**Rapport de TP2**

Opérations arithmétiques/logiques – Mixage d’ASM/C

**Architecture des ordinateurs**

**Groupe A – Mercredi 16h30/19h45**

17/02/2021

RIGHI Racim

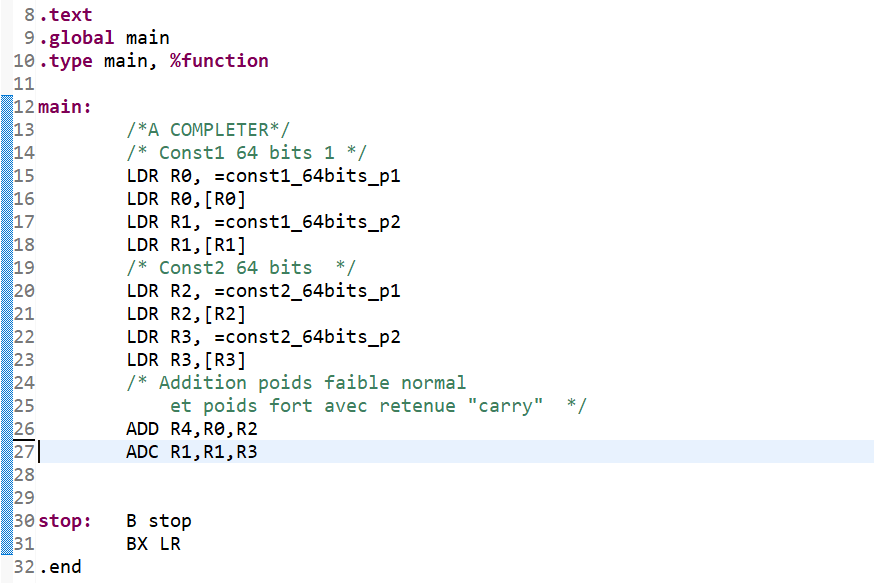
# **Exercice** 1 **:**

## Représentation numérique des données

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Op1 | Op2 | Type d'opération | Résultat | N | Z | C | V |
| 0x08000000 | 0x07000000 | + | 0xF0000000 | 0 | 0 | 0 | 1 |
| - | 0x10000000 | 0 | 0 | 1 | 0 |
| 0x40000000 | 0x40000000 | + | 0x80000000 | 1 | 0 | 0 | 1 |
| 0x40000000 | 0x80000000 | - | 0xC0000000 | 1 | 0 | 0 | 1 |
| 0x00F00000 | 0xFFFFFFFF | + | 0xEFFFFFFF | 0 | 0 | 1 | 0 |
| 0x7F000000 | 0x0F000000 | + | 0x8E000000 | 1 | 0 | 0 | 1 |
| - | 0x70000000 | 0 | 0 | 1 | 0 |
| 0x0F000000 | 0x7F000000 | + | 0x8E000000 | 1 | 0 | 0 | 1 |
| - | 0x90000000 | 1 | 0 | 0 | 0 |

## Addition d’entiers longs sur 64 bits

Pour faire la somme d’entiers 64 bits, il suffit de faire la somme normale des bits de poids faible, ensuite la somme avec retenue des bits de poids fort.



## Masquage

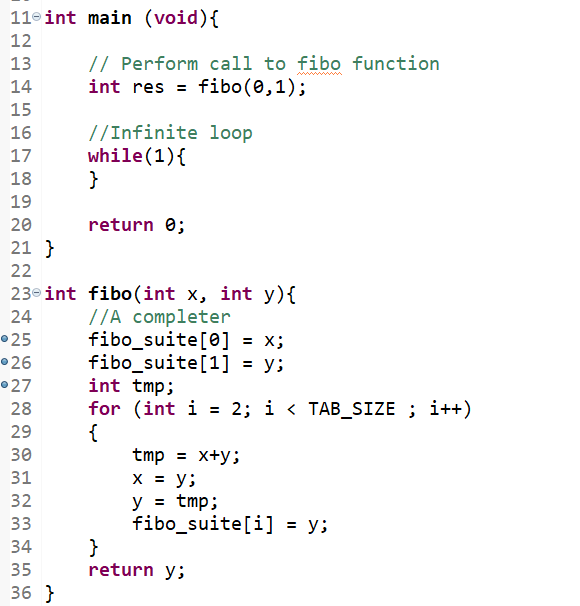
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **R0** | **R1** | **R2** | **Justification** |
| **MOV R0, #0x3A** | **0x3A** | ? | ? | / |
| **MOV R1, #0x0F** | 0x3A | **0x0F** | ? | / |
| **MOV R2, #0x10** | 0x3A | 0x0F | **0x10** | / |
| **AND R1, R0, R1** | 0x3A | **0x0A** | 0x10 | 0011 1010 AND 0000 1111 = 0000 1010 |
| **AND R0, R0, R2** | **0x10** | 0x0A | 0x10 | 0011 1010 AND 0001 0000 = 0001 0000 |
| **ORR R0, R1, R0** | **0x1A** | 0x0A | 0x10 | 0000 1010 AND 0001 0000 = 0001 1010 |
| **BIC R0, R0, R2** | **0x0A** | 0x0A | 0x10 | Les bits à 1 de R2 seront mis à 0 dans R0  0001 1010 BIC 0001 0000 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **R0** | **R1** | **R2** | **R3** | **R4** | **Justification** |
| **LDR R0, =VAL** | **@val** | **?** | **?** | **?** | **?** | **/** |
| **LDR R0, [R0]** | **0x87654321** | ? | ? | ? | ? | **/** |
| **MOV R1, #0xFF** | 0x87654321 | **0xFF** | ? | ? | ? | **/** |
| **LSL R2, R1, #4** | 0x87654321 | 0xFF | **0xFF0** | ? | ? | **Décalage de R1 de 4 bits vers la gauche** |
| **ASR R3, R2, #2** | 0x87654321 | 0xFF | 0xFF0 | **0x3FC** | ? | **Décalage de R2 de 2 bits vers la droite** |
| **ASR R4, R0, #2** | 0x87654321 | 0xFF | 0xFF0 | 0x3FC | **0xE1D950C8** | **1000 0111 0110 0101 0100 0011 0010 0001**  **Devient 1110 0001 1101 1001 0101 0000 1100 1000** |
| **LSR R4, R1, #1** | 0x87654321 | 0xFF | 0xFF0 | 0x3FC | **0x7f** | **Décalage de R4 de 1 bit vers la droite** |
| **EOR R2, R4, R2** | 0x87654321 | 0xFF | **0xF8F** | 0x3FC | 0x7f | **Ou exclusif entre R2 et R4** |
| **BIC R4, R4, R1** | 0x87654321 | 0xFF | 0xF8F | 0x3FC | **0x0** | **Les bits à 1 de R1 sont mis à 0 dans R4** |

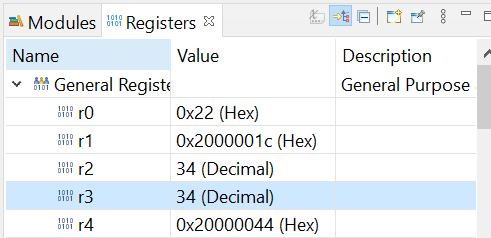
* Exercice 2 :

## Suite de Fibonacci C

On effectue le calcul de la suite de Fibonacci, on commençant avec 2 entiers de départ x et y, et en mettant les résultats de la suite dans un tableau de 10 éléments, on s’arrête par conséquent au calcul de U9.

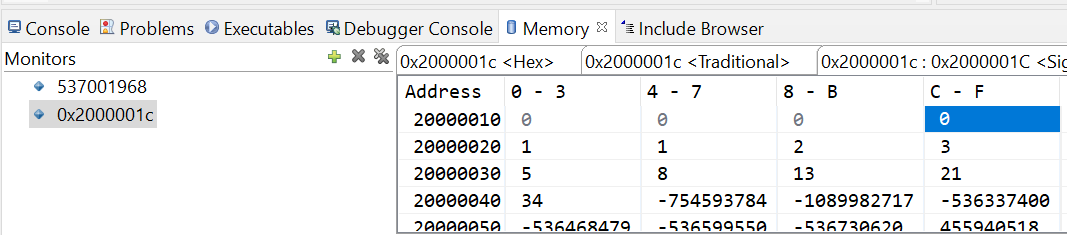
Code source de la fonction de calcul en C de Fibonacci

On remarque dans le registre R2 le résultat de la suite en commençant par 0 et 1, qui est 34.



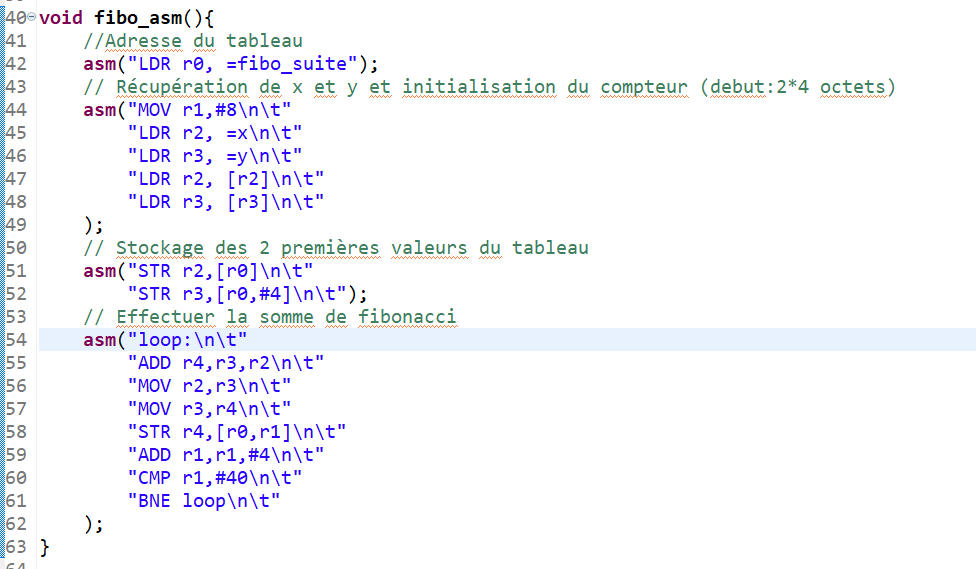
Contenu des registres à la fin de l’exécution

Et le contenu du tableau dans la zone mémoire qu’on trouve en récupérant l’adresse du registre R1, on remarque bien les 10 valeurs de 0 à 34.



Contenu du tableau dans la mémoire

## Mixer assembleur et C

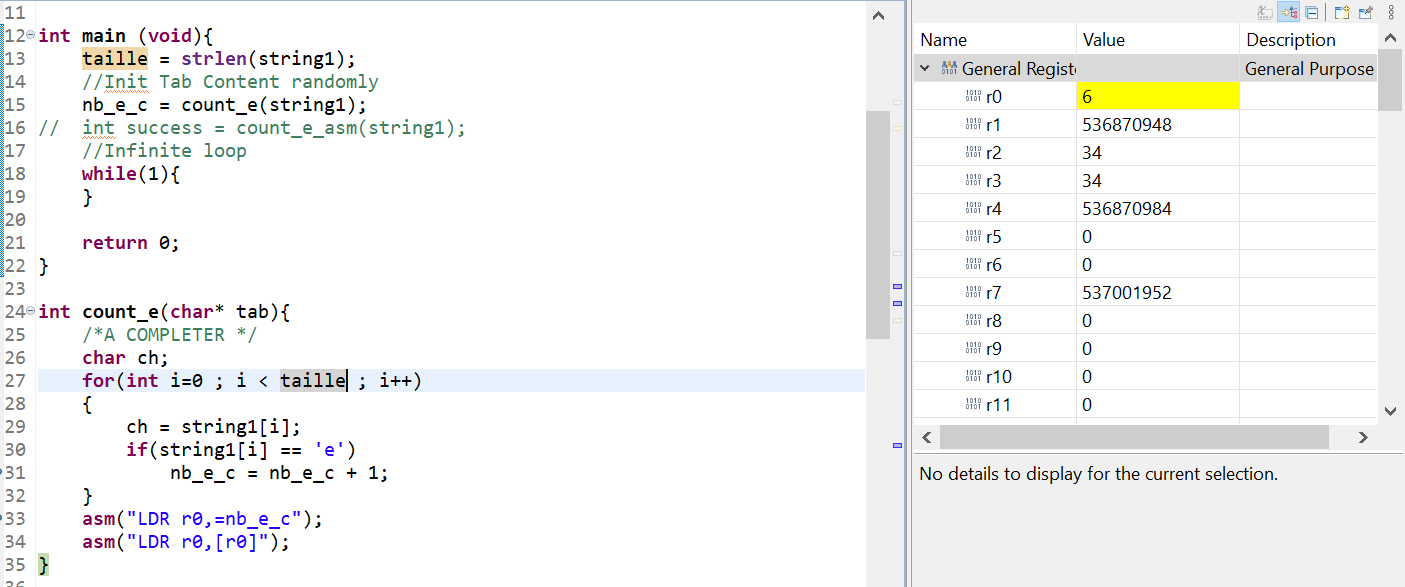
Pour pouvoir utiliser les variables x et y en assembleur, on les déclare comme variables globales

Code source de la solution assembleur de fibonacci

**Remarque :** On a aussi la possibilité de faire la boucle en C pour faciliter la solution, de cette manière on aurait qu’à récupérer la valeur de i à chaque itération et éviter tout les branchements conditionnels en assembleur.

* Exercice 3 :

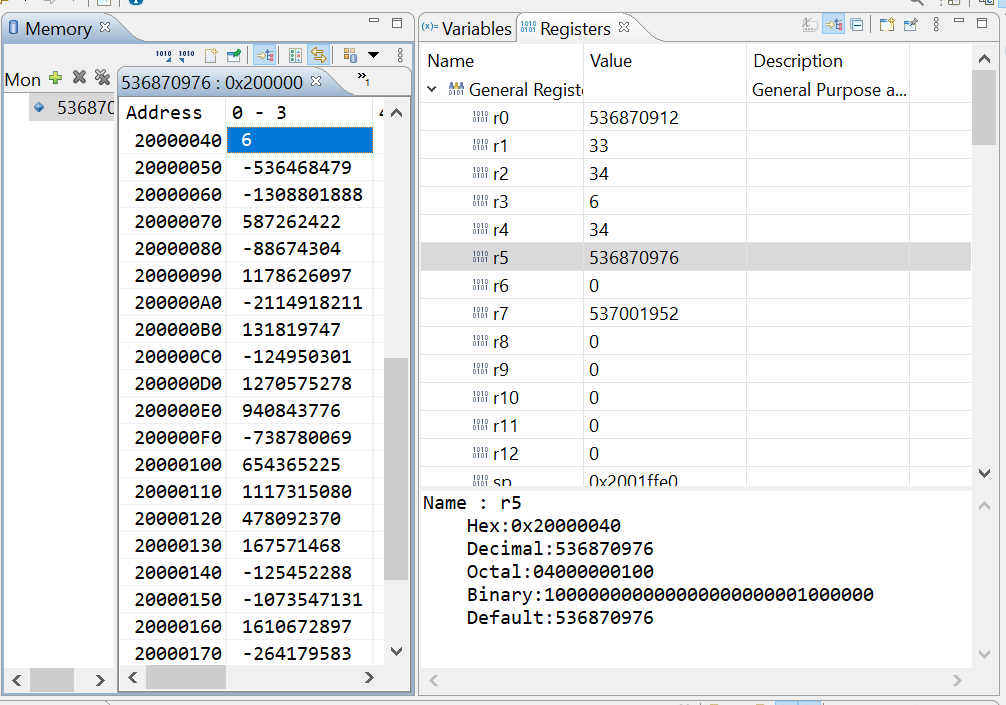
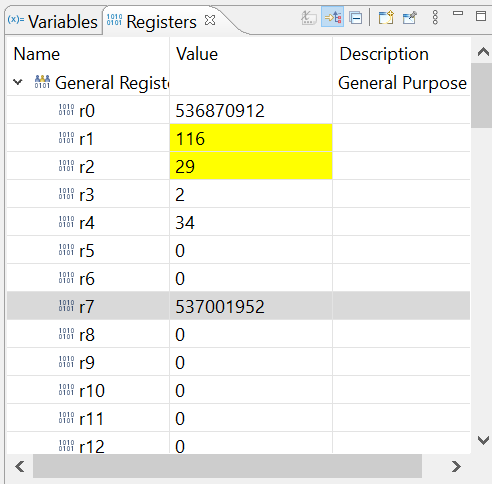
## En C

La fonction de calcul est relativement simple, on parcours la chaine de caractères et on incrémente un compteur à chaque ‘e’, on peut utiliser la fonction strlen de string.h pour récupérer la taille de la chaine. A la fin on peut voir le résultat en récupérant l’adresse de la variable ‘nb\_e\_c’ et mettre son contenu dans un registre.

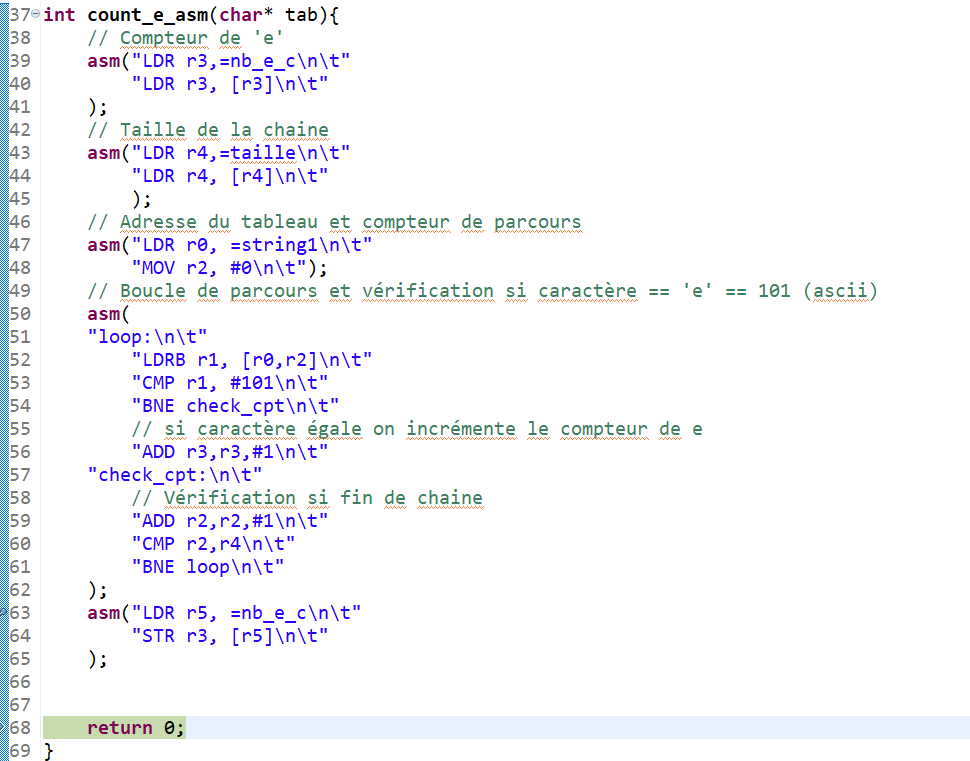
Fonction de calcul du nombre de ‘e’ et résultats

## En assembleur

On peut voir sur la capture d’écran suivante le contenu des registres à une étape intermédiaire, où le compteur est au caractère 29 qui a pour code ascii 116, donc ‘t’ dans « Universiteeee! ». Le nombre de ‘e’ actuel étant de 2

  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Résultat intermédiaire (caractères 29)

Résultat final de la mémoire et des registres

Code source de la solution assembleur

* Conclusion

Dépôt github de toutes les solutions :

https://github.com/RacimRgh/TP-Archi-STM32