme principal utilise des variables de type long long int c'est-à-dire de 64 bits (voir le tableau dans la section 2 pour savoir quels registres sont utilisés pour passer des paramètres entiers de 64 bits).

Intéressons-nous à présent au programme assembleur. Celui-ci comporte cette fois-ci la définition de 4 variables de types QWORD (64 bits) a, b, c, n servant à stocker temporairement les valeurs des paramètres reçus. Une cinquième variable temporaire tmp est également utilisée pour stocker des calculs temporaires. Les cinq variables ne sont pas initialisées, d'où le point d'interrogation.

Les premières instructions du programme copient les valeurs des paramètres dans les variables.

Le programme procède ensuite aux calculs nécessaires en utilisant l'instruction IMUL qui réalise une multiplication signée ainsi que l'instruction ADD pour l'addition. A la fin le résultat est retourné dans le registre RAX.

**Remarque** : il n'est pas possible de faire une addition ou une multiplication directement entre 2 variables.

## 5. Autres utilisations du débogueur

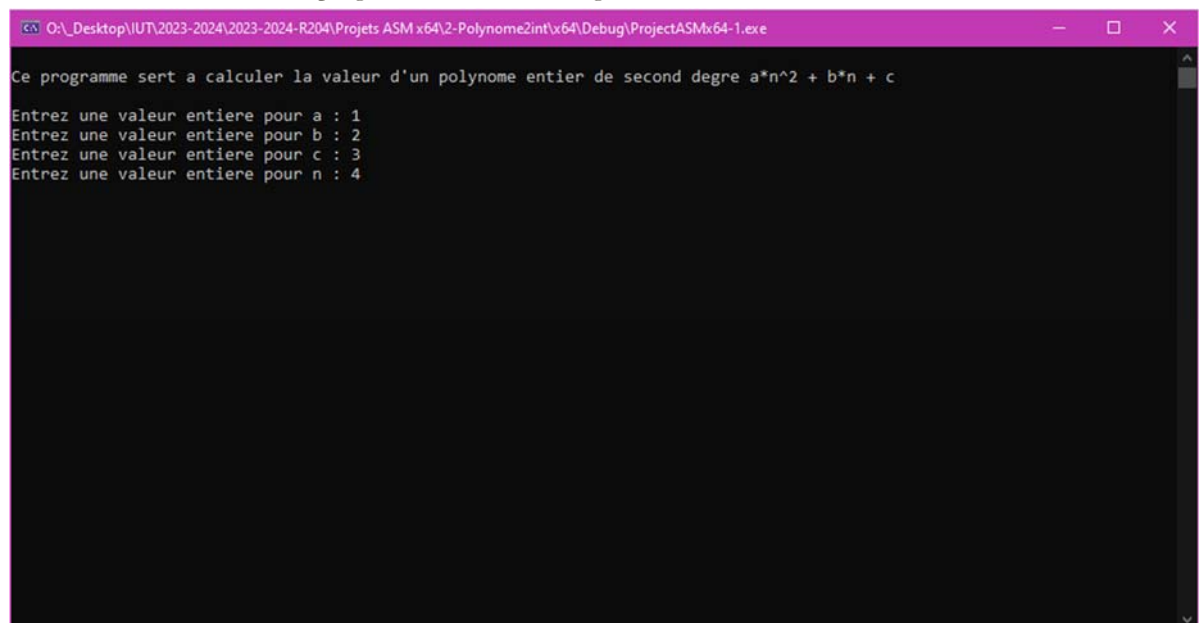
Le débogueur peut également être utilisé pour :

- Surveiller la valeur d'une variable
- Consulter le code machine des instructions du programme
- Consulter le contenu de la mémoire

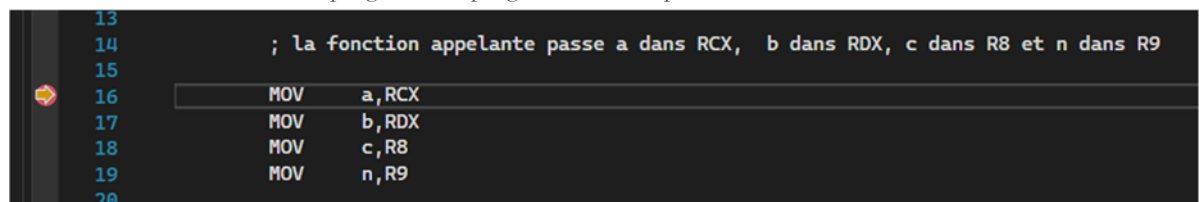
### Surveiller la valeur d'une variable

Commencez par ajouter un point d'arrêt sur la ligne MOV a,RCX (F9).

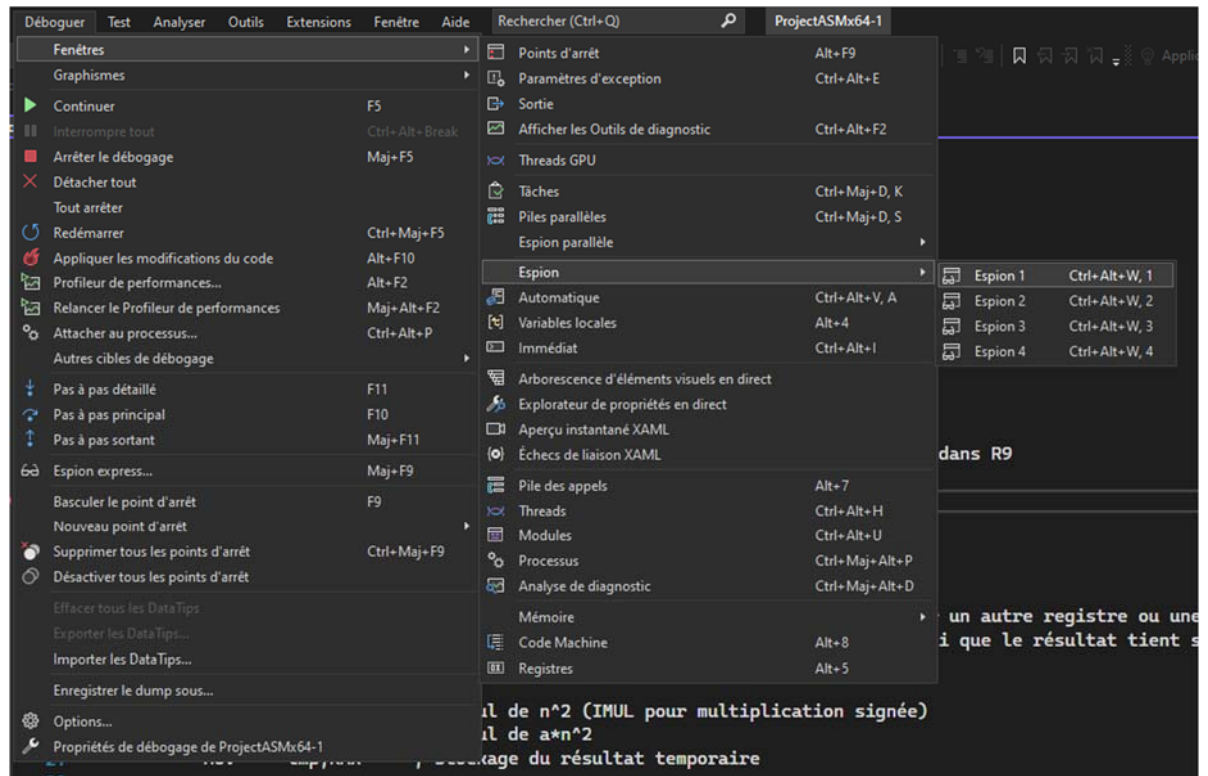
Ensuite lancez l'exécution du programme en mode debug. Saisissez 4 valeurs entières.



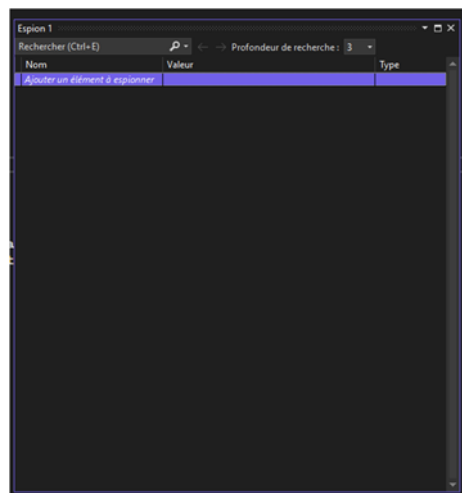
Revenez à la fenêtre du fichier prog.asm. Le programme est en pause avant d'exécuter l'instruction MOV a,RCX.



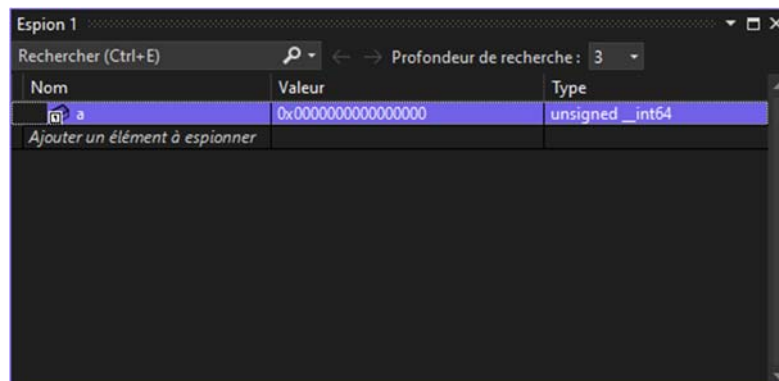
Allez dans le menu Déboguer → Fenêtres → Espion → Espion 1



Une nouvelle fenêtre **Espion 1** apparaît. Si elle est en bas à gauche la rendre flottante.

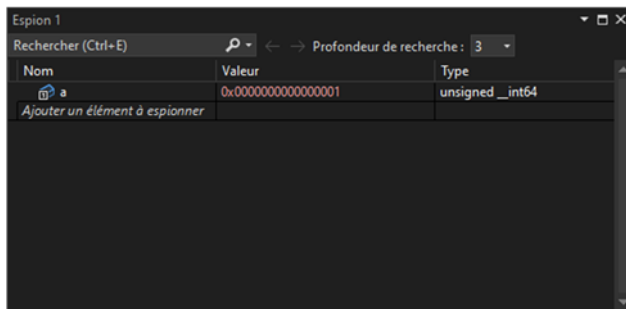


Cliquez sur « Ajouter un élément à espionner » et tapez le nom de la variable **a**.



Nous disposons à présent d'un affichage permanent de la valeur de la variable **a** (du programme ASM) et nous pouvons suivre son évolution au cours de l'exécution. Il est possible d'afficher sa valeur soit en hexa, soit en décimal (bouton droit sur la variable puis « affichage hexadécimal »).

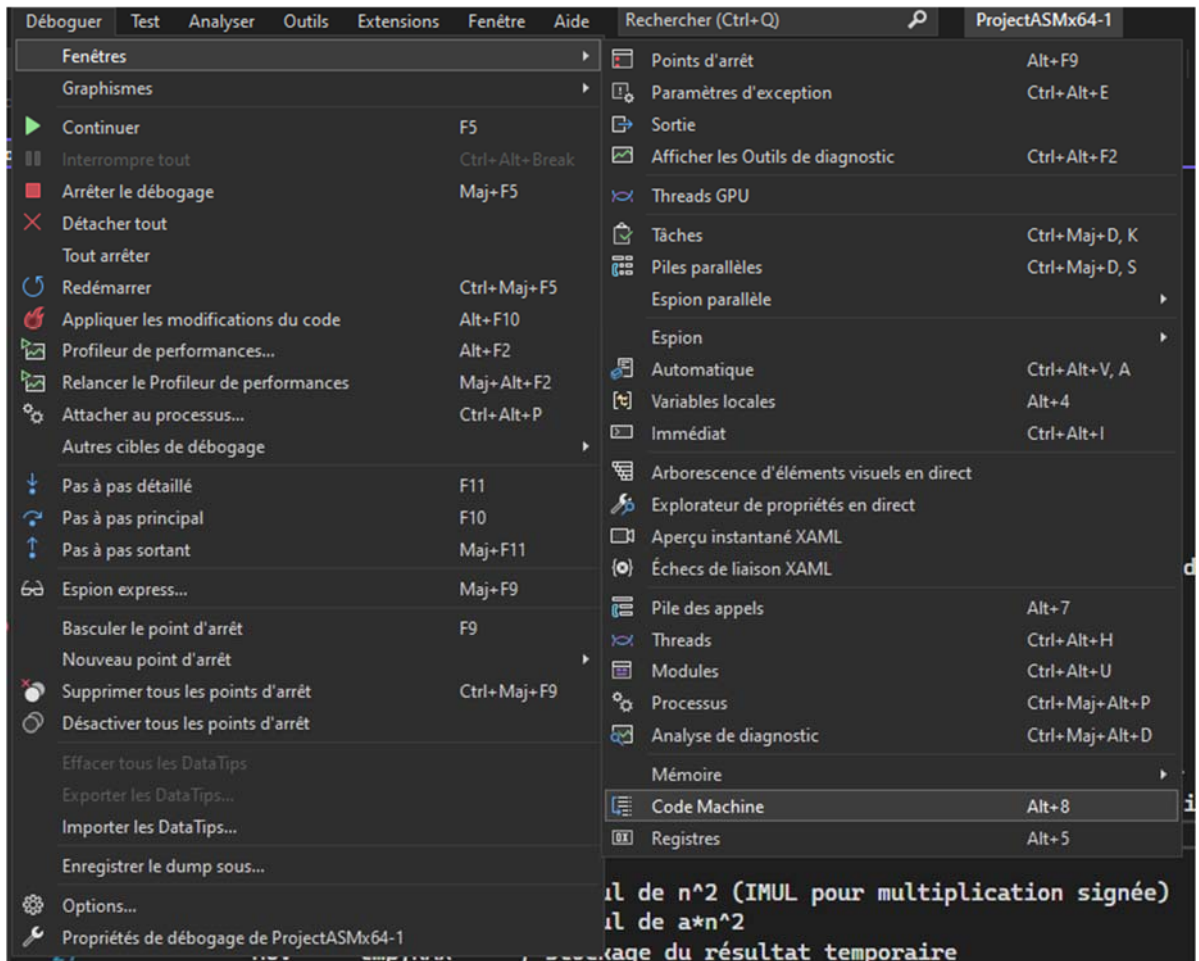
La valeur de **a**, n'ayant pas été initialisée, elle apparaît à 0. Appuyez à présent sur F11 pour exécuter l'instruction `MOV a,RCX` et observez la nouvelle valeur de la variable **a**.



Il est bien sûr possible ainsi de surveiller les valeurs d'autres variables en cliquant simplement sur « Ajouter un élément à espionner »

### Affichage du code machine des instructions

Pour afficher le code machine des instructions il faut aller dans le menu Déboguer → Fenêtres → Code Machine





Une nouvelle fenêtre contenant le code machine du programme s'affiche. Cliquez avec le bouton droit dans la fenêtre et choisissez la façon dont vous voulez afficher les choses :

Avec code source

```

--- O:\_Desktop\IUT\2023-2024\2023-2024-R204\Projets ASM x64\2-Polynome2int\prog.asm

; la fonction appelante passe a dans RCX, b dans RDX, c dans R8 et n dans R9

MOV     a,RCX
00007FF6B8112640 48 89 0D B9 B9 00 00 mov     qword ptr [a (07FF6B811E000h)],rcx
MOV     b,RDX
00007FF6B8112647 48 89 15 BA B9 00 00 mov     qword ptr [b (07FF6B811E008h)],rdx
MOV     c,R8
00007FF6B811264E 4C 89 05 BB B9 00 00 mov     qword ptr [c (07FF6B811E010h)],r8
MOV     n,R9
00007FF6B8112655 4C 89 0D BC B9 00 00 mov     qword ptr [n (07FF6B811E018h)],r9

; ici IMUL fait une multiplication entière signée de RAX (64 bits) par un autre registre ou une autre variable entière de 64 bits
; et stocke le résultat sur 128 bits dans RDX:RAX mais on supposera ici que le résultat tient sur 64 bits seulement (soit RAX)

MOV     RAX,n
00007FF6B811265C 48 8B 05 B5 B9 00 00 mov     rax,qword ptr [n (07FF6B811E018h)]
IMUL     n                ; calcul de n^2 (IMUL pour multiplication signée)
00007FF6B8112663 48 F7 2D AE B9 00 00 imul     qword ptr [n (07FF6B811E018h)]
IMUL     a                ; calcul de a*n^2
00007FF6B811266A 48 F7 2D 8F B9 00 00 imul     qword ptr [a (07FF6B811E000h)]
MOV     tmp,RAX           ; stockage du résultat temporaire
00007FF6B8112671 48 89 05 A8 B9 00 00 mov     qword ptr [tmp (07FF6B811E020h)],rax

MOV     RAX,n
00007FF6B8112678 48 8B 05 99 B9 00 00 mov     rax,qword ptr [n (07FF6B811E018h)]
IMUL     b                ; calcul de b*n
00007FF6B811267F 48 F7 2D 82 B9 00 00 imul     qword ptr [b (07FF6B811E008h)]

ADD     RAX,tmp           ; calcul de a*n^2 + b*n
00007FF6B8112686 48 03 05 93 B9 00 00 add     rax,qword ptr [tmp (07FF6B811E020h)]
ADD     RAX,c             ; calcul de a*n^2 + b*n + c
00007FF6B811268D 48 03 05 7C B9 00 00 add     rax,qword ptr [c (07FF6B811E010h)]

RET     ; le résultat à retourner est déjà dans RAX
00007FF6B8112694 C3                               ret

```

Sans code source

```

--- O:\_Desktop\IUT\2023-2024\2023-2024-R204\Projets ASM x64\2-Polynome2int\prog.asm
00007FF6B8112640 48 89 0D B9 B9 00 00 mov     qword ptr [a (07FF6B811E000h)],rcx
00007FF6B8112647 48 89 15 BA B9 00 00 mov     qword ptr [b (07FF6B811E008h)],rdx
00007FF6B811264E 4C 89 05 BB B9 00 00 mov     qword ptr [c (07FF6B811E010h)],r8
00007FF6B8112655 4C 89 0D BC B9 00 00 mov     qword ptr [n (07FF6B811E018h)],r9
00007FF6B811265C 48 8B 05 B5 B9 00 00 mov     rax,qword ptr [n (07FF6B811E018h)]
00007FF6B8112663 48 F7 2D AE B9 00 00 imul     qword ptr [n (07FF6B811E018h)]
00007FF6B811266A 48 F7 2D 8F B9 00 00 imul     qword ptr [a (07FF6B811E000h)]
00007FF6B8112671 48 89 05 A8 B9 00 00 mov     qword ptr [tmp (07FF6B811E020h)],rax
00007FF6B8112678 48 8B 05 99 B9 00 00 mov     rax,qword ptr [n (07FF6B811E018h)]
00007FF6B811267F 48 F7 2D 82 B9 00 00 imul     qword ptr [b (07FF6B811E008h)]
00007FF6B8112686 48 03 05 93 B9 00 00 add     rax,qword ptr [tmp (07FF6B811E020h)]
00007FF6B811268D 48 03 05 7C B9 00 00 add     rax,qword ptr [c (07FF6B811E010h)]
00007FF6B8112694 C3                               ret

```

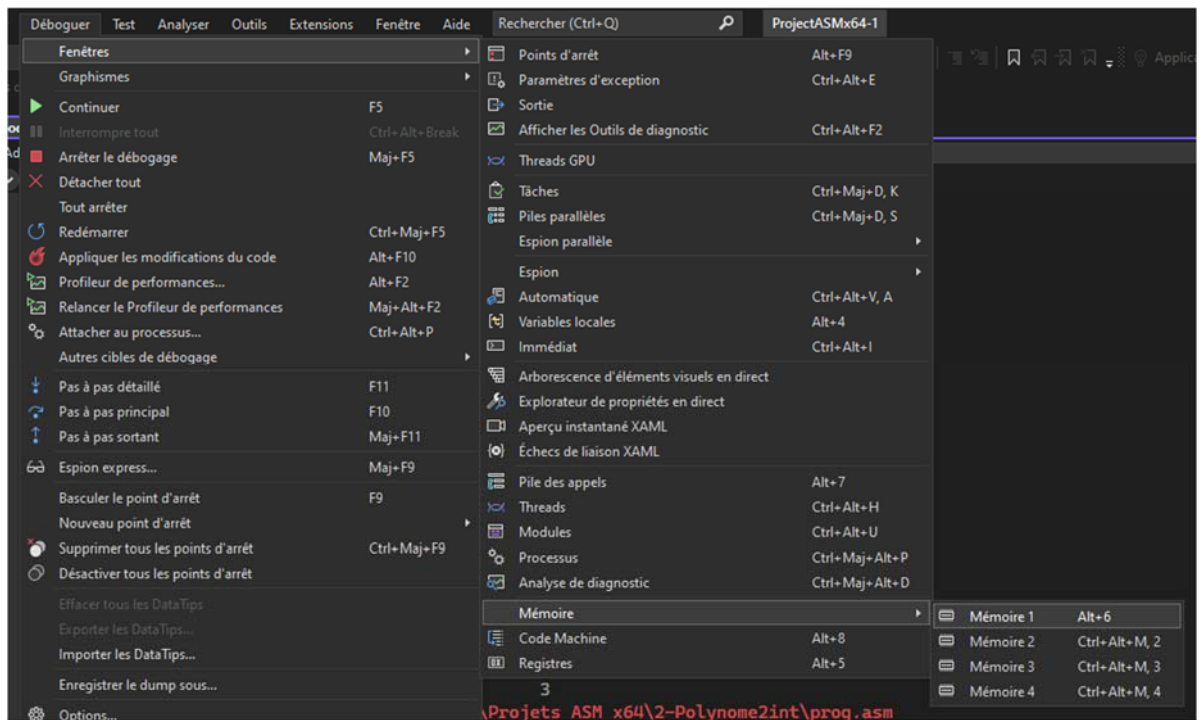
La première colonne représente l'adresse en mémoire de l'instruction puis viennent les codes de l'instruction.

## Affichage du contenu de la mémoire

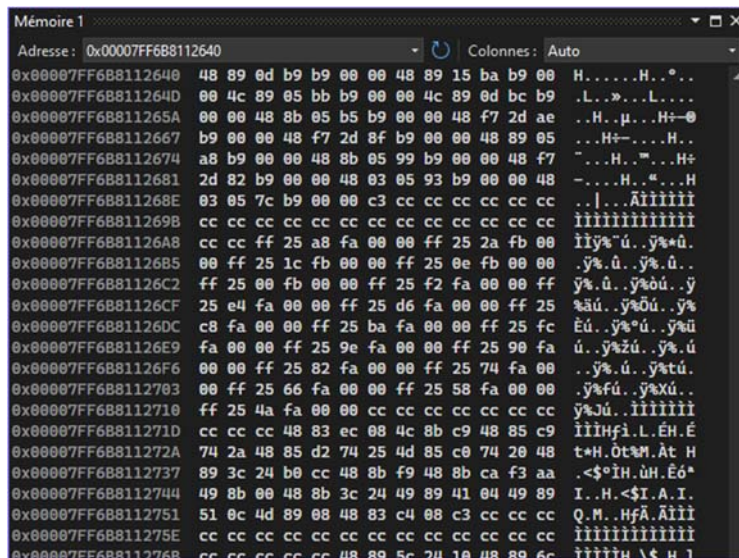
Dans la capture d'écran ci-dessus, nous pouvons voir dans la première instruction l'adresse de la variable a : 07FF6B811E000 en hexa.

Nous allons maintenant afficher le contenu de la mémoire à cette adresse. Pour cela allez dans le menu Débuguer → Fenêtres → Mémoire → Mémoire 1

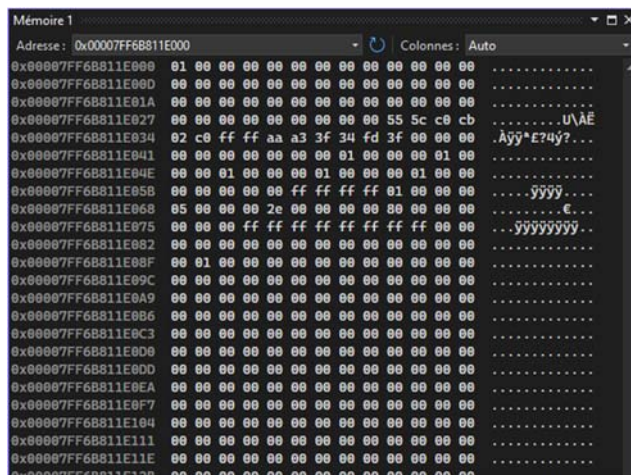




Une nouvelle fenêtre en bas à gauche apparaît. La rendre éventuellement flottante.

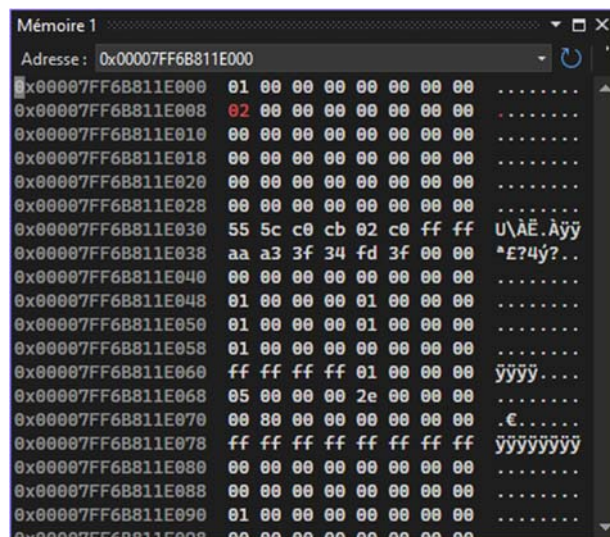


Tapez dans le champ Adresse l'adresse de la variable a : 0x07FF6B811E000



On voit que la valeur de a est bien 1.

Redimensionnez la fenêtre de façon à afficher 8 octets de mémoire par ligne puis appuyez sur F11 pour exécuter l'instruction suivante MOV b,RDX



Observez la nouvelle valeur de la variable b (2 en rouge). On voit bien que chaque variable occupe 8 octets (64 bits) et que c'est l'octet de poids faible qui est stocké en premier (les processeurs intel sont des processeurs little-endian).

## 6. A vous de jouer...

Modifiez le programme précédent de façon à calculer la valeur d'un polynôme du second degré  $a*x^2+b*x+c$  mais cette fois-ci avec des valeurs réelles a, b, c, x.

Voici quelques éléments à prendre en compte pour modifier le programme :

1. Dans le programme C++, utilisez le type double (8 octets = 64 bits). N'oubliez pas de modifier aussi le type de retour de la fonction.
2. L'équivalent du type double en assembleur est REAL8 (réel sur 8 octets).
3. Pour les opérations sur les réels, il faut utiliser les registres XMM0, XMM1, XMM2, XMM3
4. Le tableau ci-dessous montre à travers quels registres sont passés les paramètres lorsqu'il s'agit de réels :

Taille Position	double (64 bits)
1	XMM0
2	XMM1
3	XMM2
4	XMM3
Valeur de retour	XMM0

5. Pour les opérations sur des réels il faut utiliser les instructions suivantes :

- |    |       |                     |                                                                                             |
|----|-------|---------------------|---------------------------------------------------------------------------------------------|
| a. | MOVSD | variable , registre | (le registre doit être XMM0, ou XMM1, ou XMM2, etc.)                                        |
| b. | MOVSD | registre , variable | (le registre doit être XMM0, ou XMM1, ou XMM2, etc.)                                        |
| c. | MULSD | registre , variable | (le registre doit être XMM0, ou XMM1, ou XMM2, etc réalise registre * variable → registre.) |
| d. | ADDSD | registre , variable | (le registre doit être XMM0, ou XMM1, ou XMM2, etc réalise registre + variable → registre.) |

## 7. Factorielle

Ecrire un programme en assembleur qui calcule la factorielle d'un nombre entier  $n$  de 64 bits (long long int).  
Se référer au tableau de la section 2 pour savoir dans quels registres sont passés les paramètres entiers 64 bits.

## 8. PGCD

Ecrire un programme en assembleur permettant de calculer le **Plus Grand Commun Diviseur** de deux nombres entiers de 32 bits (int)  $a$  et  $b$ . Se référer au tableau de la section 2 pour savoir dans quels registres sont passés les paramètres entiers 32 bits.

On utilisera pour cela l'algorithme d'Euclide.

Exemple :  $a=36$  et  $b=48$

- Première division :  $36/48 \rightarrow \text{quotient}=0$  et  $\text{reste}=36$
- Permutations :  $a \leftarrow b$  et  $b \leftarrow \text{reste}$  (soit  $a=48$  et  $b=36$ )
- On reprend les divisions et les permutations jusqu'à ce que le reste soit nul.
- Deuxième division :  $48/36 \rightarrow \text{quotient}=1$  et  $\text{reste}=12$
- Permutations :  $a \leftarrow b$  et  $b \leftarrow \text{reste}$  (soit  $a=36$  et  $b=12$ )
- Troisième division :  $36/12 \rightarrow \text{quotient}=3$  et  $\text{reste}=0$
- Le reste est nul, on s'arrête. Le PGCD est le dernier diviseur soit **PGCD=12**

Pour réaliser les divisions, on utilisera la division 32 bits (c'est-à-dire que le diviseur sera sur 32 bits alors que le dividende sera sur 64 bits).

### RAPPEL

DIV ECX réalise une division de **EDX:EAX** par **ECX**. Le quotient sera dans **EAX** et le reste dans **EDX**.

## 9. PPCM

Ecrire un programme en assembleur qui calcule le **Plus Petit Commun Multiple** de deux nombres entiers  $m$  et  $n$  de 16 bits (short int) en vous inspirant de la fonction ci-dessous.

Se référer au tableau de la section 2 pour savoir dans quels registres sont passés les paramètres entiers 16 bits.

```
int ppcm(const short int m, const short int n)
{
    int a = m, b = n;
    while (a != b)
    {
        while (a < b) a = a + m;
        while (a > b) b = b + n;
    }
    return a;
}
```