

Sommaire

LE MODELE DE MEMOIRE	1	<i>Adressage "base + déplacement"</i>	<i>4</i>
LES REGISTRES	1	<i>Transfert d'octets</i>	<i>4</i>
LA MEMOIRE	2	<i>Adressage de pile</i>	<i>5</i>
LE JEU D'INSTRUCTIONS ARM	2	EXECUTION CONDITIONNELLE	5
INSTRUCTIONS DE TRAITEMENT DE DONNEES	2	INSTRUCTIONS DE CONTROLE DE FLOT	6
Opérandes	2	Branchements	6
Opérandes registres	2	Branchements conditionnels	6
Opérandes immédiats (constantes)	2	Branchement avec lien de retour	6
Opérandes registres décalés	2	SYNTAXE D'UN FICHIER SOURCE (GNU GAS)	6
Opérations arithmétiques	3	Structure d'une ligne de code	6
Opérations logiques	3	Commentaires	6
Mouvements entre registres	3	Directives pour l'assembleur	7
Comparaisons	3	PSEUDO-INSTRUCTIONS : ADR ET LDR	7
Mise à jour des codes condition	4	Initialisation d'un registre avec une adresse (ADR)	7
INSTRUCTIONS D'ACCES A LA MEMOIRE	4	Initialisation d'un registre avec une constante (LDR)	7
Adressage indirect par registre	4		

Le modèle de mémoire

Les registres

Pour une programmation en mode utilisateur, le processeur ARM dispose de :

- **15 registres** de 32 bits, à usage général, désignés par **r0 à r14**
- un registre de 32 bits contenant le **compteur de programme**, désigné par **r15**
- un **registre d'état**, désigné par **CPSR** (*Current Program Status Register*) dont voici la structure :

31	28	27					8	7	6	5	4	0
N	Z	C	V	bits inutilisés				IF		T	mode	

Les bits de poids faible définissent le mode de fonctionnement, le jeu d'instructions et les activations d'interruptions (voir plus loin). Les **codes conditions** sont représentés par les bits de poids fort et ont les significations suivantes :

- **N** (*Negative*) : reflète le bit 31 du résultat de la dernière opération qui a positionné les codes conditions. Si ce résultat doit être considéré comme un nombre signé, N=1 indique que le résultat est négatif, N=0 dit qu'il est positif.
- **Z** (*Zero*) : Z=1 indique que la dernière opération qui positionné les codes conditions a produit un résultat nul
- **C** (*Carry*) : représente la retenue de la dernière opération qui a positionné les codes conditions. Si cette opération est une addition, C=1 indique que l'addition a généré une retenue sortante. Si c'est une soustraction, C=0 dit que la soustraction a généré une retenue sortante. Pour d'autres opérations dont le second opérande est un registre décalé, C reflète le dernier bit éjecté lors du décalage.
- **V** (*oVerflow*) : V=1 indique que la dernière opération qui a positionné les codes conditions a produit un débordement.

La mémoire

La mémoire est vue comme un tableau linéaire d'octets numérotés de 0 à $2^{32}-1$. Les données manipulées sont des **octets** (8 bits), des **demi-mots** (16 bits) ou des **mots** (32 bits). Les mots sont toujours alignés sur des limites de 4 octets (leur adresse est toujours multiple de 4) et les demi-mots sont alignés sur des limites de 2 octets (adresse multiple de 2).

La mémoire est gérée selon le mode *little endian* : l'octet de poids faible est rangé en tête, c'est-à-dire à l'adresse précédant celle de l'octet de poids juste supérieur.

Sur la figure ci-dessous, les adresses (en décimal) sont notées dans le coin inférieur droit des cellules mémoire.

AA <i>0</i>	BB <i>1</i>	CC <i>2</i>	DD <i>3</i>
33 <i>4</i>	22 <i>5</i>	11 <i>6</i>	00 <i>7</i>
78 <i>8</i>	56 <i>9</i>	34 <i>10</i>	12 <i>11</i>

octet **0xBB** à l'adresse 1
octet **0xCC** à l'adresse 2
demi-mot **0x2233** à l'adresse 4
demi-mot **0x0011** à l'adresse 6
mot **0x12345678** à l'adresse 8

Le jeu d'instructions ARM

L'architecture ARM est de type *load-store* : seules certaines instructions peuvent accéder à la mémoire et transférer une valeur depuis la mémoire vers un registre (*load*) ou inversement (*store*). Toutes les autres instructions opèrent sur des registres.

Instructions de traitement de données

Opérandes

Opérandes registres

Une opération ARM typique de traitement de données est écrite en langage d'assemblage comme suit :

```
ADD          r0,r1,r2          @ r0 ← r1 + r2
```

Le @ indique que tout ce qui suit est un commentaire. Les registres r1 et r2 sont les opérandes en entrée. L'instruction demande au processeur d'ajouter les contenus de ces deux registres et de ranger le résultat dans le registre r0.

Attention à l'ordre des opérandes : le 1er registre est celui où le résultat doit être rangé, les 2 registres suivants sont les opérandes en entrée (1er, puis 2nd opérande)

Opérandes immédiats (constantes)

Pour certaines opérations, un des deux opérandes peut être une valeur (constante) au lieu d'un registre. Cette constante est indiquée par un #. Par exemple :

ADD	r3,r3,#1	@ r3 ← r3 + 1	
AND	r8,r7,#0xFF	@ r8 ← r7 _[7:0]	@ 0x pour une constante en hexa @ 0b pour une constante binaire

Comme la valeur immédiate doit être spécifiée dans les 32 bits du code de l'instruction, il n'est pas possible de donner n'importe quelle valeur sur 32 bits comme opérande immédiat. Dans le jeu d'instruction ARM, une valeur immédiate doit être codée sur 12 bits. Les valeurs possibles ont au plus 8 bits à 1, placés dans 8 positions adjacentes alignées sur une frontière de deux bits, c'est-à-dire des valeurs de la forme :

$(0 \text{ à } 255) \times 2^{2n}$, avec $0 \leq n < 16$ ([0-255] codé sur 8 bits, n codé sur 4 bits).

NB : on peut contourner la difficulté grâce à la pseudo-instruction LDR (voir page 7).

Opérandes registres décalés

Il est possible de faire subir un décalage au second opérande avant de lui appliquer une opération. Par exemple :

```
ADD      r3,r2,r1,LSL #3 @ r3 ← r2 + r1*8
```

Ici, `r1` subit un décalage logique vers la gauche de 3 positions.

Les décalages possibles sont :

LSL	logical shift left	décalage de 0 à 31 positions vers la gauche avec introduction de 0
LSR	logical shift right	décalage de 0 à 32 positions vers la droite avec introduction de 0
ASL	arithmetic shift left	identique à LSL
ASR	arithmetic shift right	décalage de 0 à 32 positions vers la droite avec extension du bit de signe
ROR	rotate right	décalage de 0 à 32 positions circulaire vers la droite
RRX	rotate right extended	décalage d'une position vers la droite avec introduction du bit C

Le nombre de positions de décalage peut être spécifié par une constante (précédée de #) ou par un registre (octet de poids faible). Par exemple :

```
ADD    r5,r5,r3,LSL r2    @ r5 ← r5 + r3*2r2
```

Opérations arithmétiques

Ces instructions réalisent des additions, soustractions et soustractions inverses (c'est l'ordre des opérandes qui est inversé) sur des opérandes 32 bits. La retenue C, quand elle est utilisée, est la valeur du bit C du registre CPSR.

ADD	r0,r1,r2	@ r0 ← r1 + r2	addition
ADC	r0,r1,r2	@ r0 ← r1 + r2 + C	addition with carry
SUB	r0,r1,r2	@ r0 ← r1 - r2	subtract
SBC	r0,r1,r2	@ r0 ← r1 - r2 + C - 1	subtract with carry
RSB	r0,r1,r2	@ r0 ← r2 - r1	reverse subtract
RSC	r0,r1,r2	@ r0 ← r2 - r1 + C - 1	reverse subtract with C
MUL	r4,r3,r2	@ r4 ← (r3 * r2) _[31:0]	multiplication
MLA	r4,r3,r2,r1	@ r4 ← (r3 * r2 + r1) _[31:0]	multiplication-addition

NB pour les multiplications :

- le second opérande ne peut pas être une valeur immédiate
- le registre résultat ne peut pas être identique au premier registre opérande
- si le bit S est positionné (mise à jour des codes condition), le code V n'est pas modifié et le code C est non significatif (cf. page 4 : Mise à jour des codes conditions)

Multiplier entre eux deux entiers sur 32 bits donne un résultat sur 64 bits : les 32 bits de poids faible sont placés dans le registre résultat, les bits de poids fort sont ignorés.

Opérations logiques

Ces opérations sont appliquées pour chacun des 32 bits des opérandes.

AND	r0,r1,r2	@ r0 ← r1 et r2	and
ORR	r0,r1,r2	@ r0 ← r1 ou r2	or
EOR	r0,r1,r2	@ r0 ← r1 ouexcl r2	exclusive-or
BIC	r0,r1,r2	@ r0 ← r1 et non r2	bit clear

Mouvements entre registres

Ces instructions n'ont pas de premier opérande en entrée (il est omis) et copient le second opérande vers la destination.

MOV	r0,r2	@ r0 ← r2	move
MVN	r0,r2	@ r0 ← non r2	move negated

Comparaisons

Ces instructions ne produisent pas de résultat et ne font que positionner les codes condition dans le CPSR.

CMP	r1,r2	@ CPSR ← cc (r1 - r2)	compare
CMN	r1,r2	@ CPSR ← cc (r1 + r2)	compare negated
TST	r1,r2	@ CPSR ← cc (r1 et r2)	test
TEQ	r1,r2	@ CPSR ← cc (r1 ⊕ r2)	test equal

Mise à jour des codes condition

Les opérations de comparaison positionnent toujours les codes condition (c'est leur seul rôle). Par contre, les autres instructions de traitement de données ne les modifient que si le programmeur le demande en ajoutant un S (*set condition codes*) au code opération. Par exemple, l'addition de deux nombres 64 bits contenus dans $r0-r1$ et $r2-r3$ peut être réalisée par :

```
ADDS    r2, r2, r0    @ retenue dans C
ADC     r3, r3, r1    @ ajouter la retenue au mot de poids fort
```

Une opération arithmétique suivie de S, de même que l'instruction CMP ou CMN, positionne tous les codes condition. Une opération logique ou de mouvement entre registres ne positionne que N et Z (V n'est pas modifié, C non plus, sauf s'il s'agit d'un décalage, auquel cas C reçoit le dernier bit éjecté par le décalage).

Instructions d'accès à la mémoire

Les transferts peuvent se faire dans les deux sens :

load : mémoire → registre

store : registre → mémoire

Adressage indirect par registre

Il s'agit de la forme la plus simple : l'adresse de la donnée en mémoire est contenue dans un registre (appelé *registre de base*). L'instruction de transfert s'écrit alors :

```
LDR     r0, [r1]      @ r0 ← mem32[r1]
STR     r0, [r1]      @ mem32[r1] ← r0
```

NB : l'initialisation d'un registre avec une adresse mémoire peut se faire avec la pseudo-instruction ADR (voir page 7)

Adressage "base + déplacement"

A l'adresse contenue dans le registre de base, il est possible d'ajouter un déplacement pour calculer l'adresse du transfert. Par exemple :

```
LDR     r0, [r1, #4]   @ r0 ← mem32[r1+4]
```

Il s'agit ici d'un mode d'adressage **pré-indexé**. Le contenu du registre de base n'est pas modifié. Ce mode permet d'avoir une même adresse de base pour plusieurs transferts (avec des déplacements différents). C'est utile pour accéder à des données de type tableau : le registre de base contient l'adresse de début du tableau, et les déplacements successifs sont 0, 1, 2, 3, ... pour un tableau d'octets, ou 0, 4, 8, 12, ... pour un tableau de mots.

Parfois, il est pratique de modifier le registre de base pour y mettre l'adresse du transfert : c'est le mode **auto-indexé**. Par exemple :

```
LDR     r0, [r1, #4]!  @ r0 ← mem32[r1+4] ; r1 ← r1 + 4
```

Un autre mode intéressant est le mode **post-indexé** : il permet d'utiliser le registre de base comme adresse de transfert, puis de l'indexer ensuite. Par exemple :

```
LDR     r0, [r1], #4   @ r0 ← mem32[r1] ; r1 ← r1 + 4
```

Dans les exemples ci-dessus, le déplacement est spécifié sous la forme d'une constante. Il peut aussi être contenu dans un registre, éventuellement décalé. Par exemple :

```
LDR     r0, [r1, r2, LSL #2] @ r0 ← mem32[r1+r2*4]
```

Transfert d'octets

Les instructions LDR et STR transfèrent des mots (32 bits). Il est possible aussi de transférer des octets avec les instructions LDRB et STRB et des demi-mots avec LDRH et STRH.

Attention : lorsque l'on lit un octet, il est placé dans l'octet de poids faible du registre destination, et les octets de poids fort restants sont mis à 0. Si l'octet lu représente un nombre signé, c'est son bit de poids fort, c'est à dire le bit 7, qui indique le signe. Or, dans le registre, c'est le bit 31 (mis à 0 lors du transfert) qui représente le signe. Ainsi, le résultat de la lecture est toujours considéré comme un nombre positif. Pour conserver le signe, il faut utiliser l'instruction LDRSB qui étend le bit de signe : le bit 7 de l'octet lu est recopié sur les bits 8 à 31 du registre destination.

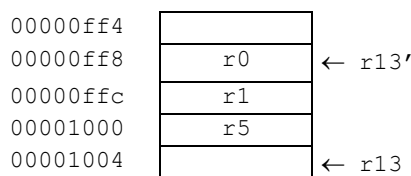
C'est la même chose pour les demi-mots.

Adressage de pile

Une pile est une zone de mémoire allouée dynamiquement et gérée en mode "dernier entré, premier sorti". L'ARM dispose des instructions nécessaires pour gérer différents types de piles. Nous utiliserons le modèle *full descending* (FD) dans lequel la pile grandit vers les adresses décroissantes et le pointeur de pile contient l'adresse du dernier élément rangé dans la pile. On peut utiliser des instructions de transfert multiples (LDM et STM) avec le suffixe FD pour transférer des valeurs entre la pile et des registres. Par exemple :

STMFD r13!, {r0,r1,r5} @ pour empiler

L'effet de cette instruction est illustré sur la figure ci-dessous. Par convention, le registre r13 contient le pointeur de pile. L'instruction transfère les registres r0, r1 et r5 dans la pile. Leur positionnement est automatique : les registres de numéros les plus petits sont placés aux adresses les plus petites (donc au sommet de la pile). La valeur de r13 après le transfert est ici représentée par r13'.



Pour un transfert inverse, on peut écrire :

LDMFD r13!, {r0,r1,r5} @ pour dépiler

Les valeurs lues aux adresses les plus petites sont rangées dans les registres de numéros les plus petits.

r0-r4 est équivalent à r0, r1, r2, r3, r4

NB : l'ordre des registres dans les accolades n'a aucune importance.

Exécution conditionnelle

A toute instruction on peut ajouter un suffixe qui indique sous quelle condition elle doit être exécutée. La condition est exprimée à partir des 4 codes conditions (voir page 1), selon le tableau ci-dessous. Si la condition n'est pas vérifiée, l'instruction n'est pas exécutée (et le processeur passe à l'instruction suivante).

<i>suffixe</i>	<i>signification</i>	<i>condition</i>	<i>suffixe</i>	<i>signification</i>	<i>condition</i>
AL	<i>always</i>	<i>1</i>	VC	<i>overflow clear</i>	\bar{V}
EQ	<i>equal</i>	<i>Z</i>	VS	<i>overflow set</i>	<i>V</i>
NE	<i>not equal</i>	\bar{Z}	HI	<i>higher</i>	$C.\bar{Z}$
PL	<i>plus</i>	\bar{N}	LS	<i>lower or same</i>	$\bar{C} + Z$
MI	<i>minus</i>	<i>N</i>	GT	<i>greater than</i>	$\bar{Z}.N.V + \bar{N}.\bar{V}$
CC / LO	<i>carry clear</i>	\bar{C}	GE	<i>greater or equal</i>	$N.V + \bar{N}.\bar{V}$
CS / HS	<i>carry set</i>	<i>C</i>	LT	<i>less than</i>	$N.\bar{V} + \bar{N}.V$
			LE	<i>less or equal</i>	$Z + N.\bar{V} + \bar{N}.V$

Une instruction conditionnelle doit être précédée d'une instruction qui positionne les codes conditions. S'ils sont positionnés par une instruction du type CMP x, y (selon le résultat de la soustraction x-y), voici la condition à choisir si on veut que l'instruction conditionnelle soit exécutée :

<i>condition</i>	<i>nombres non signés</i>	<i>nombres signés</i>
x = y	EQ	EQ
x ≠ y	NE	NE
x > y	HI	GT
x ≥ y	CS / HS	GE
x < y	CC / LO	LT
x ≤ y	LS	LE

Par exemple :

CMP r0, r1
ADDNE r3, r3, #1 @ le ADD n'est exécuté que si r0 ≠ r1

Instructions de contrôle de flot

Branchements

Il est possible de placer des étiquettes devant des instructions (voir page 6).

La façon la plus courante de rompre une exécution en séquence est d'utiliser une instruction de branchement :

```
                B          etiquette
                ...
etiquette:      ...
```

Quand le processeur atteint le branchement, il continue l'exécution à partir de l'instruction désignée par *etiquette* au lieu de continuer en séquence (c'est-à-dire au lieu d'exécuter l'instruction qui se trouve à la suite du branchement). Dans l'exemple ci-dessus, *etiquette* est après le branchement, donc il s'agit de sauter les instructions qui se trouvent entre les deux. Mais *etiquette* pourrait aussi bien se situer avant le branchement, et dans ce cas le processeur reprendrait l'exécution en arrière et ré-exécuterait éventuellement les mêmes instructions.

Branchements conditionnels

Parfois, on veut que le processeur puisse choisir au moment de l'exécution s'il doit effectuer un branchement ou pas. Par exemple, pour réaliser une boucle, un branchement au début du corps de boucle est nécessaire, mais il ne doit être réalisé que tant que la condition de boucle est vraie. Il suffit pour cela d'utiliser un suffixe comme expliqué page 5 .

Voici un exemple de boucle qui itère 10 fois :

```
                MOV        r0,#0                @ init compteur de boucle (r0)
boucle:         CMP        r0,#10              @ comparer compteur et borne (10)
                BHS        sortie              @ bcht si borne atteinte (r0 ≥ 10)
                ...                          @ corps de boucle
                ADD        r0,r0,#1            @ incr. compteur de boucle
                B          boucle              @ retour au test (inconditionnel)
sortie:         ...                          @ sortie de boucle
```

Branchement avec lien de retour

Dans un programme, on souhaite parfois faire un branchement vers une fonction et pouvoir ensuite reprendre l'exécution après le point d'appel. Ceci nécessite de mémoriser l'adresse de retour.

Cette possibilité est présente dans l'assembleur ARM, avec l'instruction BL qui sauvegarde automatiquement l'adresse de retour dans le registre *r14*. Par exemple :

```
principal:     BL          routine              @ r14 ← suite ; pc ← routine
suite:         ...
               ...
routine:       ...
               MOV        pc,r14              @ pc ← r14
```

Syntaxe d'un fichier source (GNU gas)

Structure d'une ligne de code

La structure générale d'une ligne de code est :

```
<étiquette>:      <code instruction>          <opérande, opérande,... >
```

<étiquette> est un nom qui permet de représenter l'adresse de l'instruction en mémoire.

Elle peut être omise, mais le code de l'instruction doit être précédé d'au moins un espace. Il est préférable d'utiliser des tabulations pour une meilleure lisibilité du programme.

<code instruction> est un mnémonique. S'il y a plusieurs opérandes, ils sont séparés par des virgules.

Commentaires

Ce qui suit le caractère @ (jusqu'à la fin de la ligne) est considéré comme un commentaire.

Directives pour l'assembleur

Les directives ne sont pas des instructions. Elles ont essentiellement pour rôle de donner des indications sur le fichier exécutable attendu. Par exemple, la directive `.fill` permet de réserver de la place en mémoire pour des données. Sa syntaxe est :

```
<étiquette>: .fill <nb>,<taille>[,<valeur>]
```

Cette directive demande d'initialiser `<nb>` emplacements de `<taille>` octets avec la valeur `<valeur>`. Si les arguments `<taille>` et/ou `<valeur>` sont omis, ils sont fixés par défaut à `<taille> = 1` et `<valeur> = 0`.

Par exemple :

```
tab: .fill 4,1 @ réserve de la place pour 4 octets
tab: .fill 8,4,0xFF @ réserve de la place pour 8 entiers
initialisés à 0x000000FF
```

Les directives `.int` et `.byte` permettent de réserver de la place et d'initialiser des entiers ou des octets :

```
<étiquette>: .int <val1>[,<val2>[,<val3>...]]
```

Par exemple :

```
liste: .byte 0, 1, 2 @ réserve de la place pour 3 octets initialisés à 0, 1 et 2
```

La directive `.asciz` permet de réserver de la place et d'initialiser les octets qui codent une chaîne de caractères (y compris le 0 final). Par exemple :

```
chaîne: .asciz "aha" @ réserve 4 octets initialisés à 0x61, 0x68, 0x61, 0
```

La directive `.equ` permet de définir une constante symbolique. Le symbole sera remplacé par sa valeur dans le code source avant assemblage. La syntaxe est :

```
.equ <nom>, <valeur>
```

Exemple :

```
.equ N,10
```

La directive `.align 4` permet d'aligner ce qui suit sur une adresse multiple de 4 (ce n'est pas automatique). C'est utile quand on réserve de la place pour un nombre d'octets qui n'est pas multiple de 4.

Par exemple :

```
chaîne: .asciz "hahaha" @ réserve 7 octets (6 caractères suivis d'un 0)
        .align 4
var: .int 12 @ sans l'alignement, l'adresse de var ne serait pas multiple de 4
        @ et un LDR à cette adresse provoquerait une erreur
```

Pseudo-instructions : ADR et LDR

Initialisation d'un registre avec une adresse (ADR)

Avant un transfert de donnée, il faut initialiser un registre avec l'adresse correspondante.

Généralement cette adresse est repérée par une étiquette (suivie d'une directive comme `.fill` ou `.byte` ou `.int`).

On a vu qu'il y avait des restrictions sur les opérandes immédiats : il n'est donc pas envisageable d'utiliser une instruction de mouvement (MOV) d'une valeur immédiate vers un registre, car il est probable que l'adresse ne respecte pas les restrictions.

On peut par contre utiliser la pseudo-instruction ADR :

```
ADR <registre destination>,<étiquette>
```

Il ne s'agit pas d'une instruction du jeu d'instructions ARM mais d'une directive qui est transformée en une vraie instruction (ADD) au moment de l'assemblage¹.

Initialisation d'un registre avec une constante (LDR)

La pseudo-instruction LDR (à ne pas confondre avec l'instruction LDR) sert à placer dans un registre une constante qui ne répond pas aux règles des constantes codables dans une instruction. Sa syntaxe est la suivante :

```
LDR rd,=<valeur_32_bits>
```

Cette *pseudo-instruction* est transformée par l'assembleur en une ou plusieurs instructions ARM, en fonction de la valeur (MOV ou LDR)¹.

¹ La manière dont ADR et LDR sont transformées sera expliquée en cours.

[illegible]

Champ “condition” (cond)

La plupart des instructions ARM peuvent être conditionnelles : elles ne sont exécutées que si la condition est vraie. Par exemple, `addne r0, r0, #1` n’incrémente `r0` que si le bit Z est à 0. Le format du champ `cond` est le suivant :

cond	condition	cond	condition	cond	condition	cond	condition
0000	EQ	0001	NE	0010	CS/HS	0011	CC/LO
0100	MI	0101	PL	0110	VS	0111	VC
1000	HI	1001	LS	1010	GE	1011	LT
1100	GT	1101	LE	1110	{AL}		

La valeur {AL} (ALways) est utilisée pour toutes les instructions inconditionnelles.

Instructions de traitement de données (arithmétiques ou logiques)

Champ “code opération” (opcode)

Il indique le type d’une instruction arithmétique ou logique selon le codage suivant :

opcode	opération	opcode	opération	opcode	opération	opcode	opération
0000	AND	0001	EOR	0010	SUB	0011	RSB
0100	ADD	0101	ADC	0110	SBC	0111	RSC
1000	TST	1001	TEQ	1010	CMP	1011	CMN
1100	ORR	1101	MOV	1110	BIC	1111	MVN

Champ “numéro de registre” (Rd, Rn, Rm, Rs)

Il indique un numéro de registre, sur 4 bits (pour représenter un registre parmi `r0 – r15`).

`Rd` désigne le registre destination, `Rn` le premier registre opérande, `Rm` le second registre opérande s’il y en a un, `Rs` le registre qui contient le nombre de positions de décalage (si le second opérande est décalé).

Spécification du second opérande (bits 11 à 0)

Il y a trois formats possibles :

- **constante** (bit 25 à 1) : elle est spécifiée par une valeur immédiate (sur 8 bits) et un nombre de rotations (sur 4 bits). La constante est calculée de la manière suivante :

$$\text{constante} = (\text{immediate}) \ggg (2 * \text{rotate})$$

où \ggg représente un décalage circulaire vers la droite.

Par exemple, la valeur `0x3F0` est encodée par :

`immediate = 0x3F` et `rotate = 0xE`

(décalage circulaire de $2 * 14$ positions vers la droite, équivalent à un décalage de 4 positions vers la gauche)

NB : plusieurs codages sont possibles, on choisit celui ayant la plus petite valeur `rotate`.

- **registre décalé d’un nombre de positions constant** (bit 25 à 0, bit 4 à 0) : le numéro de registre est donné dans le champ `Rm`, le type de décalage est indiqué par les bits 6 et 5 (00 = LSL, 01 = LSR, 10 = ASR, 11 = ROR), le nombre de positions de décalage est spécifié par les bits 11 à 7 (sur 5 bits : possibilité de décaler de 31 positions).
Cas particulier : **registre non décalé**. Les bits 11 à 4 sont à 0 : décalage LSL de 0 positions

- **registre décalé d'un nombre de positions dynamique, contenu dans un registre** (bit 25 à 0, bit 4 à 1) : le numéro de registre est dans le champ `Rm`, le type de décalage est indiqué par les bits 6 et 5 (voir ci-dessus), le champ `Rs` désigne le registre qui contient le nombre de positions de décalage.

Bit "S"

Lorsqu'il est à 1, ce bit indique que les indicateurs C, Z, N et V doivent être mis à jour (selon le type de l'instruction).

Pour les instructions CMP, CMN, TST, TEQ, ce bit est toujours à 1. Pour les autres instructions, il est à 1 quand l'instruction comporte le suffixe 's' (exemple : `adds`).

Instructions de branchement

Champ "déplacement" (24-bit offset)

Le déplacement est la valeur à ajouter à r15 pour atteindre la cible du branchement. Il faut se rappeler que le processeur ARM traite les instructions dans un pipeline à trois étages. Aussi, quand il exécute l'instruction n (par exemple un branchement), le registre r15 contient l'adresse de l'instruction $n+2$ (soit : [adresse de n] + 8, puisque les instructions sont codées sur 4 octets). La valeur du déplacement est donc égale à la distance entre le branchement et sa cible, moins 8.

Par ailleurs, puisque les instructions sont alignées sur des adresses multiples de 4, le déplacement est toujours un multiple de 4. Ses deux bits de poids faible sont donc à 00. Pour économiser de la place (et pouvoir coder des déplacements plus grands), les deux bits de poids faible ne sont pas présents dans le champ offset (le champ contient donc la valeur du déplacement divisée par 4).

Ainsi, le calcul de l'adresse cible lors de l'exécution du branchement se fait de la manière suivante :

- initialiser le déplacement (sur 32 bits) à la valeur présente dans le champ `offset`. Ce champ étant sur 24 bits, et les déplacements pouvant être négatifs, il faut étendre le bit de signe.
- décaler le déplacement de deux positions vers la gauche (le multiplier par 4)
- ajouter le déplacement à r15

Bit "L"

Lorsqu'il est à 1, ce bit indique que l'adresse de l'instruction suivante $n+1$ (« adresse de retour ») doit être sauvegardée dans le registre r14 avant exécution du branchement. Puisque r15 pointe sur l'instruction en cours de chargement (instruction $n+2$), l'adresse de retour est égale à $(r15 - 4)$.

Instructions d'accès à la mémoire simples

Champ "numéro de registre" (`Rd`, `Rn`, `Rm`)

`Rd` représente le registre destination (pour un load) ou source (pour un store). `Rn` désigne le registre de base utilisé pour calculer l'adresse. `Rm`, s'il est présent, désigne le registre d'index.

Bits "L, B, P, U, W"

Le bit L indique s'il s'agit d'un load ($L = 1$) ou d'un store.

Le bit B indique s'il s'agit d'un accès à un octet ($B=1$) ou à un mot de 4 octets ($B=0$).

Les bits P, U et W spécifient le mode d'adressage :

- $P = 0$ signifie que l'adressage est post-indexé : l'adresse est la valeur du registre de base, et ce dernier est ensuite modifié par ajout de l'index.

P = 1 et W = 0 : il s'agit d'un adressage indexé simple sans modification du registre de base.

P = 1 et W = 1 : l'adressage est pré-indexé. L'adresse est calculée en ajoutant l'index au registre de base et ce dernier est ensuite mis à jour avec cette adresse.

- U = 1 indique que l'index est ajouté au registre de base ; U = 0 indique que l'index est soustrait du registre de base.

Spécification de l'index

Deux formats sont possibles :

- **index immédiat** (bit 25 à 0) : la valeur absolue de l'index est codée sur les bits 11 à 0 (le signe est indiqué par le bit U).
- **index registre avec mise à l'échelle** (registre décalé) (bit 25 à 1) : le bit 4 est à 0 ; les bits 11 à 7 indiquent le nombre de positions de décalage ; les bits 6 et 5 indiquent le type de décalage (00 = LSL, 01 = LSR, 10 = ASR, 11 = ROR).

Cas particulier : si les bits 11 à 4 sont à 0, le registre d'index n'est pas décalé.

Instructions d'accès à la mémoire multiples

Champ Rn

Il désigne le registre de base pour le calcul d'adresse.

Registres à sauvegarder/restaurer (**register list**)

Chacun des 16 bits de ce champ correspond à un registre (bit 0 pour r0 ... bit 15 pour r15). Le bit i est à 1 si le registre r_i doit être sauvegardé/restauré, et à 0 sinon.

Spécification du mode d'adressage (bits "L, P, U, S, W »)

On ne détaille pas les significations de ces bits. Pour un `ldmfd`, les bits (L,P,U) sont à (1,0,1). Pour un `stmfd`, ils sont à (0,1,0).

W = 1 indique si le registre de base doit être mis à jour après le transfert (incrémenté si U est à 1, décrémenté si U = 0, d'une valeur égale à quatre fois le nombre de registres transférés).

Pour un LDM (load multiple), le bit "S" a une signification que nous verrons lorsque nous étudierons les interruptions (pour le moment, on suppose que S = 0).