Chapitre 2

Langage machine et langage d'assemblage ARM

2.1 Résumé de documentation technique ARM

2.1.1 Organisation des registres

Dans le mode dit "utilisateur" le processeur `ARM` a 16 registres visibles de taille 32 bits nommés `r0, r1, ..., r15` :

- `r13` (synonyme `sp`, comme "stack pointer") est utilisé comme registre pointeur de pile.
- `r14` (synonyme `lr` comme "link register") est utilisé par l'instruction "branch and link" (`bl`) pour sauvegarder l'adresse de retour lors d'un appel de procédure.
- `r15` (synonyme `pc`, comme "program counter") est le registre compteur de programme.

Les conventions de programmation des procédures (ATPCS="ARM-Thumb Procedure Call Standard, Cf. Developer Guide, chapitre 2) précisent :

- les registres `r0, r1, r2 et r3` sont utilisés pour le passage des paramètres (données ou résultats)
- le registre `r12` (synonyme `ip`) est un "intra-procedure call scratch register" ; autrement dit il peut être modifié par une procédure appelée.
- le compilateur `arm-elf-gcc` utilise le registre `r11` (synonyme `fp` comme "frame pointer") comme base de l'environnement de définition d'une procédure.

Le processeur a de plus un registre d'état, `cpsr` pour "Current Program Status Register", qui comporte entre autres les codes de conditions arithmétiques. Le registre d'état est décrit dans la figure 2.1.

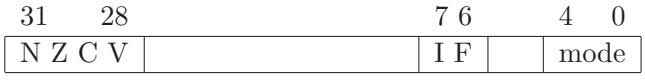

FIGURE 2.1 – Registre d'état du processeur ARM

Les bits `N, Z, C` et `V` sont les codes de conditions arithmétiques, `I` et `F` permettent le masquage des interruptions et `mode` définit le mode d'exécution du processeur (`User`, `Abort`, `Supervisor`, `IRQ`, etc).

2.1.2 Les instructions

Nous utilisons trois types d'instructions : les instructions arithmétiques et logiques (paragraphe 2.1.5), les instructions de rupture de séquence (paragraphe 2.1.6) et les instructions de transfert

d'information entre les registres et la mémoire (paragraphe 2.1.7).

Les instructions sont codées sur 32 bits.

Certaines instructions peuvent modifier les codes de conditions arithmétiques `N, Z, C, V` en ajoutant un `S` au nom de l'instruction.

Toutes les instructions peuvent utiliser les codes de conditions arithmétiques en ajoutant un mnémonique (Cf. figure 2.2) au nom de l'instruction. Au niveau de l'exécution, l'instruction est exécutée si la condition est vraie.

2.1.3 Les codes de conditions arithmétiques

La figure 2.2 décrit l'ensemble des conditions arithmétiques.

code	mnémonique	signification	condition testée
0000	EQ	égal	Z
0001	NE	non égal	$\overline{Z}$
0010	CS/HS	$\geq$ dans $\mathbb{N}$	C
0011	CC/LO	$<$ dans $\mathbb{N}$	$\overline{C}$
0100	MI	moins	N
0101	PL	plus	$\overline{N}$
0110	VS	débordement	V
0111	VC	pas de débordement	$\overline{V}$
1000	HI	$>$ dans $\mathbb{N}$	$C \wedge \overline{Z}$
1001	LS	$\leq$ dans $\mathbb{N}$	$\overline{C} \vee Z$
1010	GE	$\geq$ dans $\mathbb{Z}$	$(N \wedge V) \vee (\overline{N} \wedge \overline{V})$
1011	LT	$<$ dans $\mathbb{Z}$	$(N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
1100	GT	$>$ dans $\mathbb{Z}$	$\overline{Z} \wedge ((N \wedge V) \vee (\overline{N} \wedge \overline{V}))$
1101	LE	$\leq$ dans $\mathbb{Z}$	$Z \vee (N \wedge \overline{V}) \vee (\overline{N} \wedge V)$
1110	AL	toujours	

FIGURE 2.2 – Codes des conditions arithmétiques

Toute instruction peut être exécutée sous une des conditions décrites dans la figure 2.2. Le code de la condition figure dans les bits 28 à 31 du code de l'instruction. Par défaut, la condition est `AL`.

2.1.4 Description de l'instruction de chargement d'un registre

Nous choisissons dans ce paragraphe de décrire en détail le codage d'une instruction.

L'instruction `MOV` permet de charger un registre avec une valeur immédiate ou de transférer la valeur d'un registre dans un autre avec modification par translation ou rotation de cette valeur.

La syntaxe de l'instruction de transfert est : `MOV [<COND>] [S] <rd>, <opérande>` où `rd` désigne le registre destination et `opérande` est décrit par la table ci-dessous :

`opérande`	commentaire
`#immédiate-8`	entier sur 32 bits (Cf. remarque ci-dessous)
`rm`	registre
`rm, shift #shift-imm-5`	registre dont la valeur est décalée d'un nombre de positions représenté sur 5 bits
`rm, shift rs`	registre dont la valeur est décalée du nombre de positions contenu dans le registre `rs`

Dans la table précédente le champ `shift` de l'opérande peut être `LSL, LSR, ASR, ROR` qui signifient respectivement "logical shift left", "logical shift right", "arithmetic shift right", "rotate right".

Une valeur immédiate est notée selon les mêmes conventions que dans le langage C ; ainsi elle peut être décrite en décimal (15), en hexadécimal (0xF, le chiffre zéro suivi du caratère x, suivi du caractère F) ou en octal (017, le chiffre zéro suivi des chiffres 1 et 7).

Le codage de l'instruction `MOV` est décrit dans les figures 2.3 et 2.4. `<COND>` désigne un mnémonique de condition ; s'il est omis la condition est `AL`. Le bit `S` est mis à 1 si l'on souhaite une mise à jour des codes de conditions arithmétiques. Le bit `I` vaut 1 dans le cas de chargement d'une valeur immédiate. Les codes des opérations `LSL, LSR, ASR, ROR` sont respectivement : $00, 01, 10, 11$.

Remarque concernant les valeurs immédiates : Une valeur immédiate sur 32 bits (opérande `#immediate`) sera codée dans l'instruction au moyen, d'une part d'une constante exprimée sur 8 bits (bits 7 à 0 de l'instruction, figure 2.4, 1^{er} cas), et d'autre part d'une rotation exprimée sur 4 bits (bits 11 à 8) qui sera appliquée à la dite constante lors de l'exécution de l'instruction.

La valeur de rotation, comprise entre 0 et 15, est multipliée par 2 lors de l'exécution et permet donc d'appliquer à la constante une rotation à **droite** d'un nombre pair de positions compris entre 0 et 30. La rotation s'applique aux 8 bits placés initialement à droite dans un mot de 32 bits (qui n'est pas celui qui contient l'instruction).

Il en résulte que ne peuvent être codées dans l'instruction toutes les valeurs immédiates sur 32 bits...

Une rotation nulle permettra de coder toutes les valeurs immédiates sur 8 bits.

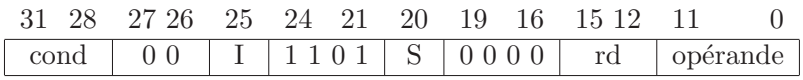

FIGURE 2.3 – Codage de l'instruction `mov`

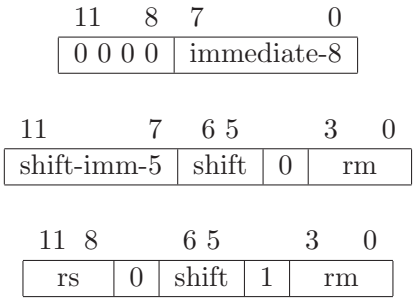

FIGURE 2.4 – Codage de la partie `opérande` d'une instruction

Exemples d'utilisations de l'instruction `mov`

```
MOV r1, #42         @ r1 <-- 42
MOV r3, r5          @ r3 <-- r5
MOV r2, r7, LSL #28 @ r2 <-- r7 décalé à gauche de 28 positions
MOV r1, r0, LSR r2  @ r1 <-- r0 décalé à droite de n pos., r2=n
MOVS r2, #-5        @ r2 <-- -5 + positionnement N, Z, C et V
MOVEQ r1, #42       @ si cond(EQ) alors r1 <-- 42
MOVLTS r3, r5       @ si cond(LT) alors r3 <-- r5 + positionnement N, Z, C et V
```

2.1.5 Description des instructions arithmétiques et logiques

Les instructions arithmétiques et logiques ont pour syntaxe :
`code-op[<cond>][s] <rd>, <rn>, <opérande>`, où `code-op` est le nom de l'opération, `rn` et `opérande` sont les deux opérandes et `rd` le registre destination.

Le codage d'une telle instruction est donné dans la figure 2.5. `opérande` est décrit dans le paragraphe 2.1.4, figure 2.4.

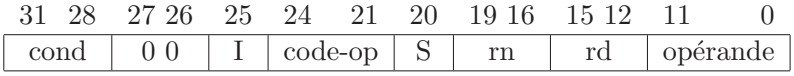

FIGURE 2.5 – Codage d'une instruction arithmétique ou logique

La table ci-dessous donne la liste des intructions arithmétiques et logiques ainsi que les instructions de chargement d'un registre. Les instructions `TST, TEQ, CMP, CMN` n'ont pas de registre destination, elles ont ainsi seulement deux opérandes ; elles provoquent systématiquement la mise à jour des codes de conditions arithmétiques (dans le codage de l'instruction les bits 12 à 15 sont mis à zéro). Les instructions `MOV` et `MVN` ont un registre destination et un opérande (dans le codage de l'instruction les bits 16 à 19 sont mis à zéro).

code-op	Nom	Explication du nom	Opération	remarque
0000	`AND`	AND	et bit à bit	
0001	`EOR`	Exclusive OR	ou exclusif bit à bit	
0010	`SUB`	SUBstract	soustraction	
0011	`RSB`	Reverse SuBstract	soustraction inversée	
0100	`ADD`	ADDition	addition	
0101	`ADC`	ADdition with Carry	addition avec retenue	
0110	`SBC`	SuBstract with Carry	soustraction avec emprunt	
0111	`RSC`	Reverse Substract with Carry	soustraction inversée avec emprunt	
1000	`TST`	TeST	et bit à bit	pas `rd`
1001	`TEQ`	Test EQuivalence	ou exclusif bit à bit	pas `rd`
1010	`CMP`	CoMPare	soustraction	pas `rd`
1011	`CMN`	CoMpare Not	addition	pas `rd`
1100	`ORR`	OR	ou bit à bit	
1101	`MOV`	MOVe	copie	pas `rn`
1110	`BIC`	BIt Clear	et not bit à bit	
1111	`MVN`	MoVe Not	not (complément à 1)	pas `rn`

Exemples d'utilisations

```
ADD r1, r2, r5           @ r1 <-- r2 + r5
ADDS r0, r2, #4          @ r0 <-- r2 + 4 + positionnement NZCV
SUB r3, r7, r0           @ r3 <-- r7 - r0
SUBS r3, r7, r0          @ r3 <-- r7 - r0 + positionnement NZCV
SUBGES r3, r7, r0        @ si cond(GE) r3 <-- r7 - r0 et positionnement NZCV
CMP r1, r2               @ calcul de r1-r2 et positionnement NZCV
TST r3, #1               @ calcul de r3 ET 1 et positionnement NZCV
ANDS r1, r2, #0x0000ff00 @ r1 <-- r2 ET 0x0000ff00 et positionnement NZCV
```

2.1.6 Description des instructions de rupture de séquence

Nous utilisons trois instructions de rupture de séquence : `B[<cond>] <déplacement>`, `BL[<cond>] <déplacement>`, `BLX[<cond>] <registre>`.

a) Instruction `B[<cond>] <déplacement>` L'instruction `B` provoque la modification du compteur de programme si la condition est vraie ; le texte suivant est extrait de la documentation ARM :

```
if ConditionPassed(cond) then
   PC <-- PC + (SignExtend(déplacement) << 2)
```

L'expression `(SignExtend(déplacement) << 2)` signifie que le `déplacement` est tout d'abord étendu de façon signée à 32 bits puis mutiplié par 4. Le `déplacement` est en fait un entier relatif (codé sur 24 bits comme indiqué ci-dessous) et qui représente le nombre d'instructions (en avant ou en arrière) entre l'instruction de rupture de séquence et la cible de cette instruction.

Dans le calcul du déplacement, il faut prendre en compte le fait que lors de l'exécution d'une instruction, le compteur de programme ne repère pas l'instruction courante mais deux instructions en avant.

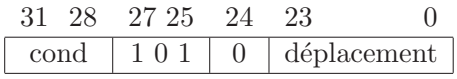

FIGURE 2.6 – Codage de l'instruction de rupture de séquence `b{cond}`

Exemples d'utilisations

```
BEQ +5      @ si cond(EQ) alors pc <-- pc + 4*5
BAL -8      @ pc <-- pc - 4*8
```

Dans la pratique, on utilise une étiquette (Cf. paragraphe 2.2.4) pour désigner l'instruction cible d'un branchement. C'est le traducteur (i.e. l'assembleur) qui effectue le calcul du déplacement.

La figure 2.7 résume l'utilisation des instructions de branchements conditionnels après une comparaison.

Conditions des instructions de branchement conditionnel				
Type	Entiers relatifs ($\mathbb{Z}$)		Naturels ($\mathbb{N}$) et adresses	
Instruction C	Bxx	Condition	Bxx	Condition
goto	BAL	1110	BAL	1110
if (x== y) goto	BEQ	0000	BEQ	0000
if (x != y) goto	BNE	0001	BNE	0001
if (x < y) goto	BLT	1011	BLO, BCC	0011
if (x<= y) goto	BLE	1101	BLS	1001
if (x > y) goto	BGT	1100	BHI	1000
if (x>= y) goto	BGE	1010	BHS,BCS	0010

FIGURE 2.7 – Utilisation des branchements conditionnels après une comparaison

b) Instruction `BL[<cond>] <déplacement>` L'instruction `BL` provoque la modification du compteur de programme avec sauvegarde de l'adresse de l'instruction suivante (appelée `adresse de retour`) dans le registre `lr` ; le texte suivant est extrait de la documentation ARM :

```
lr <-- address of the instruction after the branch instruction
PC <-- PC + (SignExtend(déplacement) << 2)
```

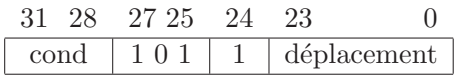

FIGURE 2.8 – Codage de l'instruction de branchement à un sous-programme `bl`

Exemples d'utilisations

```
BL 42       @ lr <-- pc+4 ; pc <-- pc +4*42
```

c) Instruction `BLX[<cond>] <registre>` L'instruction `BLX Rm` provoque la modification du compteur de programme avec sauvegarde de l'adresse de l'instruction suivante (appelée `adresse de retour`) dans le registre `lr` ; le texte suivant est extrait de la documentation ARM :

```
lr <-- address of the instruction after the branch instruction
PC <-- Rm
```

Attention : Dans le cadre des TP, l'adresse passée en paramètre à `BLX` doit être paire. En effet, l'exécution de `BLX` avec une adresse impaire active un mode spécial du processeur (THUMB) avec un autre jeu d'instructions (codées sur 16 bits). Ce type d'erreur peut avoir des effets assez variés en fonction du programme concerné : on peut obtenir un message d'erreur relatif au mode THUMB ou un comportement arbitraire du simulateur.

Exemples d'utilisations

```
BLX R5       @ lr <-- pc+4 ; pc <-- R5
```

Pour désigner une procédure on utilisera une étiquette ; des exemples sont donnés dans le paragraphe 2.2.4.

2.1.7 Description des instructions de transfert d'information entre les registres et la mémoire

Transfert entre un registre et la mémoire

L'instruction `LDR` dont la syntaxe est : `LDR <rd>, <mode-adressage>` permet le transfert du mot mémoire dont l'adresse est spécifiée par `mode-adressage` vers le registre `rd`. Nous ne donnons pas le codage de l'instruction `LDR` parce qu'il comporte un grand nombre de cas ; nous regardons ici uniquement les utilisations les plus fréquentes de cette instruction.

Le champ `mode-adressage` comporte, entre crochets, un registre et éventuellement une valeur immédiate ou un autre registre, ceux-ci pouvant être précédés du signe + ou −. Le tableau ci-dessous indique pour chaque cas le mot mémoire qui est chargé dans le registre destination. L'instruction `ldr` permet beaucoup d'autres types de calcul d'adresse qui ne sont pas décrits ici.

mode-adressage	opération effectuée
`[rn]`	rd <– mem [rn]
`[rn, #offset12]`	rd <– mem [rn + offset12]
`[rn, #-offset12]`	rd <– mem [rn - offset12]
`[rn, rm]`	rd <– mem [rn + rm]
`[rn, -rm]`	rd <– mem [rn - rm]

Il existe des variantes de l'instruction `LDR` permettant d'accéder à un octet : `LDRB` ou à un mot de 16 bits : `LDRH`. Et si l'on veut accéder à un octet signé : `LDRSB` ou à un mot de 16 bits signé : `LDRSH`. Ces variantes imposent cependant des limitations d'adressage par rapport aux versions 32 bits (exemple : valeur immédiate codée sur 5 bits au lieu de 12).

Pour réaliser le transfert inverse, registre vers mémoire, on trouve l'instruction `STR` et ses variantes `STRB` et `STRH`. La syntaxe est la même que celle de l'instruction `LDR`. Par exemple, l'instruction `STR rd, [rn]` provoque l'exécution : `MEM [rn] <-- rd`.

Exemples d'utilisations

```
LDR r1, [r0]           @ r1 <-32bits-- Mem [r0]
LDR r3, [r2, #4]       @ r3 <-32bits-- Mem [r2 + 4]
LDR r3, [r2, #-8]      @ r3 <-32bits-- Mem [r2 - 8]
LDR r3, [pc, #48]      @ r3 <-32bits-- Mem [pc + 48]
LDRB r5, [r3]          @ 8bits_poids_faibles (r5) <-- Mem [r3],
                       @ extension aux 32 bits avec des 0
STRH r2, [r1, r3]      @ Mem [r1 + r3] <-16bits-- 16bits_poids_faibles (r2)
```

L'instruction `LDR` est utilisée entre autres pour accéder à un mot de la zone text en réalisant un adressage relatif au compteur de programme. Ainsi, l'instruction `LDR r2, [pc, #depl]` permet de charger dans le registre `r2` avec le mot mémoire situé à une distance `depl` du compteur de programme, c'est-à-dire de l'instruction en cours d'exécution. Ce mode d'adressage nous permet de recupérer l'adresse d'un mot de données (Cf. paragraphe 2.2.4).

Pré décrémentation et post incrémentation

Les instructions `LDR` et `STR` offrent des adressages post-incrémentés et pré-décrémentés qui permettent d'accéder à un mot de la mémoire et de mettre à jour une adresse, en une seule instruction. Cela revient à combiner un accès mémoire et l'incrémentation du pointeur sur celle-ci en une seule instruction.

instruction ARM	équivalent ARM	équivalent C
`LDR r1, [r2, #-4]!`	`SUB r2, r2, #4`	`r1 = *--r2`
	`LDR r1, [r2]`	
`LDR r1, [r2], #4`	`LDR r1, [r2]`	`r1 = *r2++`
	`ADD r2, r2, #4`	
`STR r1, [r2, #-4]!`	`SUB r2, r2, #4`	
	`STR r1, [r2]`	
`STR r1, [r2], #4`	`STR r1, [r2]`	
	`ADD r2, r2, #4`	

La valeur à incrémenter ou décrémenter (4 dans les exemples ci-dessus) peut aussi être donnée dans un registre.

Transfert multiples

Le processeur ARM possède des instructions de transfert entre un ensemble de registres et un bloc de mémoire repéré par un registre appelé registre de base : `LDM` et `STM`. Par exemple, `STMFD r7!, {r0,r1,r5}` range le contenu des registres `r0, r1` et `r5` dans la mémoire à partir de l'adresse contenue dans r7 et met à jour le registre `r7` après le transfert ; après l'exécution de l'instruction `MEM[r7]` contient `r0` et `MEM[r7+8]` contient `r5`.

Il existe 8 variantes de chacune des instructions `LDM` et `STM` selon que :

- les adresses de la zone mémoire dans laquelle sont copiés les registres croissent (Increment) ou décroissent (Decrement).
- l'adresse contenue dans le registre de base est incrémentée ou décrémentée avant (Before) ou après (After) le transfert de chaque registre. Notons que l'adresse est décrémentée avant le transfert quand le registre de base repère le mot qui a l'adresse immédiatement supérieure à celle où l'on veut ranger une valeur (Full) ; l'adresse est incrémentée après le transfert quand le registre de base repère le mot où l'on veut ranger une valeur (Empty).
- le registre de base est modifié à la fin de l'exécution quand il est suivi d'un `!` ou laissé inchangé sinon.

Ces instructions servent aussi à gérer une pile. Il existe différentes façons d'implémenter une pile selon que :

- le pointeur de pile repère le dernier mot empilé (`Full`) ou la première place vide (`Empty`).
- le pointeur de pile progresse vers les adresses basses quand on empile une information (`Descending`) ou vers les adresses hautes (`Ascending`).

Par exemple, dans le cas où le pointeur de pile repère l'information en sommet de pile (case pleine) et que la pile évolue vers les adresses basses (lorsque l'on empile l'addresse décroit), on parle de pile `Full Descending` et on utilise l'instruction `STMFD` pour empiler et `LDMFD` pour dépiler.

Les modes de gestion de la pile peuvent être caractérisés par la façon de modifier le pointeur de pile lors de l'empilement d'une valeur ou de la récupération de la valeur au sommet de la pile. Par exemple, dans le cas où le pointeur de pile repère l'information en sommet de pile et que la pile évolue vers les adresses basses, pour empiler une valeur il faut décrémenter le pointeur de pile avant le stockage en mémoire ; on utilisera l'instruction `STMDB` (`Decrement Before`). Dans le même type d'organisation pour dépiler on accède à l'information au sommet de pile puis on incrémente le pointeur de pile : on utilise alors l'instruction `LDMIA` (`Increment After`).

Selon que l'on prend le point de vue gestion d'un bloc de mémoire repéré par un registre ou gestion d'une pile repérée par le registre pointeur de pile, on considère une instruction ou une autre ... Ainsi, les instructions `STMFD` et `STMDB` sont équivalentes ; de même pour les instructions `LDMFD` et `LDMIA`.

Les tables suivantes donnent les noms des différentes variantes des instructions `LDM` et `STM`, chaque variante ayant deux noms synonymes l'un de l'autre.

nom de l'instruction	synonyme
LDMDA (decrement after)	LDMFA (full ascending)
LDMIA (increment after)	LDMFD (full descending)
LDMDB (decrement before)	LDMEA (empty ascending)
LDMIB (increment before)	LDMED (empty descending)

nom de l'instruction	synonyme
STMDA (decrement after)	STMED (empty descending)
STMIA (increment after)	STMEA (empty ascending)
STMDB (decrement before)	STMFD (full descending)
STMIB (increment before)	STMFA (full ascending)

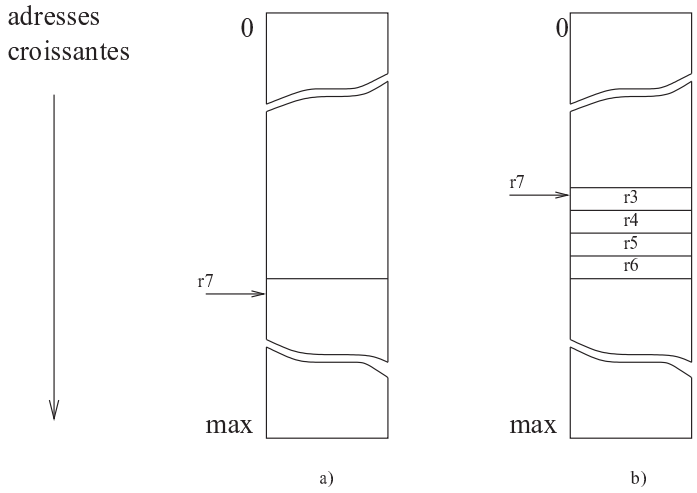

FIGURE 2.9 – Transfert multiples mémoire/registres : `STMFD r7!, {r3,r4,r5,r6}` ou (`STMDB ...`) permet de passer de l'état `a)` de la mémoire à l'état `b)`. `LDMFD r7!, {r3,r4,r5,r6}` (ou `LDMIA ...`) réalise l'inverse.

La figure 2.9 donne un exemple d'utilisation.

2.2 Langage d'assemblage

2.2.1 Structure d'un programme en langage d'assemblage

Un programme est composé de trois types de sections :
- données intialisées ou non (`.data`)
- données non initialisées (`.bss`)
- instructions (`.text`)

Les sections de données sont optionnelles, celle des instructions est obligatoire. On peut écrire des commentaires entre le symbole `@` et la fin de la ligne courante. Ainsi un programme standard a la structure :

```
   .data
@ déclaration de données
@ ...

   .text
@ des instructions
@ ...
```

2.2.2 Déclaration de données

Le langage permet de déclarer des valeurs entières en décimal (éventuellement précédées de leur signe) ou en hexadécimal ; on précise la taille souhaitée.

Exemple :

```
    .data
  .word 4536   @ déclaration de la valeur 4536 sur 32 bits (1 mot)
```

```
  .hword -24   @ déclaration de la valeur -24 sur 16 bits (1 demi mot)
  .byte 5      @ déclaration de la valeur 5 sur 8 bits (1 octet)
  .word 0xfff2a35f @ déclaration d'une valeur en hexadécimal sur 32 bits
  .byte 0xa5   @ idem sur 8 bits
```

On peut aussi déclarer des chaînes de caractères suivies ou non du caractère de code ASCII 00. Un caractère est codé par son code ASCII (Cf. paragraphe 1).

Exemple :

```
    .data
  .ascii "un texte" @ déclaration de 8 caractères...
  .asciz "un texte" @ déclaration de 9 caractères, les mêmes que ci-dessus
                    @ plus le code 0 à la fin
```

La définition de données doit respecter les règles suivantes, qui proviennent de l'organisation physique de la mémoire :

- un mot de 32 bits doit être rangé à une adresse multiple de 4
- un mot de 16 bits doit être rangé à une adresse multiple de 2
- il n'y a pas de contrainte pour ranger un octet (mot de 8 bits)

Pour recadrer une adresse en zone `data` le langage d'assemblage met à notre disposition la directive `.balign`.

Exemple :

```
   .data
@ on note AD l'adresse de chargement de la zone data
@ que l'on suppose multiple de 4 (c'est le cas avec les outils utilisés)
  .hword 43    @ après cette déclaration la prochaine adresse est AD+2
  .balign 4    @ recadrage sur une adresse multiple de 4
  .word 0xffff1234  @ rangé à l'adresse AD+4
  .byte 3      @ après cette déclaration la prochaine adresse est AD+9
  .balign 2    @ recadrage sur une adresse multiple de 2
  .hword 42    @ rangé à l'adresse AD+10
```

On peut aussi réserver de la place en zone `.data` ou en zone `.bss` avec la directive `.skip`.
`.skip 256` réserve 256 octets qui ne sont pas initialisés lors de la réservation. On pourra par programme écrire dans cette zone de mémoire.

2.2.3 La zone text

Le programmeur y écrit des instructions qui seront codées par l'assembleur (le traducteur) selon les conventions décrites dans le paragraphe 2.1.

La liaison avec le système (chargement et lancement du programme) est réalisée par la définition d'une étiquette (Cf. paragraphe suivant) réservée : `main`.

Ainsi la zone `text` est :

```
  .text
  .global main
main:

@ des instructions ARM
@ ...
```

2.2.4 Utilisation d'étiquettes

Une donnée déclarée en zone `data` ou `bss` ou une instruction de la zone `text` peut être précédée d'une étiquette. Une étiquette représente une adresse et permet de désigner la donnée ou l'instruction concernée.

Les étiquettes représentent une facilité d'écriture des programmes en langage d'assemblage.

Expression d'une rupture de séquence

On utilise une étiquette pour désigner l'instruction cible d'un branchement. C'est le traducteur (i.e. l'assembleur) qui effectue le calcul du déplacement. Par exemple :

```
etiq: MOV r0, #22
      ADDS r1, r2, r0
      BEQ etiq
```

Accès à une donnée depuis la zone `text`

```
   .data

DD: .word 5

   .text

@ acces au mot d'adresse DD
   LDR r1, relais  @ r1 <-- l'adresse DD
   LDR r2, [r1]    @ r2 <-- Mem[DD] c'est-à-dire 5

   MOV r3, #245    @ r3 <-- 245
   STR r3, [r1]    @ Mem[DD] <-- r3
                   @ la mémoire d'adresse DD a été modifiée

@ plus loin
relais: .word DD  @ déclaration de l'adresse DD en zone text
```

L'instruction `LDR r1, relais` est codée avec un adressage relatif au compteur de programme : `LDR r1, [pc, #depl]` (Cf. paragraphe 2.1.7).

Appel d'une procédure

On utilise l'instruction `BL` lorsque la procédure appelée est désignée directement par une étiquette :

```
   ...
ma_proc: @ corps de la procedure
         mov pc, lr
    ...
    @ appel de la procedure ma_proc
    BL ma_proc
```

On utilise l'instruction `BLX` lorsque l'adresse de la procédure est rangée dans un registre, par exemple lorsqu'une procédure est passée en paramètre :

```
    LDR   r1, adr_proc    @ r1 <-- adresse ma_proc
    BLX   r1
    ...
adr_proc: .word ma_proc
```

2.3 Organisation de la mémoire : petits bouts, gros bouts

La mémoire du processeur ARM peut être vue comme un tableau d'octets repérés par des numéros appelés **adresse** qui sont des entiers naturels sur 32 bits. On peut ranger dans la mémoire des mots de 32 bits, de 16 bits ou des octets (mots de 8 bits). Le paragraphe 2.2.2 indique comment déclarer de tels mots.

Dans la mémoire les mots de 32 bits sont rangés à des adresses multiples de 4. Il y a deux conventions de rangement de mots en mémoire selon l'ordre des octets de ce mot.

Considérons par exemple le mot `0x12345678`.

- convention dite "Big endian" (Gros bouts) :
  les 4 octets `12, 34, 56, 78` du mot `0x12345678` sont rangés aux adresses respectives 4x, 4x+1, 4x+2, 4x+3.
- convention dite "Little endian" (Petits Bouts) :
  les 4 octets `12, 34, 56, 78` du mot `0x12345678` sont rangés aux adresses respectives 4x+3, 4x+2, 4x+1, 4x.

Le processeur ARM suit la convention "Little endian". La conséquence est que lorsqu'on lit le mot de 32 bits rangé à l'adresse `4x` on voit : `78563412`, c'est-à-dire qu'il faut lire "à l'envers". Selon les outils utilisés le mot de 32 bits est présenté sous cette forme ou sous sa forme externe, plus agréable...

En général les outils de traduction et de simulation permettent de travailler avec une des deux conventions moyennant l'utilisation d'options particulières lors de l'appel des outils (option `-mbig-endian`).

2.4 Commandes de traduction, exécution, observation

2.4.1 Traduction d'un programme

Pour traduire un programme écrit en C contenu dans un fichier `prog.c` :
`arm-elf-gcc -g -o prog prog.c`. L'option `-o` permet de préciser le nom du programme exécutable ; `o` signifie "output". L'option `-g` permet d'avoir les informations nécessaires à la mise au point sous débogueur (Cf. paragraphe 2.4.2).

Pour traduire un programme écrit en langage d'assemblage ARM contenu dans un fichier `prog.s` :
`arm-elf-gcc -Wa,--gdwarf2 -o prog prog.s`.

Lorsque l'on veut traduire un programme qui est contenu dans plusieurs fichiers que l'on devra rassembler (on dit "lier"), il faut d'abord produire des versions partielles qui ont pour suffixe `.o`, le `o` voulant dire ici "objet". Par exemple, on a deux fichiers : principal.s et biblio.s, le premier contenant l'étiquette `main`. On effectuera la suite de commandes :

```
arm-elf-gcc -c -Wa,--gdwarf2 biblio.s
arm-elf-gcc -c -Wa,--gdwarf2 principal.s
arm-elf-gcc -g -o prog principal.o biblio.o
```

La première produit le fichier `biblio.o`, la seconde produit le fichier `principal.o`, la troisième les relie et produit le fichier exécutable `prog`.

Noter que les deux commandes suivantes ont le même effet :
`arm-elf-gcc -c prog.s` et

`arm-elf-as -o prog.o prog.s`. Elles produisent toutes deux un fichier objet `prog.o` sans les informations nécessaires à l'exécution sous débogueur.

2.4.2 Exécution d'un programme

Exécution directe

On peut exécuter un programme directement avec :
`arm-elf-run prog`. S'il n'y a pas d'entrées-sorties, on ne voit évidemment rien...

Exécution avec un débogueur

Nous pouvons utiliser deux versions du même débogueur : `gdb` et `ddd`. On parle aussi de metteur au point. C'est un outil qui permet d'exécuter un programme instruction par instruction en regardant les "tripes" du processeur au cours de l'exécution : valeur contenues dans les registres, dans le mot d'état, contenu de la mémoire, etc.

`gdb` est la version de base (textuelle), `ddd` est la même mais graphique (avec des fenêtres, des icônes, etc.), elle est plus conviviale mais plus sujette à des problèmes techniques liés à l'installation du logiciel...

Soit le programme objet exécutable : `prog`. Pour lancer `gdb` :
`arm-elf-gdb prog`. Puis taper successivement les commandes : `target sim` et enfin `load`. Maintenant on peut commencer la simulation.

Pour éviter de taper à chaque fois les deux commandes précédentes, vous pouvez créer un fichier de nom `.gdbinit` dont le contenu est :

```
# un diese débute un commentaire
# commandes de demarrage pour arm-elf-gdb
target sim
load
```

Au lancement de `arm-elf-gdb prog`, le contenu de ce fichier sera automatiquement exécuté.

Voilà un ensemble de commandes utiles :

- placer un point d'arrêt sur une instruction précédée d'une étiquette, par exemple : `break main`. On peut aussi demander `break no` avec `no` un numéro de ligne dans le code source. Un raccourci pour la commande est `b`.
- enlever un point d'arrêt : `delete break numéro_du_point_d'arrêt`
- voir le code source : `list`
- lancer l'exécution : `run`
- poursuivre l'exécution après un arrêt : `cont`, raccourci : `c`
- exécuter l'instruction à la ligne suivante, en entrant dans les procédures : `step`, raccourci `s`
- exécuter l'instruction suivante (sans entrer dans les procédures) : `next`, raccourci `n`
- voir la valeur contenue dans les registres : `info reg`
- voir la valeur contenue dans le registre `r1` : `info reg $r1`
- voir le contenu de la mémoire à l'adresse `etiquette` : `x &etiquette`
- voir le contenu de la mémoire à l'adresse `0x3ff5008` : `x 0x3ff5008`
- voir le contenu de la mémoire en précisant le nombre de mots et leur taille.
  `x /nw adr` permet d'afficher `n` mots de 32 bits à partir de l'adresse `adr`.
  `x /ph adr` permet d'afficher `p` mots de 16 bits à partir de l'adresse `adr`.
- modifier le contenu du registre `r3` avec la valeur `0x44` exprimée en hexadécimal : `set $r3=0x44`
- modifier le contenu de la mémoire d'adresse `etiquette` : `set *etiquette = 0x44`
- sortir : `quit`
- La touche `Enter` répète la dernière commande.

Et ne pas oublier : `man gdb` sous Unix (ou Linux) et quand on est sous `gdb` : `help nom_de_commande...`

Pour lancer `ddd` : `ddd --debugger arm-elf-gdb`. On obtient une grande fenêtre avec une partie dite "source" (en haut) et une partie dite "console" (en bas). Dans la fenêtre "console" taper successivement les commandes : `file prog`, `target sim` et enfin `load`.

On voit apparaître le source du programme en langage d'assemblage dans la fenêtre "source" et une petite fenêtre de "commandes". Maintenant on peut commencer la simulation.

Toutes les commandes de `gdb` sont utilisables soit en les tapant dans la fenêtre "console", soit en les sélectionnant dans le menu adéquat. On donne ci-dessous la description de quelques menus. Pour le reste, il suffit d'essayer.

- placer un point d'arrêt : sélectionner la ligne en question avec la souris et cliquer sur l'icône break (dans le panneau supérieur).
- démarrer l'exécution : cliquer sur le bouton Run de la fenêtre "commandes". Vous voyez apparaître une flèche verte qui vous indique la position du compteur de programme i.e. où en est le processeur de l'exécution de votre programme.
- le bouton Step permet l'exécution d'une ligne de code, le bouton Next aussi mais en entrant dans les procédures et le bouton Cont permet de poursuivre l'exécution.
- enlever un point d'arrêt : se positionner sur la ligne désirée et cliquer à nouveau sur l'icône break.
- voir le contenu des registres : sélectionner dans le menu Status : Registers ; une fenêtre apparaît. La valeur contenue dans chaque registre est donnée en hexadécimal (0x...) et en décimal.
- observer le contenu de la memoire étiquettée `etiquette` : apres avoir sélectionné memory dans le menu Data, on peut soit donner l'adresse en hexadecimal 0x... si on la connaît, soit donner directement le nom `etiquette` dans la case `from` en le précédant du caractère &, c'est-à-dire `&etiquette`.

2.4.3 Observation du code produit

Considérons un programme objet : `prog.o` obtenu par traduction d'un programme écrit en langage C ou en langage d'assemblage. L'objet de ce paragraphe est de décrire l'utilisation d'un ensemble d'outils permettant d'observer le contenu du fichier `prog.o`. Ce fichier contient les informations du programme source codées et organisées selon un format appelé `format ELF`.

On utilise trois outils : `hexdump`, `arm-elf-readelf`, `arm-elf-objdump`.

`hexdump` donne le contenu du fichier dans une forme brute.
`hexdump prog.o` donne ce contenu en hexadécimal complété par le caractère correspondant quand une valeur correspond à un code ascii ; de plus l'outil indique les adresses des informations contenues dans le fichier en hexadécimal aussi.

`arm-elf-objdump` permet d'avoir le contenu des zones `data` et `text` avec les commandes respectives :
`arm-elf-objdump -j .data -s prog.o` et
`arm-elf-objdump -j .text -s prog.o`. Ce contenu est donné en hexadécimal. On peut obtenir la zone `text` avec le désassemblage de chaque instruction :
`arm-elf-objdump -j .text -d prog.o`.

`arm-elf-readelf` permet d'avoir le contenu du reste du fichier.
`arm-elf-readelf -a prog.o` donne l'ensemble des sections contenues dans le fichier sauf les zones `data` et `text`.
`arm-elf-readelf -s prog.o` donne le contenu de la table des symboles.