ENSEA

Beyond Engineering

MICROPROCESSEURS

Compte-rendu TP1 STARTUP

François-Gabriel Sourbé

1 Goals and means

L'objectif principal de ce TP est de comprendre les mécanismes du microprocesseur, notamment :

- La phase de démarrage et l'exécution des premières instructions.
- L'utilisation de la pile pour le stockage des données.
- Les mécanismes d'appel de fonctions

2 Code : reset (or startup)

Requested PREPARATION 1:

Lors du démarrage, le processeur consulte le « Vector Table Offset Register » qui donne l'adresse de la table des vecteurs qu'on appellera `offset`. En accédant à cette adresse, on obtient en premier lieu l'adresse de la pile. Puisqu'il s'agit d'une adresse, elle est codée sur 32 bits soit 4 octets. L'élément suivant est donc enregistré à l'adresse `offset + 4` : c'est là qu'est stockée l'adresse de la première instruction à exécuter.

Pour l'obtenir via STM32CubeIDE, il faut démarrer une session de debug sur la carte, ouvrir la vue « SFR » (Special Function Register) pour obtenir la valeur contenue dans VTOR. Ensuite on entre l'adresse ainsi obtenue dans la vue « Memory » et on regarde le deuxième mot de 32 bits : c'est l'adresse de la première instruction. On peut vérifier cela en entrant cette adresse dans la vue « Disassembly » (on est censé trouver le code du `Reset_Handler`)

6 Functions: how to write functions in ASM language

Requested PREPARATION
2: Software installation & 3: Creation of the project

On installe la version 1.19.0 du logiciel STM32CubeIDE et on utilise l'option « créer un projet à partir d'un fichier .ioc existant » pour créer le projet avec le fichier `.ioc` fournit

Requested PREPARATION 4: ROM and RAM

La ROM et la RAM du microcontrôleur STM32L476RG sont respectivement de 1Mo et 128Kio. Elles débutent respectivement à l'adresse `0x08000000` et `0x02000000`.

L'adresse `0x20018000` correspond à la dernière adresse du premier bloc de RAM qui est la SRAM1 et peut contenir 96Kio. En effet la Pile est ici descendante (lorsque les éléments sont ajoutés dans la pile celle-ci progresse vers les adresses inférieures à celle de départ), et comme les variables globales sont quant à elles initialisées au début de la RAM, cela permet d'assurer le plus de place possible pour que la pile puisse grandir sans risquer de déborder sur ces variables globales.

7 Startup code or « Where does it start? »

Requested work 5: Insertion of your startup code

On peut ajouter sans problème le fichier `.s` dans le dossier « Startup » du projet grâce à un glisser-déposer dans l'explorateur de fichier intégré à CubeIDE puis en validant la copie.

8 Basic uses of the Stack

8.1 Basic use of the Stack : STACKA

Requested work 6: Study of STACKA

```
//////////////////////
    .global STACKA
    .text
STACKA:
    MOV R4,#234
    MOV R5,#235
    MOV R6,#236

    PUSH {R4}
    PUSH {R5}
    PUSH {R6}

    MOV R4,#0
    MOV R5,#0
    MOV R6,#0

    POP {R4}
    POP {R5}
    POP {R6}

END_STACKA: B END_STACKA
//////////////////////
```

Fig 1 – Code de STACKA

On complète le graphique fournit dans le sujet pour visualiser comprendre l'évolution de la Pile :

Initial State before first PUSH	R4 R5 R6	0x 000000EA 0x 000000EB 0x 000000EC	SP	0x2**001 8000**	0x20017FF4> 0x20017FF8> 0x20017FFC> 0x20018000>	0x00000000 0x00000000 0x00000000 0x00000000
After **PUSH** {R4} **PUSH** {R5} and **PUSH** {R6}	R4 R5 R6	0x **000000EA** 0x **000000EB** 0x **000000EC**	SP	0x2 **0017FF4**	0x20017FF4> 0x20017FF8> 0x20017FFC> 0x20018000>	0x **00000000** 0x **000000EC** 0x **000000EB** 0x **000000EA**
After **MOV** R4,#0 **MOV** R5,#0 And **MOV** R6,#0	R4 R5 R6	0x **00000000** 0x **00000000** 0x **00000000**	SP	0x **20017FF4**	0x20017FF4> 0x20017FF8> 0x20017FFC> 0x20018000>	0x **00000000** 0x **000000EC** 0x **000000EB** 0x **000000EA**
After **POP** {R4} **POP** {R5} and **POP** {R6}	R4 R5 R6	0x **000000EC** 0x **000000EB** 0x **000000EA**	SP	0x **20018000**	0x20017FF4> 0x20017FF8> 0x20017FFC> 0x20018000>	0x **00000000** 0x **000000EC** 0x **000000EB** 0x **000000EA**

Fig. 2 – Evolution de la mémoire lors de l'exécution de STACKA

NB : lors d'un POP, les données contenues en haut de la Pile ne sont pas remises à zéro car il s'agirait d'une perte de temps. En effet le Pointeur de Pile a été modifié lors du POP, et donc lors du prochain PUSH la case redeviendra le haut de la pile et sera forcément réécrite

La pile repose sur le concept « Last In First Out ». Ainsi lors de la dernière étape avec les POP, le premier élément retiré est celui qui était contenu initialement dans le registre R6 puisqu'il était le dernier à être enregistré dans la Pile. Cette valeur se retrouve stockée dans le registre R4. Ensuite, R5 récupère sa valeur initiale, et enfin R6 prend la valeur qui avait été donnée à R4. L'état final des registres est donc inversé par rapport au départ
Ce n'est donc pas la bonne méthode pour restaurer les valeurs des registres. Il faut à la place réaliser les POP avec les registres dans l'ordre inverse aux PUSH initiaux

8.2 Basic use of the Stack : STACKB

Requested work 7: Study of STACKB

```
//////////////////////
    .global STACKB
    .text
STACKB:
    MOV R4,#234
    MOV R5,#235
    MOV R6,#236

    PUSH {R4}
    PUSH {R5}
    PUSH {R6}

    MOV R4,#0
    MOV R5,#0
    MOV R6,#0

    POP {R6}
    POP {R5}
    POP {R4}

END_STACKB: B END_STACKB
//////////////////////
```

Fig. 3 – Code de STACKB

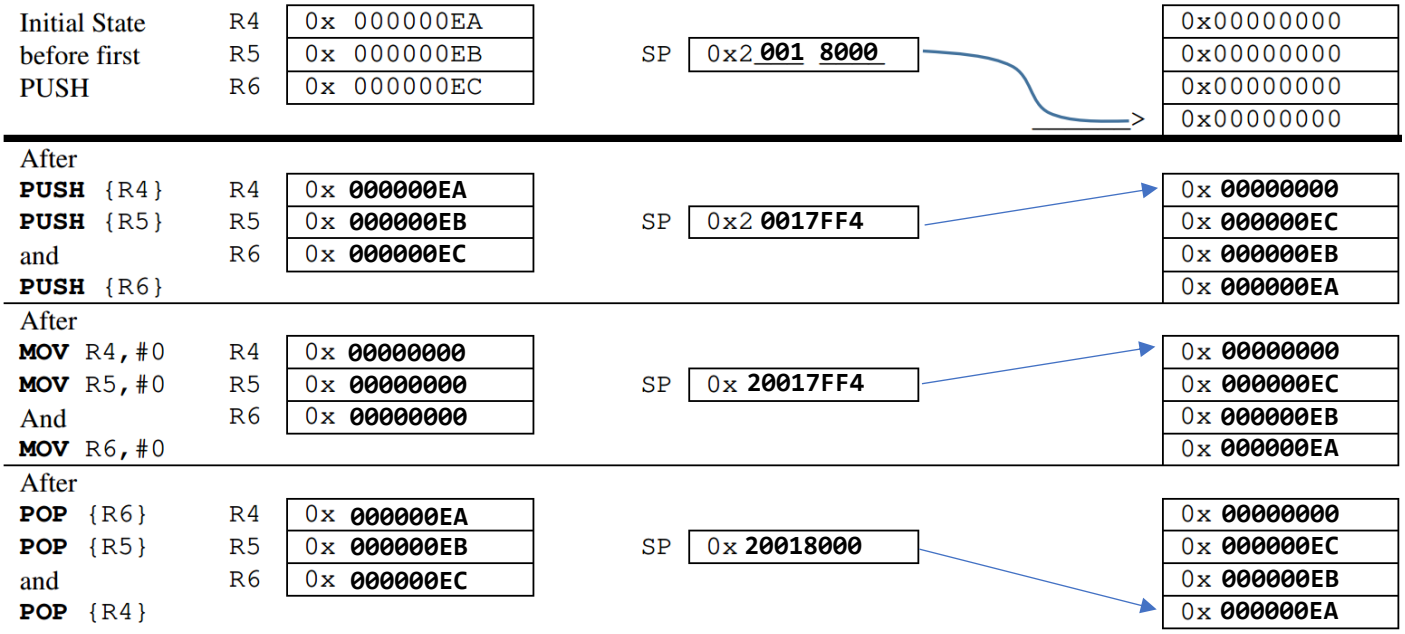


Fig. 4 – Evolution de la mémoire lors de l'exécution de STACKB

STACKB est globalement similaire à STACKA, mais la différence réside dans l'ordre dans lesquels les registres récupèrent les valeurs de la Pile avec POP.
Ici l'ordre des registres utilisé pour les instructions POP est bien l'inverse de celui utilisé auparavant pour PUSH : les registres récupèrent bien leur valeur initiale.

Puisque 3 éléments de 4 octets sont stockés dans la Pile durant l'exécution de STACKB, sont coût est alors de $3 \times 4 = 12$ octets

9 Use of the Stack to insure transparency of registers R4 to R7

9.1 A first try to use variables : Test_FUNCT1

8: Study of FUNCT1

```
//////////////////////
    .global Test_FUNCT1
    .text
Test_FUNCT1:

    MOV R4,#2 //first variable is 2
    MOV R5,#6 //second variable is 6
        BL FUNCT1   //call of FUNCT1
        ADD R4,R4,R5 // addition of the two variables
END_Test_FUNCT1: B END_Test_FUNCT1
//////////////////
    .global FUNCT1
    .text
FUNCT1:
        MOV R4,#0   // That's it!
        MOV R5,#0   // Not very useful we must admit it!
        BX LR       //and no more
////////////////////////////////
```

Fig. 5 – Code des fonctions `Test_FUNCT1` et `FUNCT1`

Lorsque que l'on appelle une fonction (ici `FUNCT1`), l'adresse de l'instruction suivant cet appel de fonction est stocké dans le registre spécial « Link Register ».
Ainsi l'instruction `BX  LR` à la fin de `FUNCT1` permet de branch à la suite de la fonction appelante `Test_FUNCT1`.

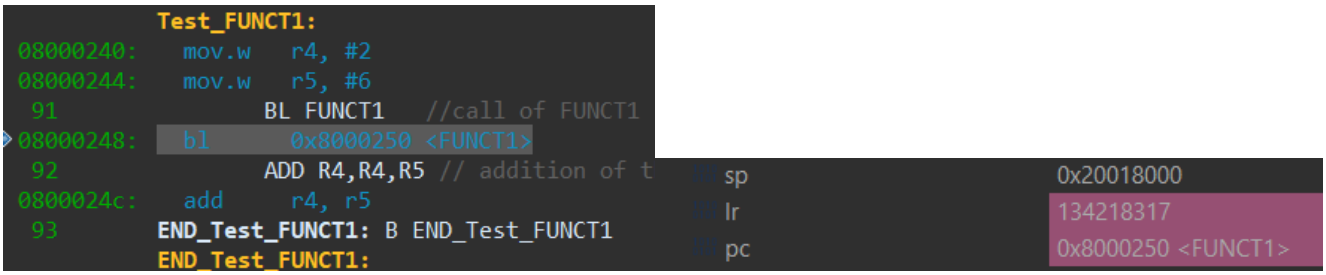


Fig. 6 et 7 – Vue de `Test_FUNCT1` dans la fenêtre « Disassembly » et du registre `LR` dans « Registers »

On peut vérifier cela en utilisant le débugueur : lors de l'exécution de `BL FUNCT1`, la valeur décimale `134218317` est stocké dans `LR`, ce qui correspond à `0x0800024D` en hexadécimal, c'est bien l'adresse de l'instruction suivante.

9.2 Another try to use variables : Test_FUNCT2

Requested work 9: Study of FUNCT2

```
//////////////////////
    .global Test_FUNCT2
    .text
Test_FUNCT2:
    MOV R4,#2 //first variable is 2
    MOV R5,#6 //second variable is 6
        BL FUNCT2   //call of FUNCT2
        ADD R4,R4,R5 // addition of the two variables
END_Test_FUNCT2: B END_Test_FUNCT2
//////////////////
    .global FUNCT2
    .text
FUNCT2:
        PUSH {LR}
        MOV R4,#0   // That's it!
        MOV R5,#0   // Not very useful again I must admit it!
        POP {PC}    //and no more
////////////////////////////////
```

Fig. 8 – Code des fonctions `Test_FUNCT2` et `FUNCT2`

Dans FUNCT2, la méthode pour reprendre la fonction appelante est différente : plutôt que d'utiliser un branch à la fin de la fonction, la valeur stockée dans LR (à savoir l'adresse de l'instruction après BL FUNCT2) est stockée dans la pile. Comme FUNCT2 n'ajoute et ne retire rien de la pile, on peut alors récupérer l'adresse qu'on avait stocké grâce à POP. Ainsi il suffit de mettre cette adresse dans le registre « Program counter » puisqu'elle indique au processeur l'adresse de la prochaine instruction à exécuter, permettant de revenir à la suite de Test_FUNCT2.
Cette méthode n'est pas meilleure en termes de mémoire puisque la méthode précédente n'utilisait justement pas la mémoire, alors qu'ici on stocke un élément dans la pile soit un coût de 4 octets.
Enfin cette fonction comme la précédente n'est pas transparente : les registres R4 et R5 sont réinitialisé à 0 par la FUNCT2 et ne reprennent pas leur valeur initiale avant de reprendre Test_FUNCT2.

9.3 Trying to protect variables : Test_FUNCT3

Requested work 10: Study of FUNCT3

```
//////////////////////
    .global Test_FUNCT3
    .text
Test_FUNCT3:
    MOV R4,#2 //first variable is 2
    MOV R5,#6 //second variable is 6
        BL FUNCT3   //call of FUNCT3
        ADD R4,R4,R5 // addition of the two variables
END_Test_FUNCT3: B END_Test_FUNCT3
//////////////////
    .global FUNCT3
    .text
FUNCT3:
        PUSH {R4,R5,LR}
        MOV R4,#0   // That's it!
        MOV R5,#0   // Not very useful again I must admit it!
        POP {R4,R5,PC}  //and no more
////////////////////////////////////
```

Fig. 9 – Code des fonctions Test_FUNCT3 et FUNCT3

La fonction est ici encore similaire à la précédente mais devient transparente : le PUSH et le POP au début et à la fin de la fonction incluent R4 et R5, ce qui permet de les restaurer lorsque la fonction se termine. On note ici que contrairement à l'utilisation de POP successif où il faut inverser l'ordre des registres par rapport à celui utilisé pour empiler, ici on garde bien le même ordre.
Par conséquent cette fonction a un coût en mémoire plus important que la précédente : on PUSH trois éléments dans la Pile donc on a un coût de 12 octets.

10 Use of the Stack to insure nested calls

10.1 Nested calls : Test_FUNCT4_5

Requested work 11: Study of nested calls

```
//////////////////////
    .global Test_FUNCT4_5
    .text
Test_FUNCT4_5:
    MOV R4,#2 //first variable is 2
    MOV R5,#6 //second variable is 6
        BL FUNCT4   //call of FUNCT4
        ADD R4,R4,R5 // addition of the two variables
END_Test_FUNCT4_5: B END_Test_FUNCT4_5
//////////////////
    .global FUNCT4
    .text
FUNCT4:
        PUSH {R4,R5,LR}
        MOV R4,#3 //first variable is 3
        MOV R5,#7 //second variable is 7
        BL FUNCT5 // remember: the call of a function should be transparent
for variables
        ADD R4,R4,R5 // addition of the 2 variables
        POP {R4,R5,PC}
//////////////////
```

Fig. 10 – Code des fonctions `Test_FUNCT4_5` et `FUNCT4`

```
//////////////////
    .global FUNCT5
    .text
FUNCT5:
        MOV R4,#0
        MOV R5,#0
        BX LR
//////////////
```

Fig. 11 – Code de la fonction et FUNCT5

Ici, on a une combinaison des méthodes vu précédemment : lorsque FUNCT4 est appelée, elle empile les valeurs de registre et l'adresse de retour. Elle continue ensuite son exécution en appelant notamment FUNCT5, écrasant donc la valeur contenue dans LR. FUNCT5 est exécutée normalement, puis revient à la suite dans FUNCT4 grâce à l'instruction BX LR. FUNCT4 continue à son tour et finit par récupérer les valeurs qu'elle a mis au départ dans la pile. Ainsi on reprend bien à la suite de Test_FUNCT4_5, et l'exécution est transparente puisque les registres ont repris leur valeur initiale.
On note toutefois que FUNCT5 n'est pas transparent : elle change les valeurs de R4 et R5 sans les remettre à celles qui avaient été donnée par FUNCT4.

10.2 Good call, bad call : Test_FUNCT6_7

Requested work 12: Study of bad nested call

```
//////////////////////
    .global Test_FUNCT6_7
    .text
Test_FUNCT6_7:
    MOV R4,#2 //first variable is 2
    MOV R5,#6 //second variable is 6
        BL FUNCT6   //call of FUNCT6
        ADD R4,R4,R5 // addition of the two variables
END_Test_FUNCT6_7: B END_Test_FUNCT6_7
//////////////////
```

Fig. 12 – Code de la fonction Test_FUNCT6_7

```
/////////////////
    .global FUNCT6
    .text
FUNCT6:
        PUSH {R4,R5}
        MOV R4,#3 //first variable is 3
        MOV R5,#7 //second variable is 7
        BL FUNCT7 // remember: the call of a function should be transparent
for variables
        ADD R4,R4,R5 // addition of the 2 variables
    POP {R4,R5}
    BX LR
/////////////////
    .global FUNCT7
    .text
FUNCT7:
    PUSH {R4,R5}
        MOV R4,#0
        MOV R5,#0
        POP {R4,R5}
        BX LR
/////////////
```

Fig. 13 – Code des fonctions FUNCT6 et FUNCT7

Lorsque FUNCT6 est appelée, elle ne sauvegarde pas l'adresse 0x080002C6 qui est stockée dans LR et qui permettrait de reprendre l'exécution de Test_FUNCT6_7. Ainsi lorsque FUNCT7 est appelée, l'adresse dans LR est écrasée par 0x080002D8 permettant de reprendre FUNCT6. FUNCT7 s'exécute donc normalement, l'instruction BX LR permet de reprendre l'exécution de FUNCT6, et à la fin de celle-ci, BX LR est à nouveau exécutée, et reprends donc au même endroit à l'intérieur de FUNCT6, puisque l'adresse de retour pour Test_FUNCT6_7 n'a pas été restaurée dans LR.

11 A more evolved use of the Stack : FUNCTN

Requested work 13: Translation in C

```
//////////////////
    .global FUNCTN
    .type  FUNCTN, %function
    .text
FUNCTN:     PUSH    {R4,LR}
// R0 is the first entering parameter of FUNCTN
        CMP     R0,#1
        BEQ     RET_FUNCTN
        MOV     R4,R0
        SUB     R0,#1 // before a call R0 is the entering parameter
        BL      FUNCTN
        MUL     R0,R4 // after a call R0 is the return value
RET_FUNCTN:
        POP     {R4,PC}
//////////////////////
```

Fig. 14 – Code de la fonction FUNCTN en assembleur

Le code assembleur et ses commentaires permettent de traduire en C :

```
int FUNCTN(int r0) {
    if (r0==1){
        return r0;
    }
    int n=r0;
    return FUNCTN(r0-1)*n;
}
```

Fig. 15 – Code de la fonction FUNCTN en C

Requested work 14: Test of FUNCTN

Lors de l'exécution pas à pas du code assembleur, on constate que FUNCTN est appelée 5 fois.
On peut représenter l'évolution de la Pile à chaque appel en s'inspirant du tableau donné dans l'énoncé :

1er appel	2e appel	3e appel	4e appel	5e appel
				2
				0x0800030D
			3	3
			0x0800030D	0x0800030D
		4	4	4
		0x0800030D	0x0800030D	0x0800030D
	5	5	5	5
	0x0800030D	0x0800030D	0x0800030D	0x0800030D
0	0	0	0	0
0x080002F9	0x080002F9	0x080002F9	0x080002F9	0x080002F9

Fig. 16 – évolution de la pile après chaque appel de FUNCTN

Le registre SP va quant à lui diminuer de $2 \times 4 = 8$ à chaque PUSH. Initialement, il vaut 0x020018000, puis après chaque appel : 0x020017FF8, 0x020017FF0, 0x020017FE8, 0x020017FE0, et 0x020017FD8.

FUNCTN vient calculer la factorielle de la valeur contenue dans R0.
En effet au tout début de la fonction, les valeurs de R4 et LR sont stockées dans la pile, pour assurer la transparence et un retour correct à la fonction précédente.
Puis, la fonction s'appelle récursivement tant que R0 est différent de 1. Mais à chaque appelle récursif, il y a un nouveau PUSH {R4,LR}, sachant que R4 a reçu la valeur de R0 lors de l'appel précédent et LR contient l'adresse de la ligne MUL R0,R4 située après l'appel récursif. Ainsi lorsque la fonction arrête de s'appeler récursivement (c'est-à-dire lorsqu'elle a été appelée assez de fois pour faire descendre R0 jusqu'à 1 grâce à l'instruction SUB R0,#1), elle appelle RET_FUNCTN, qui va dépiler jusqu'à obtenir l'adresse de retour qui a été sauvegardée au tout début. Entre temps, à chaque fois qu'on dépile, le registre PC retourne à l'adresse de MUL R0,R4 tandis que R4 retrace les valeurs qu'a prit R0 au cours de l'exécution.
Finalement lorsqu'on a assez dépilé pour revenir dans Test_FUNCTN, le registre R4 prend la valeur contenue dans R0 : 120, qui correspond bien à 5! .

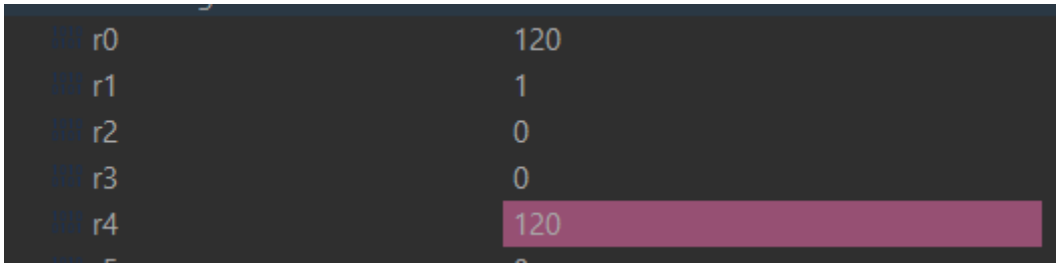


Fig. 17 – Valeurs contenues dans `R0` et `R4` à la fin de `Test_FUNCTN`

Après cette analyse, on peut ajouter un test à la fonction codée en C à la question précédente pour s'assurer qu'elle était correcte.

```c
#include <stdio.h>

int FUNCTN(int r0) {
    if (r0==1){
        return r0;
    }
    int n=r0;
    return FUNCTN(r0-1)*n;
}

int main(){
    printf("5! = %d", FUNCTN(5));
}
```

Fig. 18 – Code pour tester la traduction en C de `FUNCTN`

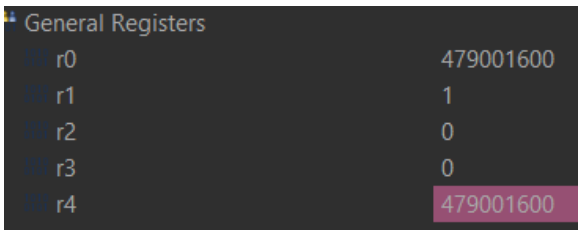
```
5! = 120

...Program finished with exit code 0
```

Fig. 19 – Test réussi de la fonction en C

En regardant le tableau précédemment réalisé (Fig. 16), on constate qu'au total 10 éléments sont stockés dans la Pile lorsqu'on appelle `FUNCTN` avec `R0=5`, soit un coût en mémoire de 40 octets. On peut généraliser en remarquant que quelque soit la valeur initiale n de `R0`, on `PUSH` une première fois, puis à chaque appel récursif on `PUSH` à nouveau jusqu'à atteindre `R0=1` (soit $n-1$ `PUSH`). On exécute donc n fois l'instruction `PUSH` avec à chaque fois deux éléments, donc on ajoute $2n$ éléments de 4 octets dans la Pile : le coût en mémoire de `FUNCTN` pour tout n est de $8n$ octets.

Au vu du coût en mémoire général que l'on vient de trouver, on en déduit que si l'on réserve 512 octets pour la Pile, alors la plus grande valeur de n que l'on peut renseigner sans dépasser cet espace réservé à la Pile est $n = \frac{512}{8} = 64$.

Toutefois, un autre problème va se présenter avant d'atteindre cette limite
En effet, le plus grand entier qu'il est possible de coder sur 32 bits est 2 147 483 647.
Or, $12! \approx 479\,000\,000$ et $13! \approx 6\,200\,000\,000$: on en déduit qu'on va avoir un dépassement si on essaie de calculer 13! avec `FUNCTN`, et que la plus grande valeur initiale de `R0` qu'il est possible de mettre est `12`. On peut faire le test avec le débugeur.



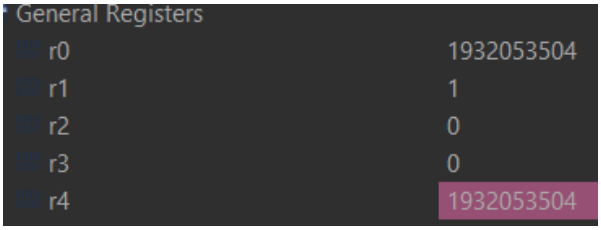


Fig. 20 et 21 – Valeurs contenue dans `R4` à la fin de `Test_FUNCTN`, lorsque `R0` est initialisé à `12` puis à `13`

Le résultat final dans le cas où `R0` est initialisé à `13` est inférieur à 2 millions et ne correspond donc pas du tout au résultat de plus de 6 millions qui était attendu : le plus grand paramètre initial que l'on peut renseigner tout en ayant un résultat final correct est `12`.