Assembleur ARM

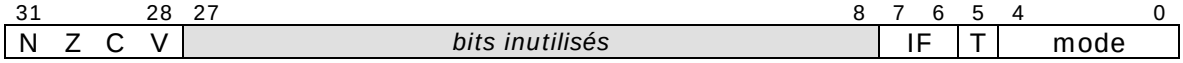
Le modèle de mémoire

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*).

Le registre CPSR contient des bits condition. Le programmeur n'a normalement pas besoin de connaître sa structure exacte mais, pour information, la voici :



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*) : N=1 indique que la dernière opération qui a mis à jour les codes conditions a produit un résultat négatif (bit de poids fort à 1).
- **Z** (*Zero*) : Z=1 indique que la dernière opération qui a mis à jour les codes conditions a produit un résultat nul
- **C** (*Carry*) : C=0 indique que la dernière opération qui a mis à jour les codes conditions a généré une retenue sortante sur une opération arithmétique ou sur un décalage
- **V** (*oVerflow*) : V=1 indique que la dernière opération qui a mis à jour les codes conditions a produit un débordement sur le bit de signe

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 (leur adresse est 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 *0*	**BB** *1*	**CC** *2*	**DD** *3*
33 *4*	**22** *5*	**11** *6*	**00** *7*
78 *8*	**56** *9*	**34** *10*	**12** *11*

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 5
mot **0x12345678** à l'adresse 8

Le jeu d'instructions ARM

L'architecture ARM 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 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)

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
```

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 mettre à jour 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
```

Opérandes registres décalés

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

```
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*2^r2
```

Opérandes immédiats

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 qui peuvent être données sont n'importe quelle valeur sur 32 bits dont tous les 1 sont dans 8 positions adjacentes alignées sur une frontière de deux bits, c'est-à-dire des valeurs de la forme : (0 à 255) x 2^{2n}, avec $0 \leq n < 16$ ([0-255] codé sur 8 bits, *n* codé sur 4 bits).

On peut contourner la difficulté grâce à la *pseudo-instruction* LDR (*à ne pas confondre avec l'*instruction *LDR*). 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 :

- si la valeur est de la forme (0 à 255) x 2^{2n} :

  `LDR rd,=<valeur>` est transformée en `MOV rd,#<valeur>`

- sinon :

  `LDR rd,=<valeur>` est transformée en

```
etiq :  LDR rd,[r15,#<depl>]
        …
        …
cste :  .int <valeur>
```

avec depl = cste – etiq - 8

Multiplications

Une forme spéciale d'instruction arithmétique permet de faire des multiplications :

```
MUL     r4,r3,r2        @ r4 ←(r3 * r2)[31:0]
```

Il y a des différences importantes avec les autres opérations arithmétiques :

- 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.

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.

Une forme alternative permet d'ajouter un produit à un registre d'accumulation :

```
MLA     r4,r3,r2,r1     @ r4 ← (r3 * r2 + r1)[31:0]
```

Une multiplication par une constante peut être implémentée en chargeant la constante dans un registre puis en utilisant une des instructions ci-dessus, mais il peut être plus efficace d'utiliser une séquence d'instructions avec des décalages et des additions. Par exemple, pour multiplier r0 par 35, on peut écrire :

```
ADD     r0,r0,r0,LSL #2     @ r0 ← 5 * r0
RSB     r0,r0,r0,LSL #3     @ r0 ← 7 * r0
```

Mise à jour des codes condition

Les opérations de comparaison mettent toujours à jour les codes condition (c'est leur seul rôle). Par contre, les autres instructions de traitement de données ne mettent à jour les codes condition 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, de même que l'instruction CMP ou CMN, met à jour tous les codes condition. Une opération logique ou de mouvement entre registres ne met à jour 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 de transfert de données

Il s'agit de transferts entre les registres et la mémoire :

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
```

Initialisation d'un pointeur

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.

Il est par contre possible de calculer l'adresse de la donnée en ajoutant une constante *d* au compteur de programme (PC). Cette constante doit tenir compte du fait que, lorsque le

calcul sera exécuté, le PC sera égal à l'adresse de l'instruction + 8 (traitement des instructions en pipeline).

Exemple :

code assembleur		*code objet*	
debut:	*r3 ← adresse de var*	00000010	*xxxxxxxx*
	…	00000014	xxxxxxxx
	…	00000018	xxxxxxxx
	…	0000001c	xxxxxxxx
	…	00000020	xxxxxxxx
var:	.fill 1,4	00000024	00000000

L'adresse var peut alors être calculée par :

```
add r3,pc,#12          @ var = PC + 12 = début + 8 + 12
```

On voit bien que ce calcul est fastidieux. De plus, tout ajout d'une instruction au milieu du code nécessite de recommencer le calcul. C'est pour cela que la pseudo-instruction ADR a été prévue. Sa syntaxe est la suivante :

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

Elle est automatiquement remplacée par l'assembleur par une instruction de calcul de l'adresse par rapport au PC, comme ci-dessus. Dans l'exemple précédent, on écrirait :

```
adr r3,var
```

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 était toujours une valeur immédiate. Il peut aussi être contenu dans un registre, éventuellement décalé.

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.

Transferts multiples

Quand on doit transférer plusieurs registres de ou vers la mémoire, on peut le faire en une seule fois avec les instructions LDM et STM. Par exemple :

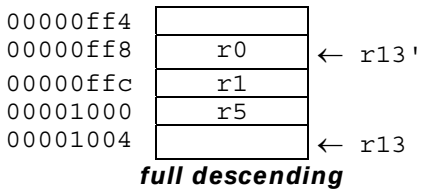
```
LDMIA   r1,{r0,r2,r5}  @ r0 ← mem32[r1]
                       @ r2 ← mem32[r1+4]
                       @ r5 ← mem32[r1+8]
```

La liste des registres de transfert (entre accolades) peut comprendre n'importe quels registres entre r0 et r15. L'ordre de la liste n'a pas d'importance, le transfert se fait de toute façon dans l'ordre croissant des numéros de registres.

L'extension IA (*increment after*) indique que l'adresse de base doit être incrémentée *après* le premier accès. D'autres extensions sont possibles : IB (*increment before*), DA (*decrement after*) et DB (*decrement before*).

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* dans lequel la pile grandit vers les adresses décroissantes et le pointeur de pile contient l'adresse du dernier élément rangé dans la pile. La figure ci-dessous montre comment on empile des données dans ce type de pile : on suppose que les registres transférés sont r0, r1 et r5 ; le pointeur de pile est r13, et on représente par r13' la valeur de ce registre après empilement.



full descending

La pile peut être manipulée avec les instructions de transfert multiples suivantes :

```
STMFD   r13!,{r0,r1,r5}        pour empiler
LDMFD   r13!,{r0,r1,r5}        pour dépiler
```

Le point d'exclamation après r13 indique que r13 doit être mis à jour après le transfert. Dans l'exemple ci-dessus, r13 vaudrait 0x1004 avant transfert, 0x0ff8 après empilement et 0x1004 après dépilement. De cette manière, on peut empiler d'autres choses au-dessus des trois registres avant de dépiler le tout.

Instructions de contrôle de flot

Branchements

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 `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, `étiquette` est après le branchement, donc il s'agit de sauter les instructions qui se trouvent entre les deux. Mais `étiquette` 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.

Une instruction de branchement conditionnel est associée à une condition (liée aux codes condition contenus dans le registre CPSR), et le branchement n'est réalisé que si la condition est vraie.

Voici un exemple de boucle :

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

Cet exemple montre un branchement conditionnel, BNE (*branch if not equal*). Il y a plusieurs conditions possibles, listées dans le tableau de la page suivante.

La signification des indicateurs mis à jour par une opération est la suivante :

- Z = 1 si le résultat de l'opération est nul
- N = bit 31 du résultat
- C = 1 si l'opération n'a *pas* produit de retenue
- V = 1 l'opération a généré un débordement en nombres signés

instruction	*signification*	*condition*	*instruction*	*signification*	*condition*
B	*unconditional*	toujours	BVC	*overflow clear*	$\overline{V}$
BEQ	*equal*	Z	BVS	*overflow set*	V
BNE	*not equal*	$\overline{Z}$	BHI	*higher*	$C.\overline{Z}$
BPL	*plus*	$\overline{N}$	BLS	*lower or same*	$\overline{C}+Z$
BMI	*minus*	N	BGT	*greater than*	$\left(N.V+\overline{N}.\overline{V}\right).\overline{Z}$
BCC	*carry clear*	$\overline{C}$	BGE	*greater or equal*	$N.V+\overline{N}.\overline{V}$
BCS	*carry set*	C	BLT	*less than*	$N.\overline{V}+\overline{N}.V$
			BLE	*less or equal*	$N.\overline{V}+\overline{N}.V+Z$

Par exemple, après une instruction de comparaison CMP x,y, qui positionne les codes conditions selon le résultat de x-y, voici le branchement conditionnel à choisir selon la condition souhaitée :

condition	*nombres non signés*	*nombres signés*
x = y	beq	beq
x ≠ y	bne	bne
x > y	bhi	bgt
x ≥ y	bcs (ou bhs)	bge
x < y	Bcc (ou blo)	blt
x ≤ y	bls	ble

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 ← pc + 4 ; pc ← routine
suite:      …
            …
routine:    …
            MOV     pc,r14           @ pc ← r14
```

Noter que comme l'adresse de retour est rangée dans un registre, un sous-programme ne peut pas en appeler un autre sans avoir sauvegardé son adresse de retour. Généralement, cette sauvegarde se fait dans la pile. Par exemple :

```
principal:  …
            BL      sp1
            …
sp1:        STMFA   r13!,{r14}       @ ou STR   r14,[r13,#4]!
            BL      sp2
            …
            LDMFA   r13!,{r15}       @ ou LDR   r15,[r13],#4
sp2:        …
            MOV     r15,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,…>
```

L'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.
Le code de l'instruction est un mnémonique. S'il y a plusieurs opérandes, ils sont séparés par des virgules.

Commentaires

Tout 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>]
```

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
```

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.
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 :

```
chaine: .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
```