IUT 'A' Paul SABATIER
Dpt Informatique

ASR => Architecture des ordinateurs : Ass2

Langage d'assemblage des microprocesseurs ARM

TRAVAUX DIRIGES

J.P. Carrara – R. Facca – P. Magnaud

6 mai 2010

TD 1
L'affectation

1 . Instructions de transfert

1.1 . Transfert entre registres

```
MOV Rd, Rs      @ Rd ← Rs
```

1.2 . Transfert de constante

```
MOV Rd, #cte    @ Rd ← #cte
```

Cette instruction permet de charger une constante dans l'intervalle [0..255].

1.3 . Transfert de mémoire à registre

1.3.1 . Transfert d'un mot

```
LDR Rd, opérandeMémoire  @ Rd ← opérandeMémoire
```

Dans cette instruction, `opérandeMémoire` désigne un mot en mémoire selon un des modes d'adressage vus en cours. L'adresse effective doit être un multiple de 4.

1.3.2 . Transfert d'un octet non signé

```
LDRB Rd, opérandeMémoire      @ Rd ← opérandeMémoire.B
```

Dans cette instruction, `opérandeMémoire` désigne un octet en mémoire selon un des modes d'adressage vus en cours. L'octet lu en mémoire est transféré dans l'octet de poids faible du registre destination. Les 3 octets de poids fort du registre sont mis à 0.

1.3.3 . Transfert d'un octet signé

```
LDRSB Rd, opérandeMémoire     @ Rd ← opérandeMémoire.B
```

Dans cette instruction, `opérandeMémoire` désigne un octet en mémoire selon un des modes d'adressage vus en cours. L'octet lu en mémoire est transféré dans l'octet de poids faible du registre destination. Les 3 octets de poids fort du registre sont mis à 0 si `opérandeMémoire` est positif et à 0xFFFFFF sinon.

1.3.4 . Transfert d'un demi-mot non signé

```
LDRH Rd, opérandeMémoire      @ Rd ← opérandeMémoire.H
```

Dans cette instruction, `opérandeMémoire` désigne un demi-mot en mémoire selon un des modes d'adressage vus en cours. L'adresse effective doit être un multiple de 2. Le demi-mot lu en mémoire est transféré dans le demi-mot de poids faible du registre destination. Le demi-mot de poids fort du registre est mis à 0.

1.3.5 . Transfert d'un demi-mot signé

```
LDRSH Rd, opérandeMémoire     @ Rd ← opérandeMémoire.H
```

Dans cette instruction, `opérandeMémoire` désigne un demi-mot en mémoire selon un des modes d'adressage vus en cours. L'adresse effective doit être un multiple de 2. Le demi-mot lu en mémoire est transféré dans le demi-mot de poids faible du registre destination. Le demi-mot de poids fort du registre est mis à 0 si `opérandeMémoire` est positif et à 0xFFFF sinon.

1.4 . Transfert de registre à mémoire

1.4.1 . Transfert d'un mot

```
STR Rd, opérandeMémoire        @ opérandeMémoire ← Rd
```

Dans cette instruction, `opérandeMémoire` désigne un mot en mémoire selon un des modes d'adressage vus en cours. L'adresse effective doit être un multiple de 4.

1.4.2 . Transfert d'un octet

```
STRB Rd, opérandeMémoire       @ opérandeMémoire.B ← Rd
```

Dans cette instruction, `opérandeMémoire` désigne un octet en mémoire selon un des modes d'adressage vus en cours. L'octet de poids faible du registre est écrit en mémoire

1.4.3 . Transfert d'un demi-mot

```
STRH Rd, opérandeMémoire       @ opérandeMémoire.H ← Rd
```

Dans cette instruction, `opérandeMémoire` désigne un demi-mot en mémoire selon un des modes d'adressage vus en cours. L'adresse effective doit être un multiple de 2. Le demi-mot de poids faible du registre est écrit en mémoire

2 . Traduction des instructions d'affectation algorithmiques

2.1 . Affectation d'une constante à une variable élémentaire

2.1.1 . Constante dans l'intervalle [0..255]

Soit à traduire l'affectation `i ← 10 ;`

On peut considérer 2 cas.

2.1.1.1 . La variable est un registre

La variable `i` est utilisée localement et est matérialisée par un registre, par exemple `R4`. Dans ce cas il n'y a pas de déclaration pour `i`. On peut écrire :

Algorithme	*Algorithme détaillé*	*Assembleur*
`i ← 10 ;`	`R4 ← #10 ;`	`MOV R4, #10`

2.1.1.2 . La variable est en mémoire

La variable `i` est utilisée plus globalement dans le programme. Elle doit donc être déclarée dans la section `.bss`. Supposons qu'elle occupe 32 bits (4 octets), la déclaration peut être :

```
      .bss
      .align 2
i:    .space 4
```

Il faut ensuite utiliser un registre intermédiaire, par exemple `R0`, de façon à transférer la constante `10` dans `R0` puis `R0` dans la variable `i`. Cependant, on ne peut pas désigner directement l'adresse de `i` dans l'instruction de transfert car il n'y a que des modes d'adressage indirects. Il faut donc passer par un **pointeur relais** alloué dans la section `.text` à une adresse proche de l'instruction de chargement et chargé dans un autre registre, par exemple `R1`.

La déclaration de ce pointeur relais peut être :

```
.Pi:   .word i
```

La traduction devient alors :

Algorithme	*Algorithme détaillé*	*Assembleur*

`i ← 10 ;`	`R0 ← #10 ;`
	`R1 ← .Pi ;`
	`R1↑ ← R0 ;`
		`MOV R0, #10`
		`LDR R1, .Pi`
		`STR R0, [R1]`

2.1.2 . Autre constante

Soit à traduire l'affectation `i ← 1234 ;`

On suppose que `i` est matérialisée par le registre `R5`. On ne peut pas utiliser le transfert de constante car celle ci est trop grande. Il faut allouer la constante en mémoire dans le section `.text`, à une adresse proche de l'instruction de chargement :

```
.C1:   .word 1234
```

La traduction est alors :

Algorithme	*Algorithme détaillé*	*Assembleur*
`i ← 1234 ;`	`R5 ← .C1 ;`	`LDR R5, .C1`

2.2 . Affectation entre variables élémentaires

L'ARM étant une architecture RISC, les transferts de mémoire à mémoire sont impossibles, il faut utiliser un registre intermédiaire.

Soit à traduire l'affectation `y ← x ;`

On suppose que `x` et `y` sont des variables globales 32 bits définies ainsi :

```
      .bss
      .align 2
x:    .space 4
y:    .space 4
```

Il faut utiliser deux pointeurs relais définis ainsi dans la section `.text` :

```
.Px:   .word x
.Py:   .word y
```

Il faut un registre pour charger les pointeurs relais, par exemple `R4`, et un registre intermédiaire, par exemple `R5`, ce qui donne :

Algorithme	*Algorithme détaillé*	*Assembleur*

`y ← x ;`	`R4 ← .Px ;`
	`R5 ← R4↑ ;`
	`R4 ← .Py ;`
	`R4↑ ← R5 ;`
		`LDR R4, .Px`
		`LDR R5, [R4]`
		`LDR R4, .Py`
		`STR R5, [R4]`

2.3 . Affectation avec un élément de tableau

On se limitera au cas de tableaux à une dimension.

Soit `t` un tel tableau et `s` la taille d'un de ces éléments. On peut considérer que `t` représente l'adresse mémoire du premier élément du tableau. Pour déterminer l'adresse de l'élément `t[i]`, il faut ajouter à `t` la taille de tous les éléments qui le précèdent. Si les indices commencent à `0`, il y a `i` éléments avant `t[i]` et cette taille est donc égale à `i` multiplié par `s`. On a donc :

Adresse `(t[i]) = t + i*s`

Il faut donc effectuer ce calcul pour accéder à un élément de tableau. En pratique, la taille d'un élément de tableau est souvent une puissance de 2 et la multiplication par la taille peut se faire par un décalage logique à gauche.

Par la suite, on se limitera à des tableaux dont les éléments sont des octets, des demi-mots ou des mots. Dans ce cas, on a bien :

Tableau d'octets

t+0	t[0]
t+1	t[1]
t+i	t[i]

Tableau de demi-mots

t+0	t[0]
t+2	t[1]
t+2i	t[i]

Tableau de mots

t+0	t[0]
t+4	t[1]
t+4i	t[i]

La déclaration d'un tableau `t1` de `N` octets peut se faire de la façon suivante :

```
t1:   .space N
```

La déclaration d'un tableau `t2` de `N` demi-mots peut se faire de la façon suivante :

```
      .align 1
t2:   .space N*2
```

La déclaration d'un tableau `t3` de `N` mots peut se faire de la façon suivante :

```
      .align 2
t3:   .space N*4
```

2.3.1 . L'indice est une constante

Dans ce cas, l'adresse est constante et le calcul peut être fait à l'assemblage du programme. On utilise donc le transfert d'une constante stockée en mémoire dans la section `.text`.

Soit l'affectation `v ← tab[5]` où `tab` est un tableau de demi-mots non signés. On défini la constante suivante :

```
.C1:   .word tab+5*2
```

En représentant `v` par le registre `R0`, la traduction est alors :

Algorithme	*Algorithme détaillé*	*Assembleur*
`v ← tab[5] ;`	`R0 ← .C1 ;` `R0 ← R0↑.H ;`	`LDR  R0, .C1` `LDRH R0, [R0]`

2.3.2 . L'indice n'est pas une constante

Dans ce cas, l'adresse ne peut être calculée que lors de l'exécution du programme.

Soit à traduire l'affectation `v ← tab[i]` pour laquelle `v` est le registre `R0`, l'adresse de `tab` est dans le registre `R1` et la valeur de `i` dans le registre `R2`.

On va distinguer 3 cas en fonction du format des éléments du tableau.

2.3.2.1 . Tableau d'octets

Dans ce cas, il n'y a pas de multiplication par la taille.

La traduction est alors :

Algorithme	*Algorithme détaillé*	*Assembleur*
`v ← tab[i] ;`	`R0 ← (R1+R2)↑.B ;`	`LDRB R0, [R1, R2]`

2.3.2.2 . Tableau de demi-mots

Dans ce cas, l'indice doit être multiplié par **2**, à l'aide d'un décalage logique à gauche. Si on ne veut pas modifier l'indice, il faut utiliser un autre registre, par exemple `R3`.

La traduction est alors :

Algorithme	*Algorithme détaillé*	*Assembleur*
`v ← tab[i] ;`	`R3 ← DLG(1,R2) ;` `R0 ← (R1+R3)↑.H ;`	`MOV R3, R2, LSL #1` `LDRH R0, [R1, R3]`

2.3.2.3 . Tableau de mots

Dans ce cas, l'indice doit être multiplié par **4** par un décalage logique de 2 positions à gauche. Contrairement au cas précédent, le décalage peut être fait dans l'instruction `LDR`, donc il n'est pas nécessaire d'utiliser un autre registre.

La traduction est alors :

Algorithme	*Algorithme détaillé*	*Assembleur*
`v ← tab[i] ;`	`R0 ← (R1+DLG(2,R2))↑ ;`	`LDR R0, [R1, R2, LSL #2]`

2.4 . Affectation avec un élément d'enregistrement

Soit le type enregistrement suivant défini en langage algorithmique :

```
type Date : enregistrement
  jour <Octet>,
  mois <Octet>,
  année <DemiMot> ;
```

Pour accéder à un champs, il faut définir un déplacement égal à la taille de tous les champs qui le précèdent. Pour le type **Date**, on peut donc définir les déplacements de la façon suivante :

```
.equiv jour, 0
.equiv mois, 1
.equiv annee, 2
```

Soit `d` une variable du type `Date`. On peut la déclarer de la façon suivante :

```
      .align 2
d:    .space 1+1+2
```

Soit à traduire l'affectation `d.mois ← 4 ;`

Il faut utiliser un pointeur relais proche dans la section `.text` déclaré ainsi :

```
.Pd:   .word d
```

En utilisant `R0` comme pointeur et `R1` comme registre intermédiaire, la traduction est alors :

Algorithme	*Algorithme détaillé*	*Assembleur*

`d.mois ← 4 ;`	`R0 ← .Pd`
	`R1 ← #4 ;`
	`(R0+#mois)↑.B ← R1;`
		`LDR R0, .Pd`
		`MOV R1, #4`
		`STRB R1, [R0, #mois]`

3 . Exercices

Pour chaque exercice, donner les déclarations nécessaires, l'algorithme détaillé et la traduction en langage d'assemblage de l'affectation donnée en langage algorithmique.

3.1 . Variables élémentaires

Soit `n1`, `n2`, `n3` et `n4` des variables au format octet et `v1` et `v2` des variables au format mot.

Les affectations à traiter sont :

`n1 ← 8 ;`

`n2 ← (F0)`$_{16}$ `;`

`n3 ← 'A' ;`

`n4 ← (1111)`$_{2}$ `;`

`v1 ← (307)`$_{8}$ `;`

`v2 ← (A5A5A5A5)`$_{16}$ `;`

3.2 . Tableaux

Les affectations à traiter sont :

```
t1[i] ← t2[j] ;
```

Où `t1` et `t2` sont deux tableaux de mots, `i` un mot et `j` un demi-mot.

```
t[tab[k]] ← 0 ;
```

Où `t` est un tableau d'octets, `tab` un tableau de mots et `k` un mot.

```
p↑ ← tp[l]↑ ;
```

Où `p` est un mot contenant un pointeur sur un octet, `tp` un tableau de mots contenant des pointeurs sur un octet et `l` un mot.

3.3 . Enregistrement

Soit les déclarations en langage algorithmique :

```
type TabOctets : tableau [1..10] de <Octet> ;
type TabMots : tableau [1..3] de <Mot> ;
type Etudiant : enregistrement
            num  <Mot>,
            nom  <TabOctets>,
            notes <TabMots> ;
etu <Etudiant> ;
```

Les affectations à traiter sont :

```
etu.num ← cpt ;
```

Où `cpt` est matérialisée par un registre.

```
etu.nom[i] ← '-' ;
```

Où `i` est matérialisée par un registre.

```
etu.notes[j] ← 10 ;
```

Où `j` est matérialisée par un registre.

TD 2
Structures de contrôle

1. Tests

1.1. Rappels sur les indicateurs

Les indicateurs sont positionnés par les instructions de comparaison. On peut toutefois **explicitement** demander, par le biais d'un suffixe au code opération (**S**), leur positionnement lors des opérations arithmétiques et logiques et lors des transferts entre registres.

Indicateur	Instruction arithmétique ou de comparaison	Instruction logique	Instruction de transfert
C : **C**arry	retenue (débordement des naturels), C = 1 si retenue, C = 0 sinon	Positionné après une opération de décalage s'il y a un 1 sortant	Pas de sens
V : o**V**erflow	dépassement de capacité (débordement des relatifs), V = 1 si débordement, V = 0 sinon	Pas de sens	Pas de sens
Z : **Z**ero	Zéro, Z = 1 si le résultat est nul, Z = 0 sinon	Positionné si tous les bits du résultat sont à zéro.	Positionné si tous les bits du résultat sont à zéro.
N : **N**egative	Négatif, N = 1 si le résultat est négatif, N = 0 sinon	Pas de sens	Recopie du 31^{éme} bit de l'opérande (bit de poids fort)

1.2. Instructions de comparaison

1.2.1. Comparaison : *CMP*

Syntaxe	*Sémantique*
CMP Rn,op2 [1]	La machine réalise Rn - op2 et positionne les indicateurs

1.2.2. Identité : *TEQ*

Syntaxe	*Sémantique*
TEQ Rn,op2 [2]	La machine réalise Rn XOR op2 et positionne l'indicateur Z en cas d'égalité.

1 Op2 est soit une valeur immédiate, soit un registre, soit un registre décalé.

2 Op2 est soit une valeur immédiate, soit un registre, soit un registre décalé.

2. Exécution conditionnelle

L'exécution de la plupart des instructions peut être conditionnée par la réalisation d'une condition simple (l'un des indicateurs du registre CSPR) ou d'une condition de relation plus complexe (cf cours d'architecture) résultant d'une combinaison des indicateurs. Ce conditionnement apparaît sous la forme d'un suffixe optionnel du code opération.

2.1. Conditionnement simple

Condition d'exécution	Suffixe
N = 0	PL
N = 1	MI
Z = 0	NE
Z = 1	EQ
V = 0	VC
V = 1	VS
C = 0	CC
C = 1	CS

2.2.2. Conditionnement après comparaison de 2 naturels

Condition d'exécution	Suffixe
op1 < op2	LO
op1 ≤ op2	LS
op1 ≥ op2	HS
op1 > op2	HI
op1 = op2	EQ
op1 ≠ op2	NE

2.2.3. Conditionnement après comparaison de 2 relatifs

Condition d'exécution	Suffixe
op1 < op2	LT
op1 ≤ op2	LE
op1 ≥ op2	GE
op1 > op2	GT
op1 = op2	EQ
op1 ≠ op2	NE

Exemple : Conditionnement simple

Syntaxe	*Sémantique*
MOV R4,#10	R4 ← 10
MOVCC R4,#10	R4 ← 10 ⇔ C = 0

3. Les instructions de branchement

3.1. Branchement général : *B*

Syntaxe	*Sémantique*
B étiquette	PC ← etiquette

3.2. Branchements conditionnels

Les instructions de branchements conditionnels sont induites de l'instruction de branchement général à laquelle est adjointe un des suffixes d'exécution conditionnelle vus précédemment. Les indicateurs auront du être préalablement positionnés.

Exemple : On considère que R4 et R5 contiennent des entiers relatifs.

Syntaxe	*Sémantique*
CMP R4,R5	Z, N, C, V ← R4 **::** R5
BGT etiquette	PC ← etiquette ⇔ R4 > R5

3.3. Branchements spéciaux

Appels de procédures, retours de procédure, retour de procédure d'exception, déroutement, …. Tous ces branchements seront vus plus tard.

4. Traduction des structures algorithmiques

4.1. Structures portant sur des conditions simples

Algorithme	*Assembleur*
tantque Cond **faire** … séquence … **fintantque**	**.Ltq**: test branchement en **.Lftq** si test faux … séquence … **B .Ltq** **.Lftq**:
si Cond **alors** … séquence … **finsi ;**	**.Lsi**: test branchement en **.Lfsi** si test faux … séquence … **.Lfsi**:

Algorithme	*Assembleur*
si Cond **alors** … séquence 1 … **sinon** … séquence 2 … **finsi ;**	**.Lsi**: test branchement en **.Lsin** si test faux … séquence 1 … **B .Lfsi** **.Lsin**: … séquence 2 … **.Lfsi**:

Remarque : Le test (évaluation de la condition) peut être une instruction de test ou de comparaison (cas général), mais aussi toute instruction positionnant les indicateurs du CSPR.

4.2. Structures portant sur des conditions multiples

On effectue ce qu'on appelle un court-circuit de l'évaluation. Ce concept a déjà été abordé dans le module A.P. lorsqu'on stipule que les opérateurs 'et' et 'ou' ne sont pas commutatifs. En effet une conjonction de conditions est fausse dès que l'une des conditions est fausse. Réciproquement une disjonction de conditions est vraie dès que l'une des conditions est vraie. L'évaluation des conditions , suivant la condition qui a forcé le résultat, devient inutile. C'est ce principe qui est mis en oeuvre dans les schémas de traduction suivants :

Algorithme	*Assembleur*
tantque (C1 **ou** C2 … **ou** Cn) **faire** … séquence … **fintantque ;**	**.Ltq**: test 1 branchement en **.Lfre** si test vrai test 2 branchement en **.Lfre** si test vrai … test n branchement en **.Lftq** si test faux **.Lfre**: … séquence … **B .Ltq** **.Lftq**:
tantque (C1 **et** C2 … **et** Cn) **faire** … séquence … **fintantque ;**	**.Ltq**: test 1 branchement en **.Lftq** si test faux test 2 branchement en **.Lftqe** si test faux … test n branchement en **.Lftq** si test faux … séquence … **B .Ltq:** **.Lftq**:

Algorithme	*Assembleur*
si Condition multiple **alors** … séquence … **finsi ;**	Partie identique au **tantque** dans laquelle on substitue : **.Ltq** par **.Lsi** (facultatif) **.Lftq** par **.Lfsi** **.Lfre** par **.Lalo** **.Lalo**: … séquence … **.Lfsi**:
si Condition multiple **alors** … séquence 1 … **sinon** … séquence 2 … **finsi ;**	Partie identique au **tantque** dans laquelle on remplace **.Ltq** par **.Lsi** (facultatif) **.Lftq** par **.Lsin** **.Lfre** par **.Lalo** **.Lalo**: … séquence 1 … **B** **.Lfsi** **.Lsin**: … séquence 2 … **.Lfsi**:

5. Exercices

Donner les algorithmes détaillés et les traductions en assembleur correspondant aux algorithmes suivants :

5.1. Conditions simples

Algorithme	*Algorithme détaillé*	*Assembleur*
i ← 0 **tantque** i < 7 **faire** … Séquence … **fintantque ;**		

I : entier relatif codé sur un demi-mot mémoire

5.2. Conditions multiples

Soit l'*algorithme* :

tantque a>b **faire**
…
séquence
…
fintantque ;

a et b sont deux entiers naturels codés sur 64 bits (8 octets) et sont déclarés de la façon suivante :

```
        .bss
        .align  2
a:      .space  8
b:      .space  8
```

Attention : les nombres sont représentés en 'little endian'

La comparaison directe des 2 nombres au format 64 bits n'est pas possible. Il faut procéder par tranches de 32 bits.

Donner l'algorithme détaillé et la traduction en assembleur correspondant :

Algorithme détaillé	*Assembleur*

5.3. Exécution conditionnée

Soit l'*algorithme* :

si a<b **alors**
    min ← a ;
sinon
    min ← b ;
finsi ;

a , b et min sont des entiers naturels en mémoire

Donner l'algorithme détaillé et la traduction en assembleur correspondant :

- en utilisant des branchements
- en utilisant l'exécution conditionnelle

Algorithme détaillé	*Assembleur*

TD 3
Opérations de décalages, arithmétiques et logiques

Préambule : Dans tout le T.D. lorsque le texte fait référence à op2 comme dernier opérande celui-ci peut être soit une valeur immédiate, soit un registre, soit un registre décalé.

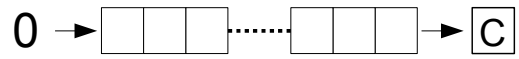
1. Décalages

1.1. Définition des opérations de décalage et règles générales.

Pour les définitions formelles, voir les Tds d'architecture. Les processeurs de la famille ARM ne disposent pas à proprement parler d'instructions de décalages. Par contre l'architecture du processeur est telle que le dernier opérande (celui noté op2) des opérations de transfert entre registres ou des opérations arithmétiques et logiques peut être décalé. **L'indicateur Carry du CPSR n'est positionné que dans le cas d'une instruction MOV avec opérande décalé et suffixe S spécifié (cf td N°2)**. Les figures suivantes donnent la sémantique d'un décalage d'une position.

1.1.1. Décalages logiques

V ← DLD(V)

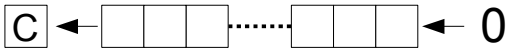


V ← DLG(V)

1.1.2. Décalages arithmétiques

V ← DAD(V)

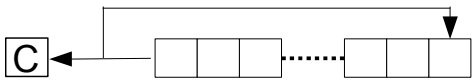
V ← DAG(V)



1.1.3. Décalages cycliques

V ← DCD(V)

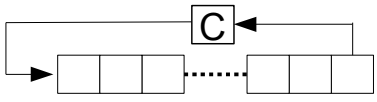
V ← DCG(V)



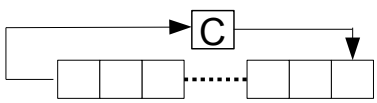
1.1.4. Décalages cycliques avec extension

Ces opérations permettent d'effectuer des décalages sur plusieurs tranches.

V ← DCXD(V)



V ← DCXG(V)



1.2. Instructions de décalage et mise en oeuvre au sein d'une instruction

Décalage	Code opération.	Décalage	Code opération
DLD	: LSR	DAD	: ASR
DLG	: LSL	DCD	: ROR
DAG	: LSL	DCXD	: RRX

Ne nous intéressant qu'à l'instruction de transfert, les syntaxes sont les suivantes :

MOV Rd,Rs1,InstDécal #Imm
MOV Rd,Rs1,InstDécal Rs2

Où #Imm représente une valeur immédiate et Rs2 un registre.

Les décalages cycliques à gauche de n positions sont synthétisés par des décalages cycliques à droite de 32 – n positions.

2. Opérations arithmétiques

2.1. Additions

2.1.1. Addition simple : ADD

Syntaxe	*Sémantique*
ADD Rd,Rs1,op2	Rd ← Rs1 + op2

2.1.2. Addition avec retenue : *ADC*

Syntaxe	*Sémantique*
ADC Rd,Rs1,op2	Rd ← Rs1 + op2 + Carry

Remarque : Cette instruction permet de faire des additions sur plusieurs tranches (cf. cours).

2.2. Soustractions

Attention : lors de l'opération de soustraction, la retenue est positionnée à l'inverse de ce qu'elle l'est alors que l'opération est effectuée à la main : 1 ⇒ pas de débordement, 0 ⇒ débordement.

2.2.1. Soustraction directe : SUB

Syntaxe	*Sémantique*
SUB Rd,Rs1,op2	Rd ← Rs1 - op2

2.2.2. Soustraction directe avec retenue : SBC

Syntaxe	*Sémantique*
SBC Rd,Rs1,op2	Rd ← Rs1 – op2 + Carry - 1

2.2.3. Soustraction inversée : RSB

Syntaxe	*Sémantique*
RSB Rd,Rs1,op2	Rd ← op2 - Rs1

2.2.4. Soustraction inversée avec retenue : RSC

Syntaxe	*Sémantique*
RSC Rd,Rs1,op2	Rd ← op2 - Rs1 + Carry - 1

2.3. Opposé d'un nombre relatif.

Synthétisé à partir de RSB

Syntaxe	*Sémantique*
RSB Rd,Rs1,#0	Rd ← - Rs1

2.4. Multiplications.

2.4.1. Multiplications avec résultat sur 1 registre (32 bits)

Ces multiplications opèrent indifféremment sur des opérandes signés ou non signés. C'est l'utilisateur qui est chargé de l'interprétation du résultat. Seuls les bits N et Z du CPSR sont positionnés. Attention les registres Rd et Rs1 doivent être différents. La multiplication étant commutative on peut permuter Rs1 avec Rs2 pour supprimer un éventuel problème.

2.4.1.1. Multiplication simple

Syntaxe	*Sémantique*
MUL Rd,Rs1,Rs2	Rd ← Rs1 * Rs2

2.4.1.2. Multiplication avec accumulation

Cette opération permet le calcul de filtres, le calcul de transformée de Fourrier ou la mise en œuvre du schéma de Horner, ...

Syntaxe	*Sémantique*
MLA Rd,Rs1,Rs2,Rs3	Rd ← (Rs1 * Rs2) + Rs3

2.4.2. Multiplications avec résultat sur 2 registres (64 bits)

Ces multiplications opèrent cette fois ci sur des opérandes de nature spécifiques. C'est un préfixe (noté Θ dans la syntaxe et égal à U pour les naturels ou à S pour les relatifs) qui précise l'opération utilisée et de ce fait la nature des opérandes. Comme précédemment seuls les bits N et Z du CPSR sont positionnés. Attention les registres RdHi, RdLo et Rs1 doivent tous être différents..

2.4.2.1. Multiplication simple

Syntaxe	*Sémantique*
ΘMULL RdLo,RdHi,Rs1,Rs2	RdHi:RdLo ← Rs1 * Rs2

2.4.2.2. Multiplication avec accumulation

Syntaxe	*Sémantique*
ΘMLAL RdLo,RdHi,Rs1,Rs2	RdHi:RdLo ← (Rs1 * Rs2) + RdHi:RdLo

3. Opérations logiques

3.1. Produit logique : AND

Syntaxe	*Sémantique*
AND Rd,Rs1,op2	Rd ← Rs1 ∧ op2

3.2. Somme logique : *OR*

Syntaxe	*Sémantique*
ORR Rd,Rs1,op2	Rd ← Rs1 ∨ op2

3.3. Union exclusive : *EOR*

Syntaxe	*Sémantique*
EOR Rd,Rs1,op2	Rd ← Rs1 ⊕ op2

3.4. Complément logique : *MVN*

Syntaxe	*Sémantique*
MVN Rd,op2	Rd ← NOT op2

4. Divers

4.1. Comptage des bits à zéro en tête.

Syntaxe	*Sémantique*
CLZ Rd,Rs	Rd ← 31-n | $2^n \leq Rs < 2^{n+1} \Leftrightarrow Rs \neq 0$ Rd ← 32 ⇔ Rs = 0

4.2. Forçage à zéro de bit.

Syntaxe	*Sémantique*
BIC Rd,Rs,op2	Rd ← Rs AND NOT op2

5. Exercices

5.1. Recherche d'un élément dans un tableau

1. Écrire l'algorithme de recherche du rang d'un élément donné dans un tableau. On appellera tab le tableau, n sa longueur, ele l'élément à rechercher, trouvé l'indicateur du résultat de la recherche et rang la position éventuelle de l'élément dans tab.
2. Écrire l'algorithme détaillé et la traduction en assembleur, en considérant tab comme un tableau d'entiers relatifs représentés sur 32 bits, n et rang comme des variables au format mot et trouvé comme une variable au format octet.

5.2. Calcul de la division binaire d'entiers naturels

1. Écrire l'algorithme de division binaire sachant que :

   R0 contient le dividende qui est un entier naturel,
   R1 contient le diviseur qui est un entier naturel.

   Après le traitement :

   R0 contient le reste,
   R2 contient le quotient.

   On commencera par faire une trace de la division de 35 par 13 avec des opérandes codés sur 8 bits afin de formaliser le problème.

1	On commence par aligner le diviseur avec le dividende en effectuant des décalages à gauche du diviseur afin que les « 1 » de poids fort du diviseur et du dividende coïncident (dans l'exemple de la trace on doit effectuer **2** décalages pour que le 1 de poids fort du 13 soit aligné avec le 1 de poids fort du 35). Si après ce premier alignement le diviseur est plus petit que le dividende alors on injecte un 1 dans le quotient et on soustrait le diviseur au dividende (ce résultat devient le prochain dividende) sinon on ne fait rien (le dividende reste tel qu'il est) Par la suite, on devra effectuer autant de « décalage à droite-soustraction» du diviseur que de décalages initiaux nécessaires à son alignement. (Dans notre cas **2**.). Méthode générale : *Après décalage, lorsque le diviseur est plus grand que le dividende on injecte un 0 dans le quotient. Lorsqu'il est inférieur ou égal on injecte un 1 et on soustrait le diviseur du dividende*
2	On effectue le **premier** « décalage à droite-soustraction». Cette fois-ci il n'y a pas de débordement lors de la soustraction. On injecte un 1 dans le quotient, on soustrait le diviseur du dividende et on décale le diviseur.
3	On effectue le **deuxième** « décalage à droite-soustraction».. Cette fois-ci il y a débordement. On injecte un 0 dans le quotient, on conserve le dividende d'avant soustraction.

2. Écrire l'algorithme détaillé et la traduction en assembleur.

TD4
Les sous-programmes

Exercice 1 :

Soit la procédure **`libererChaine`** définie dans le TP de Compléments de C sur les chaînes de caractères dynamiques et dont le prototype en C est le suivant :

void `libererChaine`(ChaineDyn * *ch) ;*

Cette procédure libère le bloc d'octets alloué à la chaîne ***ch*** en mémoire dynamique et réinitialise ***ch*** en chaîne vide. Son algorithme est le suivant :

procédure `libererChaine` (**mise-à-jour** ch <ChaineDyn>)

Glossaire

libérer <fonction> : utiliser la fonction **`free`** de la bibliothèque C

début

libérer(ch.ptrCar) ;

ch.nbCar ← 0 ;

ch.ptrCar ← **NULL** ;

fin

1°)- Définir en langage d'assemblage ARM, le fichier **`defTypes.s`** qui contiendra les déclarations de la constante symbolique **NULL** et du type enregistrement suivant :

```
typedef  char *  PointeurCar;

typedef  struct
{
  int  nbCar;                 /* nombre de caractères de la chaîne */
  PointeurCar   ptrCar;       /* pointeur sur le tableau de caractères contenant la suite de
                                 caractères ASCII constituant la chaîne */
} ChaineDyn;
```

2°)- Ecrire l'algorithme détaillé et le glossaire de ce S/P sachant qu'il est appelé par le programme **`testChaine`** en langage C.

3°)- Donner le codage de ce S/P en langage d'assemblage ARM, sous forme d'une unité de compilation complète.

1°)- Fichier **`defTypes.s`** (à inclure en tête des S/P)

@ définition de symboles

@ définition de l'enregistrement du type <ChaineDyn>

2°)- et 3°)-

Algorithme détaillé	*Assembleur*
glossaire	
début	
fin	

Exercice 2 :

Soit la procédure **`copierChaine`** définie dans le TP de Compléments de C sur les chaînes de caractères dynamiques et dont le prototype en C est le suivant :

void `copierChaine` (**ChaineDyn *** *ch1*, **const ChaineDyn** *ch2* , **jmp_buf** *ptRep*);

Cette procédure affecte la chaîne *ch2* à la chaîne *ch1*. Elle lève l'exception ″ERR_ALLOC″ en cas d'erreur d'allocation. Si une exception est levée la chaîne *ch1* ne sera pas modifiée.

Son algorithme est le suivant :

procédure `copierChaine` (mise-a-jour ch1 <ChaineDyn>, entrée ch2 < ChaineDyn >)

Glossaire

lgCh2 <Entier> : longueur de la chaîne ch2 ;

ptrAlloc <PointeurCar> : pointeur sur le bloc d'octets alloué dynamiquement;

copierBlocOctets <fonction> : utiliser la fonction **`memcpy`** de la bibliothèque C

début

```
lgCh2 ← ch2.nbCar;
si ( lgCh2 = 0 ) alors
        ptrAlloc ← NULL;
sinon
        ptrAlloc ← allouer( lgCh2 );
        si ( ptrAlloc = NULL ) alors
                lever ( ERR_ALLOC );
finsi;
        copierBlocOctets( ptrAlloc, ch2.ptrCar, lgCh2 );
finsi;
si ( ch1.ptrCar!= NULL ) alors
        libérer (ch1.ptrCar);
finsi;
ch1.nbCar ←  lgCh2;
ch1.ptrCar ←  ptrAlloc;
```

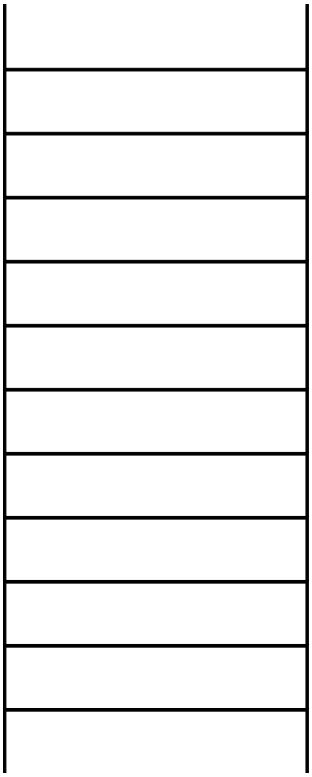
fin

1°)- Ecrire l'algorithme détaillé et le glossaire de ce S/P sachant qu'il est appelé par le programme **`testChaine`** en langage C.

2°)- Donner le codage de ce S/P en langage d'assemblage ARM, sous forme d'une unité de compilation complète.

Organisation de la pile système en début du S/P :

on utilisera le modèle proposé au chapitre 3-§ 4.4 du cours.



glossaire de l'algorithme détaillé

paramètres :

ch1 : position relative du paramètre ch1 = , passage par ;
ch2 : position relative du paramètre ch2 = , passage par ;
ptRep: position relative du paramètre ptRep = , passage par ;

variables locales :

lgCh2 : position relative de la variable locale lgCh2 =
ptrAlloc : position relative de la variable locale ptrAlloc =

Algorithme détaillé	*Assembleur*
début	
<u>fin</u>	