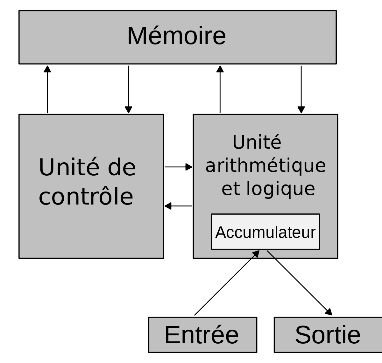
**Découverte du langage Machine**

**Capacité attendue :** Dérouler l’exécution d’une séquence d’instructions simples du type langage machine

# Introduction au langage d’assemblage

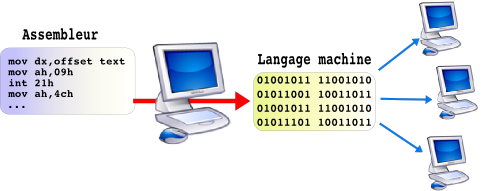
Tu utilises le langage Python qui te permet de programmer des applications, et ainsi d’utiliser le processeur d’un système.

Tu as vu l’architecture matérielle de type Von Neumann.

[](https://upload.wikimedia.org/wikipedia/commons/thumb/0/01/Von_Neumann_architecture_fr.svg/1200px-Von_Neumann_architecture_fr.svg.png)

Le langage Python permet de dialoguer facilement entre une personne et le processeur à l’aide d’un langage assez proche de nous. Ce langage est très proche de notre façon de voir les choses.

Tu vas te rapprocher maintenant de la machine et découvrir un langage encore plus proche de la machine, le langage d’assemblage. En fait tu as vu que la machine n’utilise que des séries de bits. Tu vas voir comment ces séries de bits sont créées, et interprétées par le processeur.

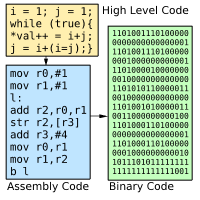
[](https://info.sio2.be/python/1/images/assembleur.jpg)

Chaque processeur possède son propre langage assemblage.

Tu vas utiliser pour cela la carte RaspberryPi 3 qui est basée sur un processeur ARM8 Quad Core 1.2GHz Broadcom BCM2837 64bit

# Des informations compréhensibles par la machine

Il y a plusieurs étapes entre un langage de haut niveau, comme ici le langage C et la machine.



Saisie de son code source dans un langage donné

Editeur de texte(IDE)

Transforme le code source en un langage d’assemblage compréhensible par la machine

Compilateur

Fichier langage assemblage

file.s

Intègre toutes les fonctions de votre programme, y compris les bibliothèques de fonctions accessibles en externe, et détermine où chaque composant sera chargé en mémoire lors de l'exécution du programme.

Editeur de liens

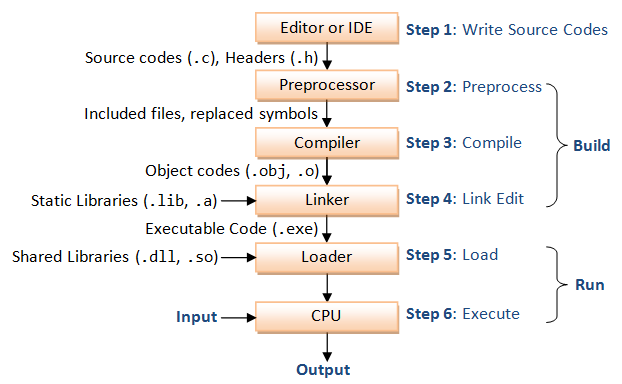
Fichier objet

file.o

Charge la version du code machine du programme dans la mémoire où la CPU peut l'exécuter

Loader

Fichier exécutable



# Langage d’assemblage

Le langage d’assemblage utilise des codes, en général, avec des commandes plus réduites qu’un langage de haut niveau.

Une instruction en langage d’assemblage se présente en générale de la façon suivante :

Mnémonique opérande1, opérande2

On peut distinguer des instructions :

* Transfert : mov, ldr…
* Opérations arithmétiques et logiques : add, sub, not…
* Sauts : jmp, b, beq…

# Utilisation de registres

Tu vas utiliser des registres qui permettent de mémoriser des données.

Dans les ordinateurs, les données ne sont pas exploitées dans la mémoire ; au lieu de cela, ces données sont chargées dans un registre du CPU, puis le traitement des données ou l’opération arithmétique est effectuée dans un registre. Les registres font partie des circuits du processeur permettant un accès instantané ; alors que la mémoire est un composant séparé et que le CPU a un temps de transfert pour y accéder.

Un registre est une petite partie de la mémoire du processeur, dans laquelle le processeur stocke les données sur lesquelles il travaille. Il en existe plusieurs, dont beaucoup ont une utilisation particulière et dédiés à une fonction spécifique. Il existe différents registres (nommés r0, r1…, r12) qui sont appelés "utilisation générale" et que tu peux utiliser pour tous les calculs que tu dois effectuer.

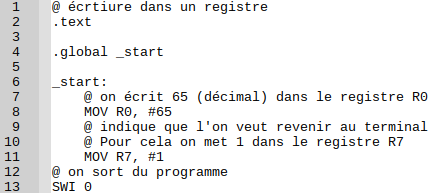
# Mon premier programme en langage d’assemblage

## Afficher un nombre

Tu vas créer un programme très simple qui permet de stocker un nombre dans un registre et d’afficher sur le terminal ce nombre.

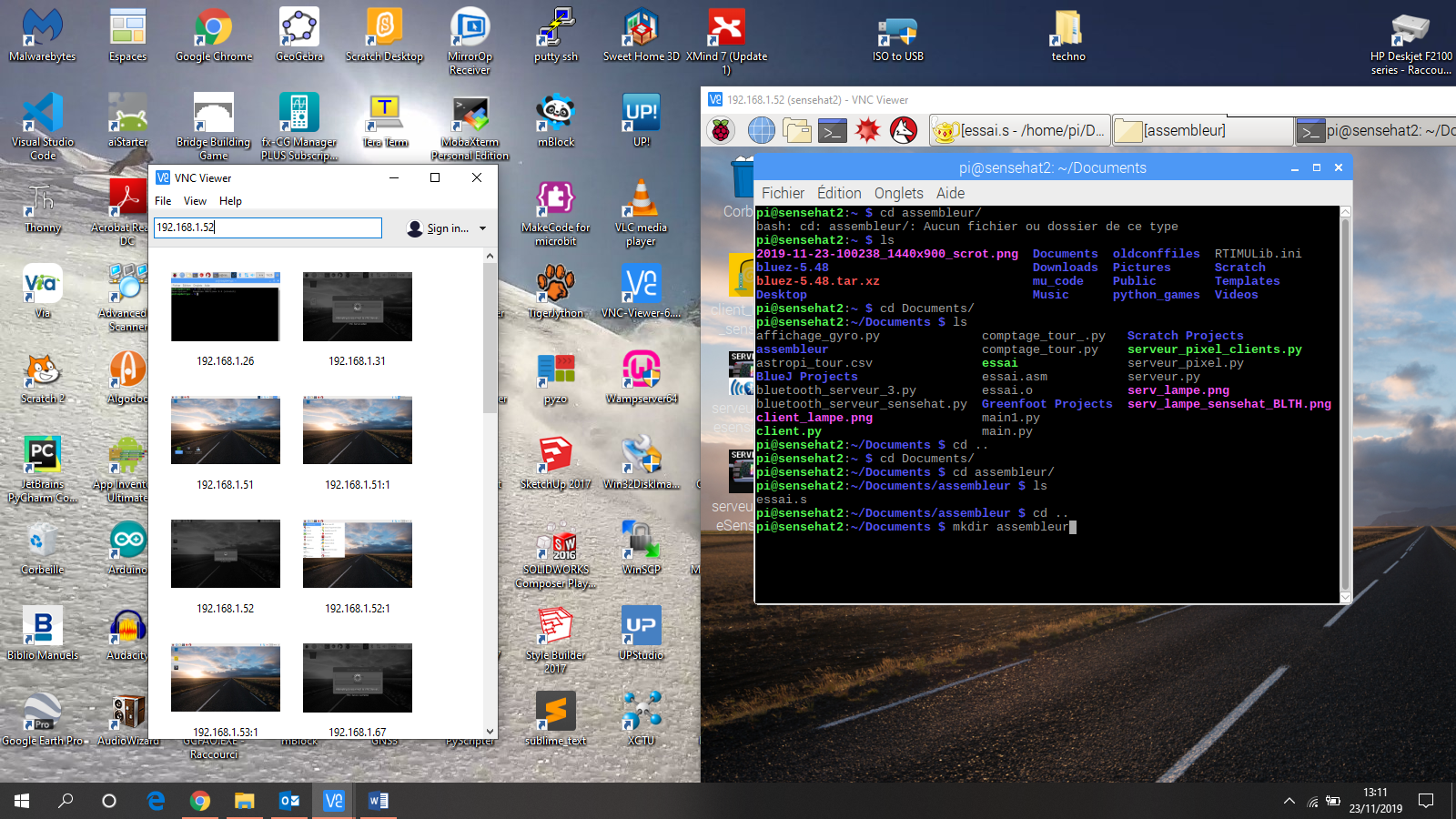
Lance l’IDE jeany sur la raspberryPi.

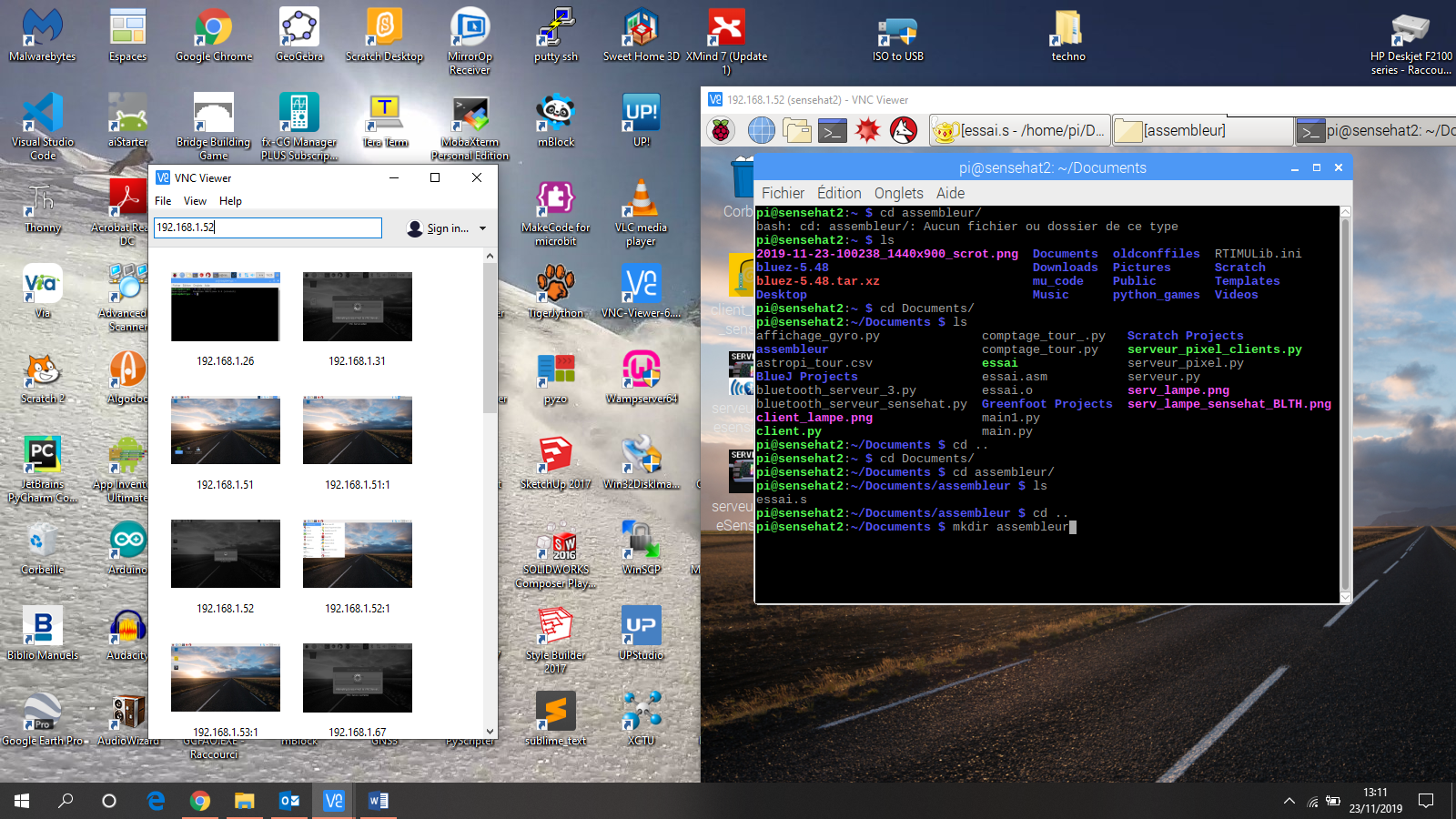
Saisie le programme suivant :



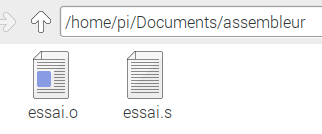
Tu vas créer maintenant le fichier objet à partir de ton fichier dans un langage d’assemblage.

Tu vas utiliser pour cela un linker.



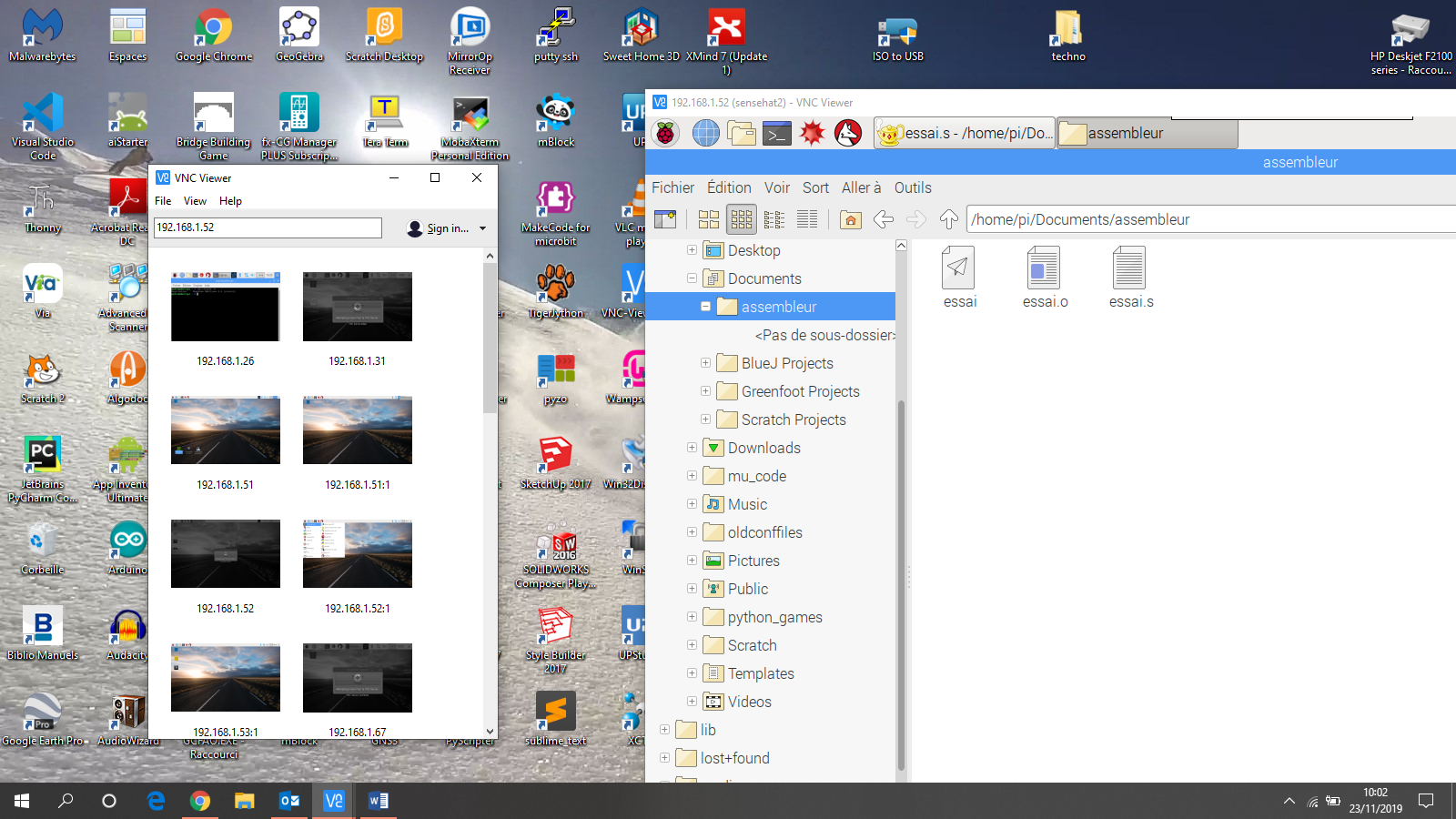






Puis tu vas créer le fichier exécutable pour qu’il soit compréhensible directement par la machine.





Il suffit maintenant de lancer le programme exécutable.



Tu obtiens bien l’affichage de la valeur 65 de départ.

## Réaliser une soustraction

Réalisation d’une soustraction et affichage du résultat dans le terminal.

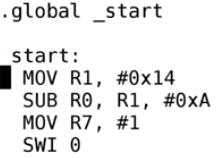
Tu vas utiliser un registre pour mémoriser le résultat

Registre résultat = terme1 – terme2

Pour cela utiliser le mnémonique **sub** <registre résultat>, <terme 1>, <terme 2>

Réalise alors ton programme et teste-le en utilisant l’exemple précédent.

Elément de correction.

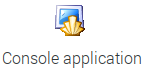


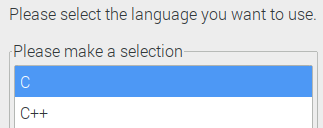
## Débogage

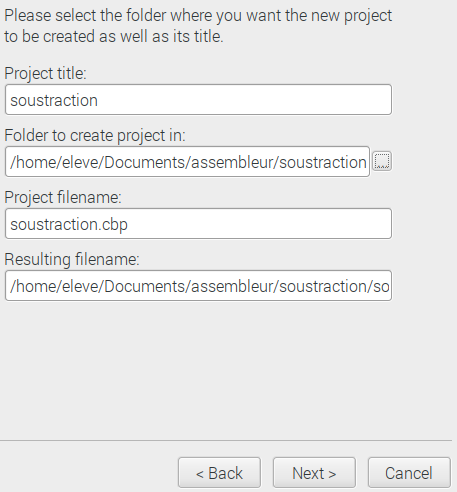
Tu vas maintenant tester cette application de soustraction en mode pas à pas pour visualiser le contenu des registres.

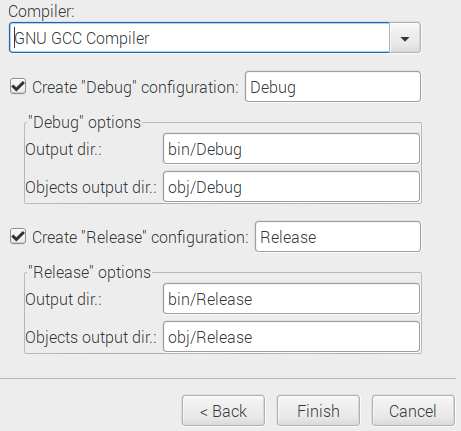
Lance la logiciel Codeblock

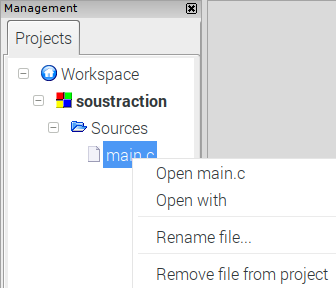




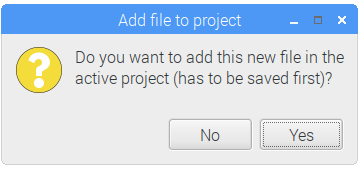






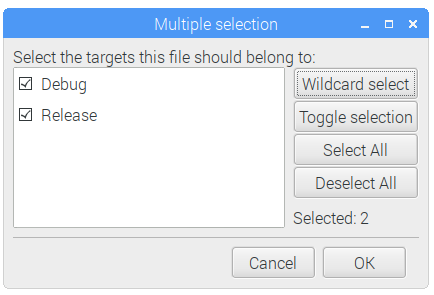


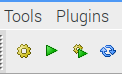


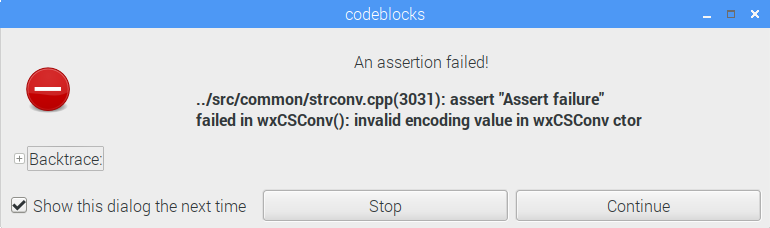






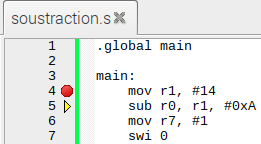




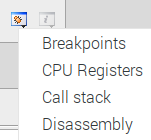


Saisie le programme d’assemblage suivant qui permet de réaliser une soustraction.

Clique entre le chiffre et la ligne pour ajouter un point d’arrêt.



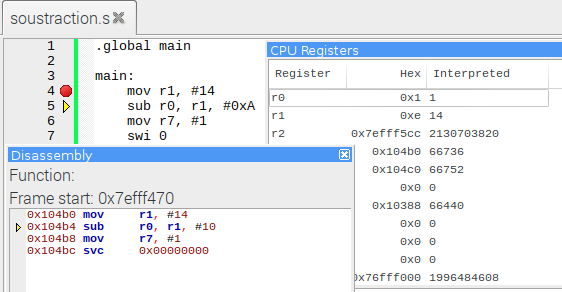
Sélectionne les visualisations des registres et du désassemblage.



Lance le mode pas à pas.

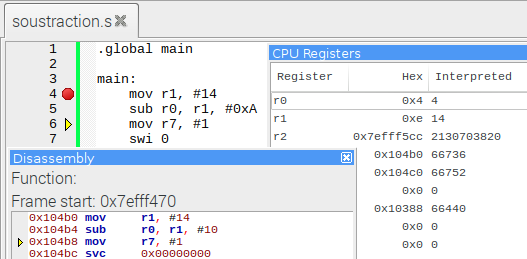


Tu visualises le contenu des registres.



Puis clique sur next instruction





On obtient bien 14-10=4

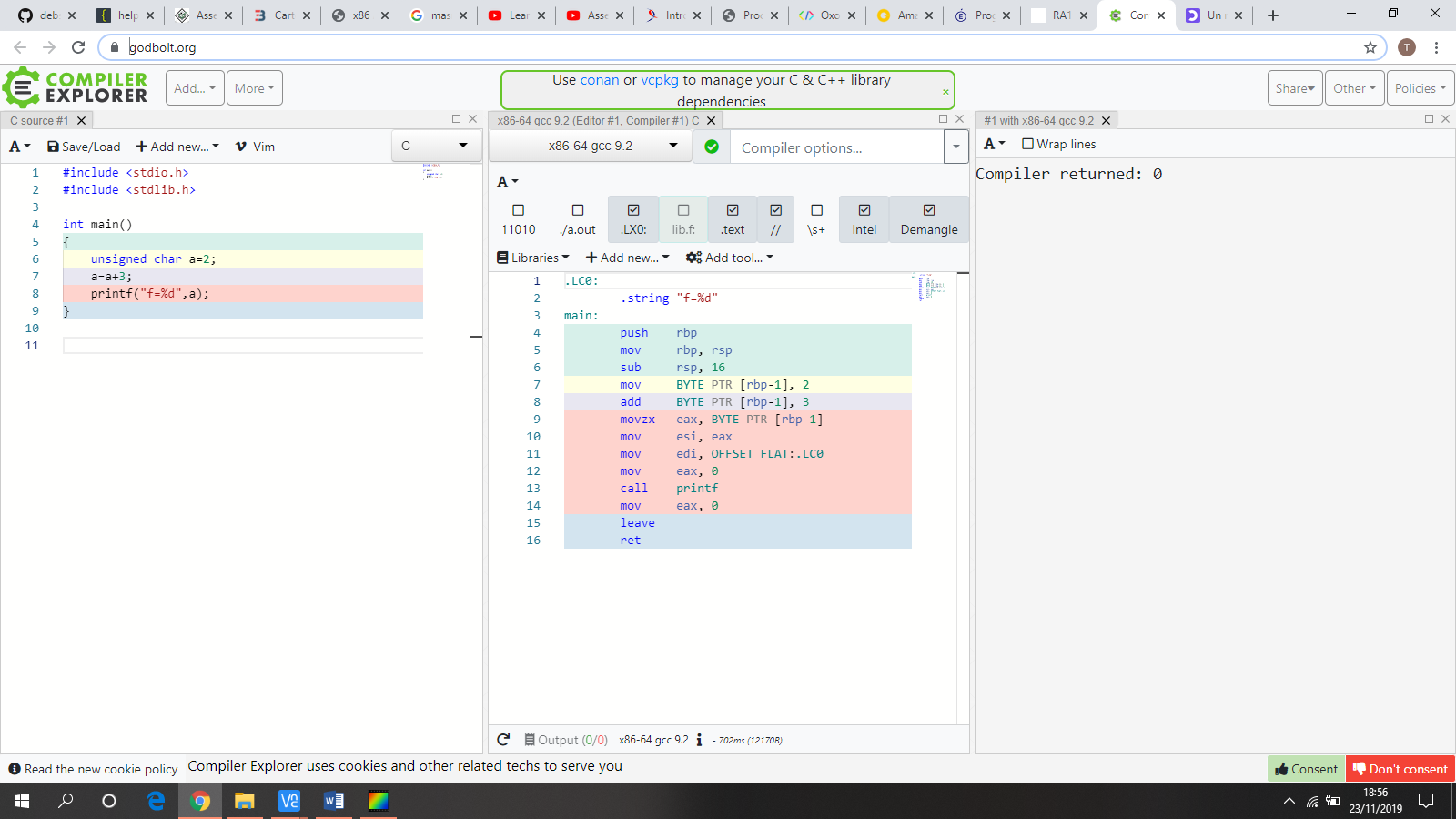
# Visualisation des étapes

Tu vas réaliser un débogage de ton programme en langage d’assemblage, et visualiser ce qui se passe à chaque étape.

Tu vas visualiser la traduction en langage d’assemblage d’un langage de haut niveau, comme le langage C

Lance l’application sur le site . http://godbolt.com

Saisie le programme suivant en C



Tu peux alors visualiser la traduction d’un langage de haut niveau en langage d’assemblage.

Pour le langage Python, on n’utilise pas directement la méthode vue. En effet, ce langage utilise un interpréteur qui permet de « lire » chaque commande Python, et de l’exécuter.

## Réaliser une addition

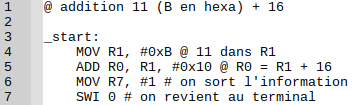
Tu vas réaliser une simple addition entre deux valeurs et afficher le résultat dans le terminal.

Pour cela tu vas utiliser le mnémonique :

ADD <registre résultat>, <terme 1>, <terme 2>

Registre résultat = terme1 + terme 2

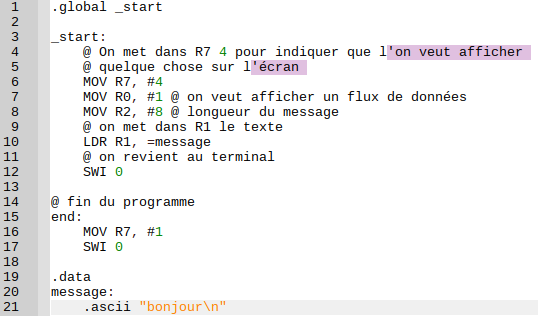
Elément de correction :





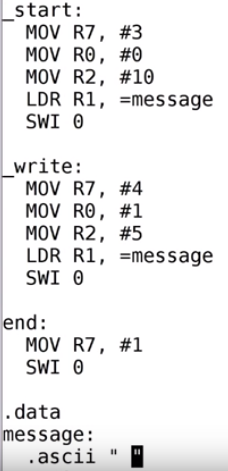
## Afficher un texte

Tu vas créer une application qui permet d’afficher un texte sur le terminal.





## Saisie d’une touche clavier

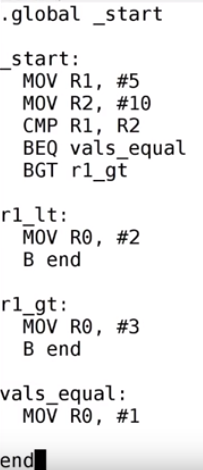


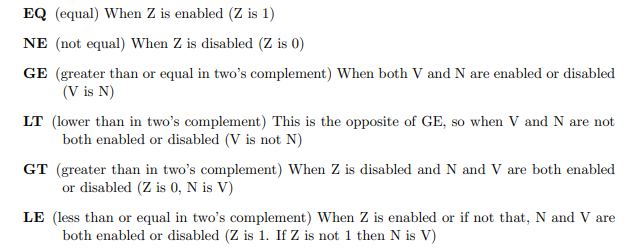


## Comment faire une multiplication



## Comment faire un test de condition.





* EQ (**eq**ual) When Z is enabled (Z is 1)
* NE (**n**ot **e**qual). When Z is disabled. (Z is 0)
* GE (**g**reater or **e**qual than, in two’s complement). When both V and N are enabled or disabled (V is N)
* LT (**l**ower **t**han, in two’s complement). This is the opposite of GE, so when V and N are not both enabled or disabled (V is not N)
* GT (**g**reather **t**han, in two’s complement). When Z is disabled and N and V are both enabled or disabled (Z is 0, N is V)
* LE (**l**ower or **e**qual than, in two’s complement). When Z is enabled or if not that, N and V are both enabled or disabled (Z is 1. If Z is not 1 then N is V)

# Algorithme

Recherche l’algorithme associé à ce programme

.**global** main

main:

**mov** r1, #2 /\* r1 ← 2 \*/

**mov** r2, #2 /\* r2 ← 2 \*/

**cmp** r1, r2 /\* update cpsr condition codes with the value of r1-r2 \*/

beq case\_equal /\* branch to case\_equal only if Z = 1 \*/

case\_different :

**mov** r0, #2 /\* r0 ← 2 \*/

b end /\* branch to end \*/

case\_equal:

**mov** r0, #1 /\* r0 ← 1 \*/

end:

**bx** lr

**text**

.**global** main

main:

**mov** r1, #0 /\* r1 ← 0 \*/

**mov** r2, #1 /\* r2 ← 1 \*/

**loop**:

**cmp** r2, #22 /\* compare r2 **and** 22 \*/

bgt end /\* branch if r2 > 22 to end \*/

**add** r1, r1, r2 /\* r1 ← r1 + r2 \*/

**add** r2, r2, #1 /\* r2 ← r2 + 1 \*/

b **loop**

end:

**mov** r0, r1 /\* r0 ← r1 \*/

**bx** lr

