

Les registres en mode utilisateur

Tous les registres du processeurs ont 32 bits de large : chaque registre stocke un **mot** binaire de 4 octets.

En assembleur les registres ne sont pas typés : entier signé ou non, pointeur, c'est au programmeur d'utiliser correctement les valeurs contenues.

Représentation compacte : hexadécimal	Représentation détaillée : binaire
contenu (4 octets) : 0x400DE5A0	contenu (32 bits) : 0b01000000000011011110010110100000
n° d'octet : 3 2 1 0	n° de bit : 31 ... 0

En mode utilisateur l'ARM utilise 16 registres numérotés r0 à r15 et un registre d'état CPSR

Registres d'usage général (calculs, résultats intermédiaires, compteurs, pointeurs ...)

r0* r1* r2* r3* r4 r5 r6 r7 r8 r9 r10 r11 r12*

* *Scratch registers* : peuvent être modifiés (sans sauvegarde/restauration) par un sous programme
r0, r1, r2 et r3 : utilisés pour passer les n 1ers arguments (n<=4) lors d'un appel de sous programme
r0 : utilisé pour passer la valeur de retour à la fin d'un appel de sous programme

Registres particuliers

r13 alias sp : **stack pointer** pointeur de pile (pointe sur la dernière valeur empilée)
r14 alias lr : **link register** adresse de retour à la procédure appelante (depuis un sous programme appelé)
r15 alias pc : **program counter** compteur ordinal, adresse de l'instruction en cours d'exécution + 8 (l'ARM regarde 2 instructions en avant)

CPSR : **Current Program Status Register** Registre d'état, contient des informations sur le mode de fonctionnement actuel et contient les indicateurs (**flags**) de code condition (bits 28 à 31 du CPSR)

Indicateurs de code condition : 4 bits du CPSR désignés par NZCV

Indicateur N (*Negative*) : N=1 indique une opération dont le résultat est négatif (bit de signe à 1)
Indicateur Z (*Zero*) : Z=1 indique une opération dont le résultat est nul ou une égalité lors d'une comparaison
Indicateur C (*Carry*) : C=1 indique une retenue sortante sur une opération (dépassement en interprétation non signée, exemple 0xffffffff + 1)
Indicateur V (*Overflow*) : V=1 indique un débordement sur une opération (en interprétation signée)

Spécificité de l'assembleur ARM : sauf pour les instruction *compare*, il faut explicitement préciser **S (Set)** en suffixe pour qu'une instruction modifie les indicateurs de code condition en fonction de son résultat. Par défaut les codes conditions ne sont pas modifiés par une instruction de calcul.

```

SUB    r1,r1,#1    @ Décrémente r1 mais ne modifie pas les indicateurs
SUBS   r0,r0,#1    @ Décrémente r0 et met à jour les indicateurs selon la valeur résultante
                    @ (en particulier si r0 vaut 0 alors Z=1 sinon Z=0)
    
```

Jeu d'instructions

Version d'architecture de l'ARM7TDMI : ARMv4T. Les instructions disponibles sont celles sans numéro de version ou de version 2, 3, 4, 4T ou M dans la colonne § de la documentation (ARM Instruction Set Quick Reference Card).

Exemples d'instructions courantes

```

MOV    r0,r2      @ r0 <- r2 la valeur dans r2 est copiée dans r0
MOV    r0,#8      @ r0 <- 8 la valeur immédiate 8 est mise dans r0 (maintenant r0=0x00000008)
MOV    r0,#256    @ r0 <- 256 la valeur immédiate 256 est mise dans r0 (maintenant r0=0x00000100)
ADD    r0,r1,r2   @ r0 <- r1 + r2 addition
SUB    r0,r1,r2   @ r0 <- r1 - r2 soustraction
MOV    r0,r2,LSL #4 @ r0 <- r2 * 16 le deuxième opérande est décalé à gauche de 4 (voir op. binaires)
ADD    r0,r1,r2,LSL #1 @ r0 <- r1 + (r2 * 2) le deuxième opérande est décalé à gauche de 1
    
```

Exemples d'instructions qui ne marchent pas

```

ADD    r0,r1,r2,r3,r4 @ Les instructions assembleur ont un format fixe
ADD    r0,#4,#20      @ Le 1er opérande est forcément un registre (voir doc)
MOV    r0,#725        @ La valeur constante 725 ne rentre pas dans l'instruction...
                    @ les seules valeurs possibles sont de type <immed_8r>
                    @ voir pseudo instruction ldr pour une solution à ce problème
    
```

Données immédiates de type <immed_8r>

Ce sont des constantes numériques préfixées par #. Exemples : #22, #-4, #0xE0, #0b01101111

<immed_8r> désigne une valeur construite en partant d'un non signé sur 8 bits (0 à 255) suivi d'une rotation à droite d'un nombre pair de bits.

Concrètement, les valeurs de 0 à 256 sont possibles, de même que les valeurs de 0 à 1024 multiples de 4, de 0 à 4096 multiples de 16 ...

Les valeurs immédiates négatives sont acceptées selon possibilités d'assemblage en instructions complémentaires (MOV en MVN, ADD en SUB) :

```

MOV    r0,#-15      @ -15 = 0xFFFFF11 (représentation signée sur 32 bits) NOT(0xFFFFF11) = 0x0000000E
-> MVN r0,#0xE      @ L'instruction qui sera effectivement utilisée dans le code.
    
```

Format des instructions et données immédiates

Chaque instruction d'un exécutable ARM est codée sur 32 bits, y compris ses arguments. A coté des autres informations (type d'opération, registres), il reste 12 bits pour chaque constante immédiate : 8 bits pour une valeur de base (0 à 255), 4 bits pour indiquer la rotation droite (0, 2, 4 ... 28, 30)

Etiquettes et directives d'assemblage

Le fichier source assembleur comporte des indications en plus des séquences d'instructions du programme. Ce sont des commandes ou des indications données à l'assembleur : directives d'assemblage (préfixées par un point).

```
.INCLUDE "fichier.s"           @ Insère le contenu de fichier.s à cet emplacement
.EQU  MACONSTANTE, 0x0200      @ MACONSTANTE devient synonyme de 0x0200
.BYTE 0,2,4,6                  @ Réserve de la place et initialise 4 octets
.HWORD 0x2E25                  @ Réserve de la place et initialise 1 demi-mot
.WORD 0x25                      @ Réserve de la place et initialise 1 mot ( 0x00000025 )
.ASCIZ "Une chaine"           @ Réserve de la place et initialise une chaine terminée par '\0'
.ALIGN                               @ Les emplacements mémoire suivants repartent à des adresses multiple de 4
.FILL 8,4,0xFF                 @ Réserve 8 entiers initialisés à 0x000000ff
```

Les étiquettes (*labels*) sont des références à l'adresse mémoire de leur emplacement dans le code ou les données. Attention chaque étiquette doit être un identifiant **unique**, suivi immédiatement par un deux-points.

```
MonEtiquette: .WORD @ MonEtiquette devient synonyme de l'adresse du mot (adresse déterminée à l'assemblage)
DebutProg:    MOV    R3,#200 @ DebutProg devient synonyme de l'adresse de cette instruction
```

Rapports entre mémoire et registres

L'architecture ARM est de type **load-store** : la plupart des instructions de traitement n'accèdent qu'aux données dans les registres, pas aux données en mémoire. Les instructions *Load* permettent de transférer les données depuis la mémoire vers les registres, et les instructions *Store* mettent le résultat d'un traitement en mémoire depuis un registre : Load(mémoire->registres) Traitements(registres->registres) Store(registres->mémoire)

Formats mots / demi-mots / octets et alignements en mémoire :

Accès mot : **LDR** ou **STR** adresse multiple de 4. Accès demi-mot : **LDRH** ou **STRH** adresse paire. Accès octet : **LDRB** ou **STRB** sans contrainte d'adresse.

```
MonCompteur: .WORD 3           @ Le code assembleur suivant réalise l'équivalent du code C MonCompteur++;
...          ...              @ Réserve l'espace mémoire statique pour un mot initialisé à 0x00000003
...          ...              @ Pas plus de 4ko entre les variables en mémoire et l'utilisation directe de
LDR r0,MonCompteur @ la pseudo instruction LDR qui charge la valeur à l'adr. mémoire de MonCompteur
ADD r0,r0,#1       @ Incrémenter de 1 la valeur du compteur dans le registre r0
STR r0,MonCompteur @ Remettre la nouvelle valeur en mémoire

MaVarX:      .WORD 0x24        @ Le code assembleur suivant réalise l'équivalent du code C MaVarX+=MaVarY;
MaVarY:      .WORD 0xE0        @ Réserve l'espace mémoire statique pour un mot initialisé à 0x000000E0
...          ...              @ Réserve l'espace mémoire statique pour un mot initialisé à 0x000000E0
...          ...              @ Pas de limite d'espace mémoire entre les variables et le code suivant
LDR r2,=MaVarX   @ Pseudo instruction LDR qui charge l'adresse mémoire absolue de MaVarX
LDR r0,[r2]      @ Instruction LDR qui utilise r2 comme pointeur pour lire à l'adresse de MaVarX
LDR r1,[r2,#4]  @ Instruction LDR avec déplacement : on accède à MaVarY (adresse MaVarX+4)
ADD r0,r0,r1     @ r0 <- r0 + r1
STR r0,[r2]     @ Remettre la nouvelle valeur en mémoire à l'adresse de MaVarX
```

L'instruction LDR et les pseudo instructions LDR (resp. LDRH pour les demi-mots et LDRB pour les octets)

On parle de pseudo-instruction quand le code du source assembleur est transformé en code machine différent ou plus complexe lors de l'assemblage.

Instruction ldr : ne peut accéder à des contenus mémoire qu'à partir d'un registre utilisé comme pointeur (éventuellement avec déplacement).

```
LDR r0,[r1] @ Equivalent C : r0 = *r1; (r1 étant un pointeur, on accède au contenu pointé)
```

Pseudo instruction ldr pour éviter d'avoir à trop s'occuper des limitations sur les données immédiates (de type #55 ou #0xAE ...)

```
LDR r0,#725 @ Equivalent C : r0 = 725; // On charge une constante 32 bits quelconque
```

Utilisation pour charger une constante qui correspond à une adresse mémoire (même syntaxe mais on utilise une étiquette)

```
LDR r0,=MaVar @ Equivalent C : r0 = &MaVar; // On charge une adresse de variable
```

Utilisation directe de la pseudo instruction pour charger la valeur contenue à l'adresse indiquée par l'étiquette

```
LDR r0,MaVar @ Equivalent C : r0 = MaVar; // On charge dans r0 la valeur de la variable MaVar
@ Pratique mais la variable ne doit pas être trop loin (4ko) de ce code
```

L'instruction STR et la pseudo instruction STR (resp. STRH pour les demi-mots et STRB pour les octets)

Instruction str : ne peut écrire des contenus mémoire qu'à partir d'un registre utilisé comme pointeur (éventuellement avec déplacement).

```
STR r0,[r1] @ Equivalent C : *r1 = r0; (r1 étant un pointeur, on écrit le contenu pointé)
```

Utilisation directe de la pseudo instruction pour écrire la valeur à l'adresse indiquée par l'étiquette

```
STR r0,MaVar @ Equivalent C : MaVar = r0; // On écrit la valeur de r0 dans la variable MaVar
@ Pratique mais la variable ne doit pas être trop loin (4ko) de ce code
```

la pile

Le programmeur en assembleur dispose d'une pile principale de données en mémoire qui permet de sauver des informations en les empilant (*push*) puis de les retrouver plus tard en les dépilant (*pop*), sans avoir à allouer des espaces mémoires statiques. Le registre r13 (alias sp) pointe sur la dernière donnée empilée au sommet de la pile, l'empilement se fait des adresses hautes vers les adresses basses.

```
@ Syntaxe longue
STMFD sp!,{r2-r5,r8} @ Store Multiple Full Descending with stack pointer      PUSH {r2-r5,r8}
...                  @ Traitements qui modifient r2,r3,r4,r5,r8                ...
LDMFD sp!,{r2-r5,r8} @ Load Multiple Full Descending with stack pointer      POP {r2-r5,r8}
@ On récupère les valeurs initiales des registres

@ Syntaxe courte utilisant les alias PUSH et POP (code équivalent)
PUSH {r2-r5,r8} @ On empile
...
POP {r2-r5,r8} @ On dépile
```

Le modèle mémoire

Par défaut l'ARM utilise un modèle mémoire de type **little endian**. Les mots et demi-mots sont stockés en mémoire respectivement sur 4 et 2 octets en écrivant d'abord les octets de poids faibles puis ceux de poids forts. Tant qu'on écrit (store) et lit (load) en mémoire avec les mêmes formats aux mêmes adresses ça n'a pas d'importance pour le programmeur. Par exemple écrire depuis un registre le mot 0x01234567 à l'adresse 100, puis lire de nouveau un mot à l'adresse 100 donne bien 0x01234567 dans le registre destination. Mais si on analyse octet par octet à l'adresse 100 on trouve :

```
Contenu mémoire      : 0x67 0x45 0x23 0x01 ...      Dans le registre   : 0x01234567
Adresses croissantes : 100 101 102 103 ...          <- poids forts / poids faibles ->
```

Exécution conditionnelle

On peut conditionner l'exécution d'une instruction à l'état des indicateurs code condition en la postfixant par un des codes conditionnels suivants :

code exécute si description

```
EQ    Z=1      Equal
NE    Z=0      Not Equal
CS    C=1      higher or same (Carry Set)
CC    C=0      lower (Carry Clear)
MI    N=1      negative (Minus)
PL    N=0      positive or zero (Plus)
VS    V=1      oVerflow Set
VC    V=0      oVerflow Clear
HI    C=1 and Z=0 Higher
LS    C=0 and Z=1 Lower or Same
GE    N=V      Greater or Equal
LT    N!=V     Less Than
GT    Z=0 and N=V Greater Than
LE    Z=1 or N!=V Less than or Equal
```

Comparaison de deux valeurs : CMP r0,r1

code conditionnel à utiliser en fonction du rapport des valeurs

condition	non signé	signé
r0==r1	EQ	EQ
r0!=r1	NE	NE
r0>r1	HI	GT
r0>=r1	CS	GE
r0<r1	CC	LT
r0<=r1	LS	LE

```
CMP    r1,r2    @ Comparaison r1 - r2 : r1 et r2 non modifiés, indicateurs code condition mis à jour
ADDEQ  r0,r0,#1 @ N'incrémente r0 que si la comparaison précédente a indiqué r1==r2 (Add if Equal)

SUBS   r1,r1,#1 @ Décrémente r1 et met à jour les indicateurs (postfixe Set)
STRNE  r1,[r2]  @ N'écrit la valeur de r1 à l'adresse pointée par r2 que si Z=0 (r1 non nul)
ADDEQ  r3,r3,#1 @ N'incrémente la valeur de r3 que si Z=1 (r1 nul)
...    @ Dans tous les cas l'exécution continue ...
```

Branchements et branchements conditionnels

Par défaut l'exécution des instructions par le processeur est séquentielle. Les branchements permettent de reprendre l'exécution à un autre endroit. En postfixant l'instruction de branchement B par une condition on obtient des branchements conditionnels, qui permettent de réaliser tests et boucles.

Boucle infinie

```
Boucler: B Boucler @ Branchement non conditionnel sur lui même : boucle infinie
```

Boucle avec décompte de 10 à 1 (10 passages)

```
MOV    r0,#10    @ Initialisation d'un compteur : mieux vaut utiliser un registre (plus efficace)
MaBoucle:
...    @ Etiquette de début de boucle
...    @ Instructions du corps de boucle : répété 10 fois
SUBS   r0,r0,#1  @ Décrémenter r0 et mettre à jour les indicateurs de code condition (Z=1 si r0=0)
BNE    MaBoucle  @ Branch if Not Equal : repartir en début de boucle si Z=0 (r0 non nul)
...    @ sinon on continue après la boucle
```

Boucle avec comptage de 0 à 9 équivalent de for (r0=0;r0<10;r0++)

```
MOV    r0,#0    @ Initialisation du compteur à 0
Boucle2:
...    @ Etiquette de début de boucle
CMP    r0,#10   @ Comparaison à la borne
BCS    Sortie2  @ Sauter en dehors de la boucle si la borne est atteinte ou dépassée (r0>=10)
...    @ Instructions du corps de boucle : répété 10 fois
ADD    r0,r0,#1 @ Incrémenter compteur de boucle r0
B      Boucle2  @ Repartir en début de boucle (pas de condition)
Sortie2:
...    @ Etiquette de sortie
...    @ on continue après la boucle
```

Structure conditionnelle de type SI ALORS (attention : penser à prendre le code conditionnel complémentaire)

```
CMP    r0,r1    @ Comparaison de r0 et r1
BHI    Suite1   @ Si r0>r1 on n'exécute PAS les instructions suivante (on saute par dessus)
...    @ Ces instructions sont donc exécutées SI r0<=r1
Suite1: ...    @ Dans tous les cas le code qui suit est exécuté
```

Structure conditionnelle de type SI ALORS SINON (comme pour le SI, penser à la condition complémentaire)

```
CMP    r0,r1    @ Comparaison de r0 et r1
BHI    Sinon2   @ Si r0>r1 on n'exécute PAS les instructions suivante (on saute au SINON)
...    @ SI r0<=r1 ces instructions sont donc exécutées
B      Suite2   @ On passe par dessus le sinon pour reprendre la suite
Sinon2: ...    @ SINON ces instructions sont exécutées
...
Suite2: ...    @ Dans tous les cas le code qui suit est exécuté
```