Licence STS mention Informatique
EL5INFGM
C. Rochange et P. Sainrat

Assembleur ARM

Sommaire

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

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 :

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 *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 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 à 255) x 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*2^r2
```

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.

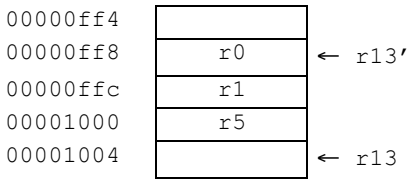
Adressage de pile

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).

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

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 :

condition	*nombres non signés*	*nombres signés*
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, 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. 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 :

```
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`

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 :

```
chaine: .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'assemblage[1].

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.