Documentation - Génération de Code - PCL2 - ARM / VisUAL

Voici une documentation / un rappel de cours sur l'assembleur ARM UAL (émulé par VisUAL) permettant de faciliter la partie Génération de Code du module Projet de Compilation des Langages.

I - Instructions natives :

Les instruction utilisables en ARM UAL sont les suivantes :
(Il est à noter que ces différentes instructions sont à retrouver sur le site de VisUAL. Vous y trouverez de plus amples détails sur les différentes options existantes pour chaque opérateur)

Opérateur	Exemples d'utilisation	Explication	Commentaires
ADD, ADC, SUB, SBC, RSB, RSC	ADD R0, R1, R2 SUB R0, R1, #1	R0 ← R1 + R2 R0 ← R1 - 1	Opérateurs arithmétiques (addition, soustraction, etc)
AND, EOR, BIC, ORR	AND R0, R1, R2 ORR R0, R1, R2	R0 ← R1 et R2 R0 ← R1 ou R2	Opérateurs binaires logiques (ET, OU, etc)
LSL, LSR, ASR, ROR	LSL R0, R1, R2 LSL R0, R1, #4	R0 ← R1 << R2	Opérateurs de *shift* (décalage)
CMP, CMN, TST, TEQ	CMP R0, #0 TEQ R0, R1	Comparaison entre R0 et 0 et mise à jours des flags en conséquence Les flags N et Z sont changés en fonction de R0 XOR R1	Opérateurs de comparaisons

MOV, MVN	MOV R0, #2
	MVN R0, #2
	
	MOV R0, R1
		R0 ← 2
		R0 ← -3 (-3 est l'inverse bit à bit de 2)
		valeur sur 8 bits uniquement (cf. LDR pour 32 bits)
		
		R0 ← R1
			Opérateurs de type *move*

ADR	ADR R0, label ADR R0, 2	Charge dans le registre R0 la valeur donnée **sur 8 bits** (cf. LDR pour 32 bits) Équivalent à MOV R0, #label	Address Load

LDR	LDR R1, [R0]
	
	LDR R1, =label
	LDR R1, =1234
		Si par exemple R0 = 2 alors R1 prend la valeur qui est contenue dans l'adresse 0x2
		
		Charge dans le registre R1 la valeur donnée **sur 32 bits**
			Load Register

STR	STR R1, [R2]	Si par exemple R2 = 2 alors on met le contenu de R1 dans l'adresse 0x2	Store Register
LDM[dir]	LDMFA SP!, {R1-R4}	Charge dans les registres R1 à R4 les données en mémoire aux adresses SP à SP-4*4 (en direction Full Ascending)	Load Multiple Registers
STM[dir]	STMFA SP!, {R1-R4}	Stocke le contenu des registres R1 à R4 en mémoire aux adresses SP à SP+4*4 (en direction Full Ascending)	Store Multiple Registers
B	B étiquette	Permet de faire un saut à l'adresse de l'étiquette	Branch
BL	BL étiquette	Charge dans le Link Register l'adresse de la prochaine instruction (PC) et effectuer ensuite un saut vers l'adresse de l'étiquette	Branch with link
DCD	étiquette DCD 1,2,3	Écrit les mots 1, 2 et 3 dans étiquette	Define Constant Word (32 bits)
EQU	quatre EQU 4	Permet la "déclaration de variables", maintenant dès que l'on écrit quatre, le programme le remplace par 4	Declare Constant Semblable au DEFINE du langage C
FILL	my_buffer FILL 100	Alloue 100 octets en mémoire, à l'adresse donnée par le label *my_buffer*	Declare Empty Word(s) in Memory
END	END	Stoppe l'émulation	Fin de programme

Les instructions peuvent aussi s'accompagner de certaines *conditions*, nous avons fait le choix de ne pas les détailler dans le tableau précédent mais de le mentionner à part par souci de lisibilité. On a alors le tableau suivant :

Suffixe	Flags	Signification
EQ	Z = 1	Equal
NE	Z = 0	Not Equal
CS / HS	C = 1	Higher or same (unsigned)
CC / LO	C = 0	Lower (unsigned)
MI	N = 1	Negative
PL	N = 0	Positive or 0
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 && Z = 0	Higher (unsigned)
LS	C = 0 || Z = 1	Lower or same (unsigned)
GE	N = V	Greater or equal (signed)
LT	N != V	Less than or equal (signed)
GT	Z = 0 && N = V	Greater than (signed)
LE	Z = 1 && N != V	Less than or equal (signed)
AL	N'importe quelle valeur	Tout le temps ! Équivalent à ne pas mettre de condition

L'option S permet de mettre à jour les flags lors de l'exécution de l'instruction possédant cette option (ex : ADDS R0, R0, #1).

II - Instructions non présentes :

Vous l'avez sans doute déjà remarqué : certaines opérations ne sont pas présentes dans le précédent tableau, c'est parce qu'elles ne sont pas implémentées nativement dans cet émulateur.

On y retrouve entre autres :

- la multiplication
- la division
- itoa (int to ascii ou int to string)
- la concaténation (pour les chaînes de caractères par exemple)
- l'affichage (print)

Les trois premières sont fournies en annexe (8), (9) et (10).

La concaténation de String est laissée comme exercice au lecteur et permet une bonne compréhension du langage ARM UAL.

Pour ce qui est de l'affichage, l'émulateur VisUAL ne possède malheureusement pas de système d'entrée/sortie et il n'est donc pas possible facilement de créer un affichage, c'est pourquoi cette opération va être détaillée dans ce document partie **VI**.

III - Exemples de code :

Cette partie consiste à étudier la génération de code dans les grandes lignes. Elle sera organisée de la façon suivante : vous trouverez d'abord des morceaux de code (écrits en langage C ou en Java pour les classes, etc. mais les principes restent les mêmes dans les autres langages) et ensuite vous trouverez le code assembleur correspondant qui sera plus ou moins commenté.

Il est à noter que R12 est défini comme étant le Base Pointer (BP).

1 - Déclaration et affectation d'une variable (code en C pour illustrer)

```
int x;   // Déclaration
x = 2;   // Affectation
```

```
ADD SP, SP, #4 ; On incrémente SP pour laisser de la place
pour la variable locale x (int = 4 octets)
LDR R7, =2 ; LDR au lieu de MOV car MOV ne permet pas la
gestion des nombres de plus de 8 bits
ADD R8, R12, #0 ; R8 := @x
STR R7, [R8] ; x := 2
```

2 - Opérations basiques (code en C pour illustrer)

```
// Déclarations
int x;
int y;
int a;
// Affectations
x = 2;
y = 3;
// Opération
a = x + y;
```

```
ADD SP, SP, #12 ; On incrémente SP pour laisser de la place pour les
variables locales x, y, a (3 * int = 3 * 4 = 12 octets)
LDR R7, =2 ; R7 := 2
STR R7, [R12, #0] ; x := 2
LDR R7, =3 ; R7 := 3
STR R7, [R12, #4] ; y := 3
LDR R8, [R12, #0] ; R8 := x
LDR R9, [R12, #4] ; R9 := y
ADD R7, R8, R9 ; R7 := x + y
STR R7, [R12, #8] ; a := x + y
```

3 - Conditionnelle IF (code en C pour illustrer)

```c
int x = 55;

if (1){
   x = 10;
}else{
   x = 20;
}
```

```
ADD SP, SP, #4 ; On incrémente SP pour laisser de la place pour la variable
locale x (int = 4 octets)
LDR R0, =55 ;
STR R0, [R12, #0] ; x := 55
LDR R0, =1 ;
CMP R0, #0 ; Z := 0
BEQ loop0 ; Ne branch pas à loop0 car Z = 0
LDR R7, =10 ; R7 := 10
STR R7, [R9, #0] ; x := 10
B loop0_end ; On saute à la fin
loop0 LDR R7, =20 ; R7 := 20
STR R7, [R12, #0] ; x := 20
loop0_end B _exit ; Fin du if
```

4 - Boucle WHILE (code en C pour illustrer)

```c
int x = 10;

while (x > 0){
   x = x - 1;
}
```

```
ADD SP, SP, #4 ; On incrémente SP pour laisser de la place pour la
variable locale x (int = 4 octets)
LDR R0, =10 ;
STR R0, [R12, #0] ; x := 10
loop0 LDR R7, [R12, #0] ; R7 := x
LDR R8, =0 ; R8 := 0 (le 0 de la condition)
CMP R7, R8 ; Z := x - 0 <? 0
MOV R0, #0 ; Le résultat est faux par défaut, R0 := 0
MOVGT R0, #1 ; Si Z = 0, R0 := 1
CMP R0, #0 ; Z := NOT R0
BEQ end0 ; On saute à la fin si Z = 1
LDR R8, [R8, #0] ; R8 := x
LDR R9, =1 ;
SUB R7, R8, R9 ; R7 := x - 1
STR R7, [R9, #0] ; x := x - 1
B loop0 ;
end0 B _exit ; Fin du while
```

5 - Manipulation de chaînes de caractères (code en Java pour illustrer)

```java
String s;
s = "Hello!";
```

```
ADD SP, SP, #4 ; On incrémente SP pour laisser de la place pour la
variable locale s (string = 4 octets)
_str_1 DCD 0x6C6C6548, 0x216F ;on définit "Hello!" avec l'étiquette _str_1
;              l l e H    ! o
LDR R7, =_str_1 ; R7 := "Hello!"
STR R7, [R12, #0] ; s := "Hello!"
```

6 - Déclaration d'une classe (code en Java pour illustrer)

Les différents codes ARM issus de langage orienté objet ont été automatiquement créés par notre compilateur, ces codes ne sont donc pas optimisés et certaines suites d'opérations peuvent être factorisées.

```java
public class Voiture {

   public int annee;
   public String proprio;

   public static int nb = 0;

   public Voiture(int annee, String proprio){
      this.annee = annee;
      this.proprio = proprio;
      nb = nb + 1;
   }

   public int getAnnee(){
      return this.annee;
   }

   public String getProprio(){
      return this.proprio;
   }

   public void setAnnee(int n){
      this.annee = n;
   }

   public void setProprio(String s){
      this.proprio = s;
   }

}
```

```
_init           MOV R12, SP ;
                MOV R3, #0 ;
                LDR R7, =_heap ;
                ADD R7, R7, R3 ;
                MOV R8, #16 ;
                STR R8, [R7] ;
instanceSize(Voiture) := 16
                LDR R8, =Voiture_Voiture ;
                STR R8, [R7, #4] ;
                LDR R8, =Voiture_setProprio ;
                STR R8, [R7, #8] ;
                LDR R8, =Voiture_getAnnee ;
                STR R8, [R7, #12] ;
                LDR R8, =Voiture_setAnnee ;
                STR R8, [R7, #16] ;
                LDR R8, =Voiture_getProprio ;
                STR R8, [R7, #20] ;
                ADD R3, R3, #28 ; HeapOffset +=
descriptorSize(Voiture)
                B _main ;
                LDR R0, =0 ;
Voiture_Voiture STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
                STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop0 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop0 ;
                ADD R0, R8, #-16 ; R0 := @annee
                LDR R0, [R0] ; R0 := annee
                MOV R7, R0 ; R7 := result
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop1 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop1 ;
                ADD R0, R8, #-12 ; R0 := @this
                LDR R0, [R0] ; R0 := this
                ADD R8, R0, #12 ;
                STR R7, [R8] ;
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop2 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop2 ;
                ADD R0, R8, #-20 ; R0 := @proprio
                LDR R0, [R0] ; R0 := proprio
                MOV R7, R0 ; R7 := result
                MOV R8, R12 ; R8 := BP
```

```
                MOV R9, #1 ;
_identifier_loop3 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop3 ;
                ADD R0, R8, #-12 ; R0 := @this
                LDR R0, [R0] ; R0 := this
                ADD R8, R0, #8 ;
                STR R7, [R8] ;
                LDR R8, =_heap ;
                ADD R8, R8, #24 ; R8 :=
@Voiture.nb
                LDR R0, [R8] ; R0 := Voiture.nb
                MOV R8, R0 ;
                LDR R0, =1 ;
                MOV R9, R0 ;
                ADD R0, R8, R9 ;
                MOV R7, R0 ; R7 := result
                LDR R8, =_heap ;
                ADD R8, R8, #24 ; R8 :=
@Voiture.nb
                STR R7, [R8] ;
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                LDR R7, [SP, #-4]! ;
                MOV PC, R7 ;
Voiture_getAnnee STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
                ADD SP, SP, #4 ; increment SP to
leave room for method variables (result)
                ADD R0, R12, #-12 ; R0 := @this
                LDR R0, [R0] ; R0 := this
                ADD R7, R0, #12 ;
                LDR R0, [R7] ; R0 :=
Voiture.annee
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                LDR R7, [SP, #-4]! ;
                MOV PC, R7 ;
Voiture_getProprio STR R12, [SP], #4 ; push old
BP
                MOV R12, SP ; BP := SP
                ADD SP, SP, #4 ; increment SP to
leave room for method variables (result)
                ADD R0, R12, #-12 ; R0 := @this
                LDR R0, [R0] ; R0 := this
                ADD R7, R0, #8 ;
                LDR R0, [R7] ; R0 :=
Voiture.proprio
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                LDR R7, [SP, #-4]! ;
                MOV PC, R7 ;
Voiture_setAnnee STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
                STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
```

```
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop4 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop4 ;
                ADD R0, R8, #-16 ; R0 := @n
                LDR R0, [R0] ; R0 := n
                MOV R7, R0 ; R7 := result
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop5 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop5 ;
                ADD R0, R8, #-12 ; R0 := @this
                LDR R0, [R0] ; R0 := this
                ADD R8, R0, #12 ;
                STR R7, [R8] ;
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                LDR R7, [SP, #-4]! ;
                MOV PC, R7 ;
Voiture_setProprio STR R12, [SP], #4 ; push old
BP
                MOV R12, SP ; BP := SP
                STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop6 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop6 ;
                ADD R0, R8, #-16 ; R0 := @s
                LDR R0, [R0] ; R0 := s
                MOV R7, R0 ; R7 := result
                MOV R8, R12 ; R8 := BP
                MOV R9, #1 ;
_identifier_loop7 LDR R8, [R8, #-4] ; R8 :=
BP_old
                SUBS R9, R9, #1 ; offsetCount--
                BNE _identifier_loop7 ;
                ADD R0, R8, #-12 ; R0 := @this
                LDR R0, [R0] ; R0 := this
                ADD R8, R0, #8 ;
                STR R7, [R8] ;
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore old
BP
                LDR R7, [SP, #-4]! ;
                MOV PC, R7 ;
                B _exit ;
```

7 - Instanciation d'une classe et appel de méthodes (code en Java pour illustrer)

```java
Voiture bipbip = new Voiture(2021, "moi");
String s = bipbip.getProprio();
bipbip.setAnnee(2018);
int n = bipbip.getAnnee();
```

```
Code précédent à inclure ici (il faut bien
déclarer la classe avant de l'utiliser)

_main           STR R12, [SP], #4 ; push old BP
                MOV R12, SP ; BP := SP
                ADD SP, SP, #12 ; increment SP
to leave room for local variables
                LDR R0, =2021 ;
_str_1 DCD 0x696F6D ;
                LDR R0, =_str_1 ;
                LDR R7, =_heap ; R7 :=
@descriptor(Voiture)
                LDR R8, =_heap ;
                ADD R8, R8, R3 ;
                STR R7, [R8] ;
                ADD R3, R3, #16 ; HeapOffset +=
instanceSize(Voiture)
                LDR R7, =_heap ;
                LDR R9, [R7, #4] ;
                STMFA SP!, {R5,R4,R6,R8,R9} ;
                ADD SP, SP, #16 ; increment SP
to leave room for 'this' and method parameters
                STR R8, [SP, #-4] ;
Voiture::param(this)
                LDR R0, =2021 ;
                STR R0, [SP, #-8] ;
Voiture::param(annee)
                MOV R7, R0 ;
                MOV R0, R8 ;
                ADD R10, R0, #12 ;
                STR R7, [R10] ;
_str_2 DCD 0x696F6D ;
                LDR R0, =_str_2 ;
                STR R0, [SP, #-12] ;
Voiture::param(proprio)
                MOV R7, R0 ;
                MOV R0, R8 ;
                ADD R10, R0, #8 ;
                STR R7, [R10] ;
                STR PC, [SP], #4 ;
                MOV PC, R9 ; Voiture::Voiture()
                SUB SP, SP, #16 ;
                LDMFA SP!, {R5,R4,R6,R8,R9} ;
                MOV R0, R8 ;
                STR R0, [R12, #0] ; bipbip :=
result
                ADD R0, R12, #0 ; R0 := @bipbip
                LDR R0, [R0] ; R0 := bipbip
                LDR R8, =_heap ;
                LDR R9, [R8, #20] ;
                STMFA SP!, {R5,R4,R6,R7,R9} ;
                ADD SP, SP, #4 ; increment SP to
leave room for 'this' and method parameters
                STR R0, [SP, #-4] ;
                STR PC, [SP], #4 ;
                MOV PC, R9 ;
Voiture::getProprio()
                SUB SP, SP, #4 ;
                LDMFA SP!, {R5,R4,R6,R7,R9} ;
                MOV R7, R0 ; R7 := result
```

```
                ADD R0, R12, #4 ; R0 := @s
                MOV R8, R0 ; R8 := @s
                STR R7, [R8] ; s := result
                ADD R0, R12, #0 ; R0 := @bipbip
                LDR R0, [R0] ; R0 := bipbip
                LDR R7, =_heap ;
                LDR R8, [R7, #16] ;
                STMFA SP!, {R5,R4,R6,R8} ;
                ADD SP, SP, #8 ; increment SP to
leave room for 'this' and method parameters
                STR R0, [SP, #-4] ;
                LDR R0, =2018 ;
                STR R0, [SP, #-8] ;
setAnnee::param(n)
                STR PC, [SP], #4 ;
                MOV PC, R8 ; Voiture::setAnnee()
                SUB SP, SP, #8 ;
                LDMFA SP!, {R5,R4,R6,R8} ;
                ADD R0, R12, #0 ; R0 := @bipbip
                LDR R0, [R0] ; R0 := bipbip
                LDR R8, =_heap ;
                LDR R9, [R8, #12] ;
                STMFA SP!, {R5,R4,R6,R7,R9} ;
                ADD SP, SP, #4 ; increment SP to
leave room for 'this' and method parameters
                STR R0, [SP, #-4] ;
                STR PC, [SP], #4 ;
                MOV PC, R9 ; Voiture::getAnnee()
                SUB SP, SP, #4 ;
                LDMFA SP!, {R5,R4,R6,R7,R9} ;
                MOV R7, R0 ; R7 := result
                ADD R0, R12, #8 ; R0 := @n
                MOV R8, R0 ; R8 := @n
                STR R7, [R8] ; n := result
                MOV SP, R12 ; SP := BP
                LDR R12, [SP, #-4]! ; restore
old BP
                B _exit ;
```

IV - Exécution d'un code :

L'objectif de cette partie est de montrer l'exécution d'un code tout en la liant au fonctionnement de la pile. Pour cela, nous allons reprendre l'exemple précédent sur l'opération simple d'addition de deux variables.

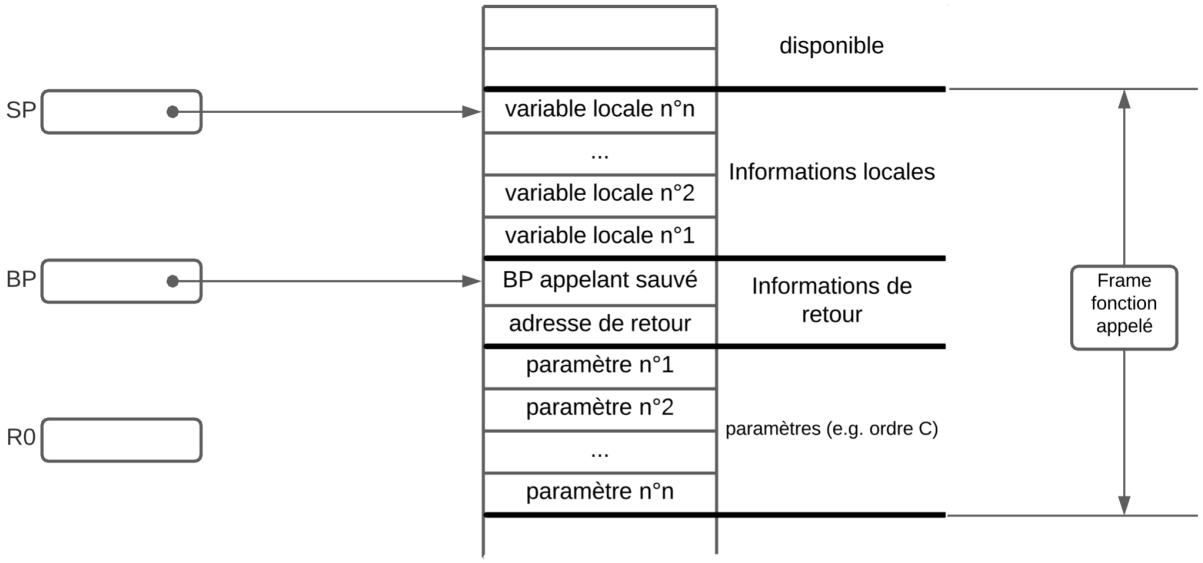
```
// Déclarations
int x;
int y;
int a;

// Affectations
x = 2;
y = 3;

// Opération
a = x + y;
```

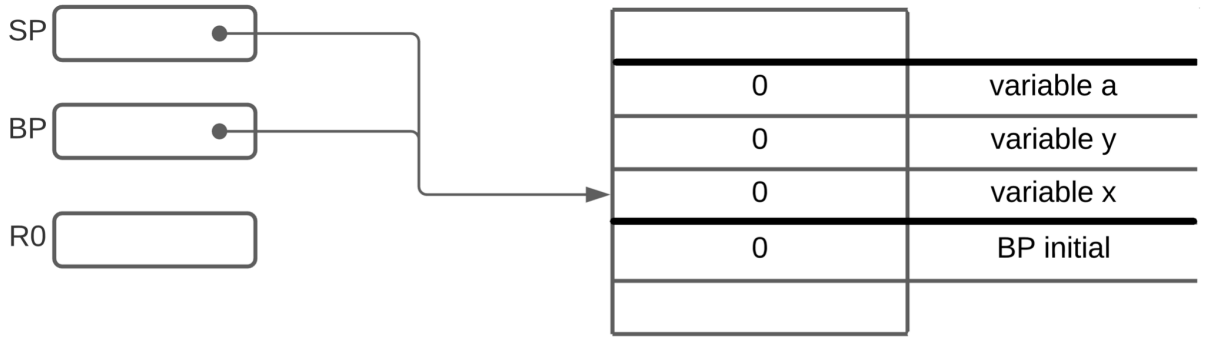
```
ADD SP, SP, #12 ; On incrémente SP pour laisser de la
place pour les variables locales x, y, a (3 * int = 3 * 4
= 12 octets)
LDR R7, =2 ; R7 := 2
STR R7, [R12, #0] ; x := 2
LDR R7, =3 ; R7 := 3
STR R7, [R12, #4] ; y := 3
LDR R8, [R12, #0] ; R8 := x
LDR R9, [R12, #4] ; R9 := y
ADD R7, R8, R9 ; R7 := x + y
STR R7, [R12, #8] ; a := x + y
```

On commence par rappeler le format classique d'une pile :

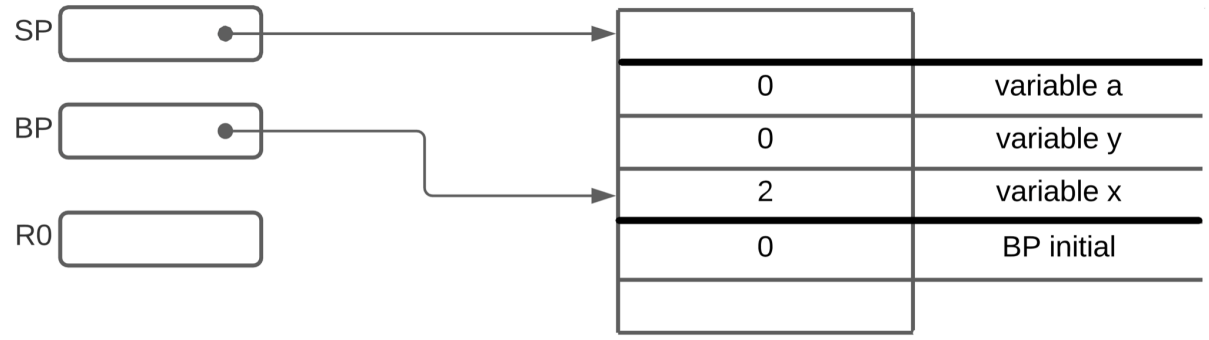


On obtient alors le déroulement suivant pour le programme effectuant a = x + y :

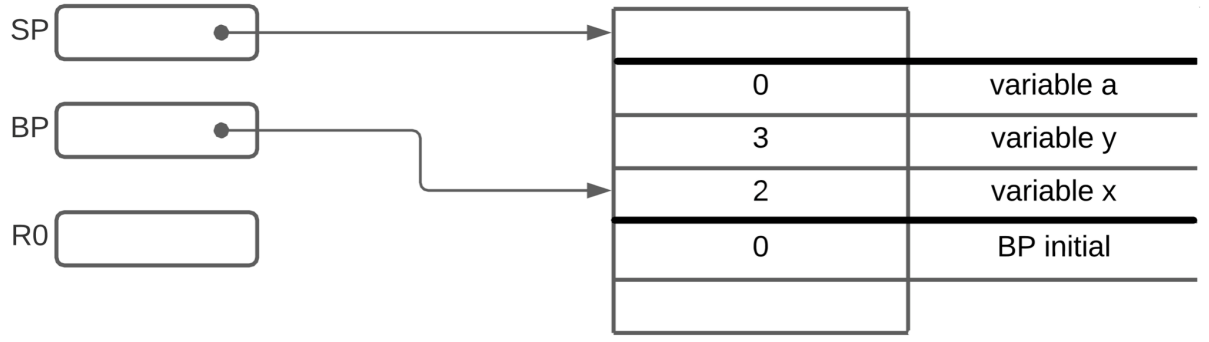
```
ADD SP, SP, #12 ; On incrémente SP pour laisser de la place pour les
variables locales x, y, a (3 * int = 3 * 4 = 12 octets)
```



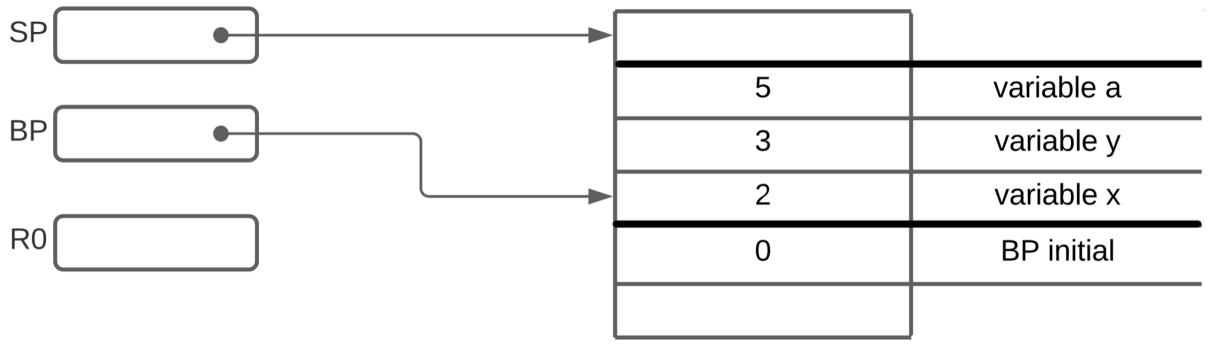

```
LDR R7, =2 ; R7 := 2
STR R7, [R12, #0] ; x := 2
```




```
LDR R7, =3 ; R7 := 3
STR R7, [R12, #4] ; y := 3
```



```
LDR R8, [R12, #0] ; R8 := x
LDR R9, [R12, #4] ; R9 := y
ADD R7, R8, R9 ; R7 := x + y
STR R7, [R12, #8] ; a := x + y
```



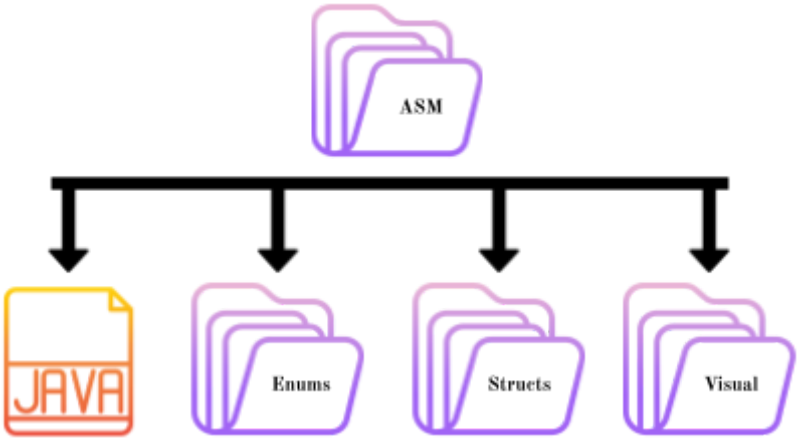
V - Structure du projet et du code ARM généré :

Le projet de compilation est un projet assez conséquent, il contient ainsi un nombre important de lignes et une mauvaise structuration du projet peut empêcher d'avancer sereinement vers un compilateur fonctionnel. Il est donc important, voire indispensable, de structurer proprement vos fichiers dans différents dossiers et classes qui auront chacun et chacune un rôle particulier.

Nous allons vous proposer une structure pour la partie Génération de Code de votre compilateur. Il est à noter que ce n'est qu'une proposition et que d'autres structures sont possibles.

Pour commencer, vous pouvez créer un nouveau dossier (ou package) *assembleur*. Ce dossier contiendra tout ce que vous allez faire durant cette partie du projet.

Dans ce dossier, vous pouvez créer différents sous-dossiers contenant par exemple vos énumérations ou vos structures et laisser à la racine de ce dossier vos codes Java permettant la génération du code ARM. Il est préférable aussi de créer un dossier *visual* qui permettra l'exécution de l'émulateur VisUAL ainsi que l'affichage d'une chaîne de caractères (développé dans le point suivant).



Pour commencer votre compilateur, il va vous falloir être capable d'écrire du code ARM dans un fichier pour pouvoir l'exécuter ensuite. Pour cela, il vous est conseillé de créer une classe dédiée permettant d'écrire toutes les instructions disponibles en ARM dans un fichier destination contenant l'ensemble de votre code compilé à partir d'un fichier source.

Ensuite, vous pouvez créer une nouvelle classe qui vous permettra quant à elle de choisir ce que vous souhaitez écrire dans ce fichier destination en fonction du nœud sur lequel vous vous trouvez.

Enfin, il vous est conseillé de structurer votre fichier ARM final de la façon suivante :

- Commencer par écrire les différentes opérations non natives (subroutines) de VisUAL (multiplier, diviser, afficher, …) en haut du fichier
- Puis écrire le code assembleur correspondant au fichier source fourni en entrée du programme.

VI - Affichage d'une chaîne de caractères :

L'émulateur VisUAL ne supporte pas nativement la manipulation et l'affichage de chaînes de caractères. Cette partie présente une solution permettant malgré tout d'émuler une telle fonctionnalité en passant par du code Java.

Cette solution se divise en trois parties :

- Les subroutines *print* et *println* présentées dans l'annexe (5). Leur rôle est de recopier les chaînes de caractères qui leur sont passées en paramètre dans l'espace mémoire *_str_out* qui leur est dédié.
- La génération de l'instruction pour stocker une chaîne de caractères en mémoire.
- Les classes Java présentées dans les annexes (1) à (4) permettant d'exécuter de manière contrôlée le code assembleur généré (en particulier en utilisant le mode *logging* décrit plus en détails ci-dessous), puis d'extraire dans les fichiers de logs générés le contenu de l'espace mémoire *_str_out* afin d'en décoder les chaînes de caractères et de les afficher.

1) *Subroutines print et println*

Ces deux subroutines ont un fonctionnement identique, mis à part l'ajout d'un caractère *newline* par la deuxième.

Il est essentiel que ces deux subroutines soient insérées immédiatement au début du fichier assembleur généré, de sorte à ce qu'elles se situent toujours à la même ligne du fichier assembleur final. Cela permettra à la classe *Launcher* de définir des breakpoints à une position fixe dans le code assembleur.

L'espace mémoire *_str_out* est alloué avec une taille prédéterminée. C'est dans cet espace que seront recopiées les chaînes de caractères dont l'adresse est passée dans le registre R0 lors d'un appel. Une chaîne de caractères doit être terminée par un *null byte*, comme dans le langage C.

2) *Stockage d'une chaîne de caractères en mémoire*

VisUAL ne permettant pas nativement de stocker une chaîne de caractères en mémoire, il est nécessaire de la convertir en série de valeurs hexadécimales qui peuvent être chargées en mémoire grâce à l'instruction DCD documentée plus haut.

L'annexe (6) montrent comment il est possible de générer l'instruction assembleur permettant de charger la chaîne de caractère "Hello, world!" en mémoire, à l'adresse donnée par le label *_str_hello*. Ce code utilise en particulier la méthode décrite dans l'annexe (5).

3) *Exécution et extraction du texte affiché*

La classe *Launcher* en annexe (4) montre comment utiliser la version *headless* (sans interface graphique) de l'émulateur VisUAL en l'exécutant directement depuis notre compilateur en Java. Cela nécessite d'indiquer à IntelliJ (ou à Gradle s'il est utilisé) d'utiliser le fichier *visual_headless.jar* en tant que bibliothèque afin d'avoir accès à toutes ses classes et méthodes. En particulier, l'utilisation des annexes (2) et (3) est nécessaire afin d'empêcher à l'émulateur d'appeler *System.exit*, ce qui aurait pour effet de faire quitter le programme entier.

On notera également l'importance de instMemSize, qui permet de fixer la taille de l'espace mémoire dédié à la zone de code (contenant les instructions en mémoire), et qui permet ainsi de connaître à l'avance l'adresse en mémoire du premier symbole qui sera l'espace mémoire *_str_out* dans lequel les strings sont recopiées.

La fonctionnalité de *logging* de l'émulateur est en particulier exploitée pour obtenir le contenu de *_str_out à* chaque exécution des subroutines de print. Pour cela, on configure le logging pour écrire dans un fichier le contenu de *_str_out* à chaque breakpoint, puis on définit les breakpoints à la fin de chacune des subroutines. Cela indique à l'émulateur de générer un fichier de log au format XML, qui contient sous forme hexadécimale le contenu de la mémoire à chaque adresse contenue dans *_str_out*.

Une fois le fichier de log XML généré, la classe *OutParser* est chargée de le lire et d'en extraire les différentes lignes de textes "affichées". Chaque ligne obtenue peut être alors affichée comme montré à la fin de la classe *Launcher* en annexe (4).

Pour exécuter un code ARM généré par votre compilateur (out.S), il ne vous reste plus qu'à lancer la fonction run de la classe *Launcher* comme suivant :

```
Launcher.run("out.S");
```

VII - Annexe :

(1) Outparser.java :

```
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;

public final class OutputParser {
    private OutputParser() {}

    private static boolean hexStringToByteArray(ByteArrayOutputStream bytes, String hex)
{
        String hexValue = hex.substring(2);
        int l = hexValue.length();
        if (l % 2 == 1) {
            /* pad to a pair number of hex digits */
            hexValue = "0" + hexValue;
            l++;
        }

        for (int i = l - 2; i >= 0; i -= 2) {
            int currentByte = (Character.digit(hexValue.charAt(i), 16) << 4)
                    + Character.digit(hexValue.charAt(i + 1), 16);
            if (currentByte == 0) return false;
            bytes.write(currentByte);
        }

        return true;
    }

    private static String parseLine(Node line) {
        NodeList children = line.getChildNodes();

        ByteArrayOutputStream bytes = new ByteArrayOutputStream();

        for (int j = 0; j < children.getLength(); j++) {
            Node child = children.item(j);
```

```
            if (child.getNodeName().equals("word")) {
                boolean more = hexStringToByteArray(bytes, child.getTextContent());
                if (!more) break;
            }
        }

        return bytes.toString(StandardCharsets.UTF_8);
    }

    public static List<String> parseOutput(String XMLPath) {
        List<String> outputs = new ArrayList<>();

        try {
            File XMLFile = new File(XMLPath);
            DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(XMLFile);
            doc.getDocumentElement().normalize();
            NodeList lines = doc.getElementsByTagName("line");

            for (int i = 0; i < lines.getLength(); i++) {
                Node line = lines.item(i);
                outputs.add(parseLine(line));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return outputs;
    }
}
```

(2) ExitTrapper.java :

```
import java.security.Permission;

public class ExitTrapper {
    public static void forbidSystemExitCall() {
        final SecurityManager securityManager = new SecurityManager() {
            public void checkPermission( Permission permission ) {
                if( permission.getName().startsWith("exitVM") ) {
                    int code = Integer.parseInt(permission.getName().split("\\.")[1]);
                    throw new ExitTrappedException(code) ;
                }
            }
        } ;
        System.setSecurityManager( securityManager ) ;
    }

    public static void enableSystemExitCall() {
        System.setSecurityManager( null ) ;
    }
}
```

(3) ExitTrappedException.java :

```
public class ExitTrappedException extends SecurityException {
    private final int code;

    public ExitTrappedException(int code) {
        super();
        this.code = code;
    }

    public int getCode() {
        return code;
    }
}
```

(4) Launcher.java :

```
import java.util.List;
import java.util.stream.Stream;

import BloodCompiler.assembly.InstructionWriter;
import BloodCompiler.assembly.visual.security.ExitTrappedException;
import BloodCompiler.assembly.visual.security.ExitTrapper;

import visual.EmulatorLogFile;
import visual.HeadlessController;

public class Launcher {
    private static final int instMemSize = 0x10000;
    /* we make sure that the output buffer is always the first symbol in memory */
    private static final int outputBufferAddress = instMemSize;

    /* VisUAL offsets line numbers by one for some reason */
    private static final List<Integer> breakpoints = List.of(11 - 1, 25 - 1);

    private Launcher() {}

    /* array of all word addresses in the output buffer */
    private static String[] getOutputRange() {
        return Stream.iterate(outputBufferAddress, n -> n + 4)
                .limit(InstructionWriter.outputBufferLength / 4)
                .map(n -> String.format("0x%X", n))
                .toArray(String[]::new);
    }

    public static List<String> executeAndParseOutput(String assemblyFile) {
        EmulatorLogFile.configureLogging("", true, false, false, false, false, false,
true, false, getOutputRange());
        HeadlessController.setLogMode(EmulatorLogFile.LogMode.BREAKPOINT);
        HeadlessController.setBreakpoints(breakpoints);
        HeadlessController.setInstMemSize(instMemSize);
        String logFile = String.format("%s_log.xml", assemblyFile);
```

```
        ExitTrapper.forbidSystemExitCall();
        int code = -1;
        try {
            HeadlessController.runFile(assemblyFile, logFile);
        } catch (ExitTrappedException e) {
            code = e.getCode();
        } finally {
            ExitTrapper.enableSystemExitCall();
            if (code != 0) {
                System.err.println("VisUAL emulator exited with error code " + code);
                System.exit(code);
            }
        }

        return OutputParser.parseOutput(logFile);
    }

    public static void run(String assemblyFile) {
        List<String> output = executeAndParseOutput(assemblyFile);

        System.out.println("---- PROGRAM OUTPUT ----");
        output.forEach(System.out::print);
        System.out.println("---- END PROGRAM OUTPUT ----");
    }
}
```

Dans ce code, InstructionWriter.outputBufferLength est défini avec la valeur suivante :

```
public static final int outputBufferLength = 0x1000;
```

(5) Utils.java :

```
import java.util.ArrayList;
import java.util.List;

public class Utils {
        public static String[] bytesToLittleEndianHex(byte[] data) {
        List<String> values = new ArrayList<>();

        for (int i = 0; i < data.length; i += 4) {
                int val = 0;
                for (int j = i; j < Math.min(i + 4, data.length); j++) {
                val += data[j] << 8 * (j - i);
                }
                values.add(String.format("0x%X", val));
        }

        if (values.isEmpty()) {
                values.add("0x0");
        }

        return values.toArray(String[]::new);
        }
}
```

(6) InstructionWriter.java contient les lignes suivantes pour convertir la chaîne de caractères *text* en une suite de valeurs hexadécimales :

```
String lbl = "_str_hello";
byte[] bytes = ArrayUtils.add("Hello, world!".getBytes(StandardCharsets.UTF_8), (byte)
0);
String[] values = Utils.bytesToLittleEndianHex(bytes);
writer.println(String.format("%s DCD %s", lbl, String.join(", ", values)));
// ce qui donne :
_str_hello DCD 0x6C6C6548, 0x77202C6F, 0x646C726F, 0x21
```

(7) Le code ARM de print pour l'affichage d'un string :

```
; ############ Print a string ############
_str_out       FILL           0x1000
_str_hello     DCD            0x6C6C6548, 0x77202C6F, 0x646C726F, 0x21
start          B              main

   ;R0 contains the address of the null-terminated string to print
print          STMFA         SP!, {R0-R2}
               LDR            R1, =_str_out  ; address of the output buffer
_print_loop           LDRB           R2, [R0], #1
                      STRB           R2, [R1], #1
                      TST            R2, R2
                      BNE            _print_loop
                      LDMFA         SP!, {R0-R2}
                      LDR          PC, [R13, #-4]!

   ;R0 contains the address of the null-terminated string to print
println               STMFA         SP!, {R0-R2}
                      LDR            R1, =_str_out  ; address of the output buffer
_println_loop         LDRB           R2, [R0], #1
                      STRB           R2, [R1], #1
                      TST            R2, R2
                      BNE            _println_loop
                      MOV            R2, #10
                      STRB           R2, [R1, #-1]
                      MOV            R2, #0
                      STRB           R2, [R1]
                      LDMFA         SP!, {R0-R2}
                      LDR          PC, [R13, #-4]!

main                  MOV            R0, #_str_hello
                      STR          PC, [R13], #4
                      BL             println

                      B              exit
exit                  END
```

(8) Le code assembleur pour l'opération de multiplication :

```
main                    MOV         R1, #-12
                        MOV         R2, #11
                        BL          mul
                        B           exit
mul                     STMFA       SP!, {R1,R2}
                        MOV         R0, #0
_mul_loop               LSRS        R2, R2, #1
                        ADDCS       R0, R0, R1
                        LSL         R1, R1, #1
                        TST         R2, R2
                        BNE         _mul_loop
                        LDMFA       SP!, {R1,R2}
                        LDR         PC, [R13, #-4]!
exit                    END
```

(9) Le code assembleur pour l'opération de division :

```
main            MOV         R1, #-12
                MOV         R2, #3
                BL          div
                B           exit
div             STMFA       SP!, {R2-R5}
                MOV         R0, #0
                MOV         R3, #0
                CMP         R1, #0
                RSBLT       R1, R1, #0
                EORLT       R3, R3, #1
                CMP         R2, #0
                RSBLT       R2, R2, #0
                EORLT       R3, R3, #1
                MOV         R4, R2
                MOV         R5, #1
_div_max        LSL         R4, R4, #1
                LSL         R5, R5, #1
                CMP         R4, R1
                BLE         _div_max
_div_loop       LSR         R4, R4, #1
                LSR         R5, R5, #1
                CMP         R4,R1
                BGT         _div_loop
                ADD         R0, R0, R5
                SUB         R1, R1, R4
                CMP         R1, R2
                BGE         _div_loop
                CMP         R3, #1
                BNE         _div_exit
                CMP         R1, #0
                ADDNE       R0, R0, #1
                RSB         R0, R0, #0
                RSB         R1, R1, #0
                ADDNE       R1, R1, R2
```

```
_div_exit       LDMFA          SP!, {R2-R5}
                LDR            PC, [R13, #-4]!
exit            END
```

(10) Le code assembleur pour la fonction intToString (ou itoa en C) :

```
main            LDR            R1, =-0x75BCD15 ; =-123456789 à transformer en String
_addr           FILL           12
                LDR            R2, =_addr
                STR            PC, [SP], #4
                B              its
its             STMFA          SP!, {R0-R10}
                MOV            R9, R2
                MOV            R3, R1
                MOV            R4, #10
                MOV            R8, #0
                MOV            R5, #0
                MOV            R6, #0
                MOV            R7, #0
                CMP            R3, #0
                MOV            R10, #0
                RSBLT          R3, R3, #0
                MOVLT          R5, #0x2D
                ADDLT          R8, R8, #8
                ADDLT          R10, R10, #1
_its_init       CMP            R3, R4
                BLT            _its_loop
                MOV            R1, R4
                MOV            R2, #10
                STR            PC, [SP], #4
                B              mul
                MOV            R4, R0
                B              _its_init
_its_loop       MOV            R1, R4
                MOV            R2, #10
                STR            PC, [SP], #4
                B              div
                MOV            R4, R0
                CMP            R4, #0
                BEQ            _its_exit
                MOV            R2, R0
                MOV            R1, R3
                STR            PC, [SP], #4
                B              div
                MOV            R1, R0
                ADD            R0, R0, #0x30
                LSL            R0, R0, R8
                ADD            R5, R5, R0
                MOV            R2, R4
                STR            PC, [SP], #4
                B              mul
                SUB            R3, R3, R0
                ADD            R8, R8, #8
```

```
                ADD             R10, R10, #1
                CMP             R8, #32
                MOVEQ   R8, #0
                BEQ             _its2
                B               _its_loop
_its2           MOV             R1, R4
                MOV             R2, #10
                STR             PC, [SP], #4
                B               div
                MOV             R4, R0
                CMP             R4, #0
                BEQ             _its_exit
                MOV             R2, R0
                MOV             R1, R3
                STR             PC, [SP], #4
                B               div
                MOV             R1, R0
                ADD             R0, R0, #0x30
                LSL             R0, R0, R8
                ADD             R6, R6, R0
                MOV             R2, R4
                STR             PC, [SP], #4
                B               mul
                SUB             R3, R3, R0
                ADD             R8, R8, #8
                ADD             R10, R10, #1
                CMP             R8, #32
                MOVEQ   R8, #0
                BEQ             _its3
                B               _its2
_its3           MOV             R1, R4
                MOV             R2, #10
                STR             PC, [SP], #4
                B               div
                MOV             R4, R0
                CMP             R4, #0
                BEQ             _its_exit
                MOV             R2, R0
                MOV             R1, R3
                STR             PC, [SP], #4
                B               div
                MOV             R1, R0
                ADD             R0, R0, #0x30
                LSL             R0, R0, R8
                ADD             R7, R7, R0
                ADD             R10, R10, #1
                MOV             R2, R4
                STR             PC, [SP], #4
                B               mul
                SUB             R3, R3, R0
                ADD             R8, R8, #8
                B               _its3
_its_exit       STMIA           R9, {R5-R7}
                LDMFA           SP!, {R0-R10}
                LDR             PC, [SP, #-4]!
```

Cette documentation vous a été rédigée par :

Dumitru Bulgaru ; Florent Caspar ; Yann Colomb ; Maxime Lucas ; Benjamin Sepe