IUT 'A' Paul SABATIER
Dépt INFORMATIQUE

ASR => Architecture des ordinateurs : Ass2

Programmation en langage d'assemblage

des microprocesseurs ARM

J.P. CARRARA – R. FACCA – P. MAGNAUD

INTRODUCTION

Un langage d'assemblage est un langage de programmation.

Il permet de coder sous une forme **symbolique** :

- les instructions du langage machine d'un processeur ou d'une famille de processeurs.
- des directives d'allocation mémoire pour les données du programme.
- des directives de contrôle pour le **traducteur**.

Le traducteur d'un langage d'assemblage est appelé un **Assembleur**.

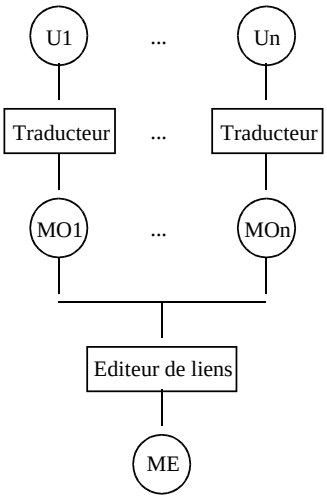
A partir du programme **source** codé en langage d'assemblage, l'Assembleur génère un programme **objet** constitué des instructions et des données du programme codées en langage machine binaire.

Sauf cas très particulier, un programme est composé d'un programme principal et de sous-programmes.

Certains sous-programmes peuvent être codés dans des fichiers différents (unités de compilation) de celui du programme principal.

Certains sous-programmes peuvent être déjà disponibles dans des bibliothèques sous forme de programme **objet**.

Les différents programmes **objets** (modules objets relogeables) doivent être fusionnés par un utilitaire appelé **éditeur de liens** (linking loader) qui génèrera le programme **exécutable** (module objet exécutable).



Processus de compilation séparée

Domaines d'utilisation des langages d'assemblages

Applications dépendantes de l'architecture de la machine :

- pilotes de périphériques (device driver)
- parties basses du noyau d'un OS (Ex : BIOS pour Windows)
- programmation de microcontrôleurs

Applications temps réel :

- moteurs de jeux vidéo
- contrôle de processus

Quel est l'intérêt d'étudier un langage d'assemblage ?

Dans le cadre du DUT, l'intérêt est uniquement pédagogique.

Pour les enseignements d'Architecture des Ordinateurs, cette étude permet d'illustrer :

- les concepts de représentations et de traitements de l'information
- la structure et le fonctionnement d'un vrai processeur
- la programmation dans un vrai langage machine

Pour les enseignements de Systèmes d'Exploitation, cette étude permet d'illustrer :

- les techniques de passage de paramètres des sous-programmes utilisées dans les compilateurs
- l'interfaçage entre des modules codés en langage machine et en langage évolué
- les techniques de compilation séparée
- l'utilisation explicite d'outils de développement tels que des éditeurs de liens ou des gestionnaires d'applications multi-fichiers (ex : make)
- les concepts d'utilitaires **natifs** ou **croisés** (ex : cross-assembleur)

Pourquoi avoir choisi les microprocesseurs ARM ?

Les microprocesseur de la famille ARM sont peu connus du grand public comparativement aux microprocesseurs des familles INTEL ou AMD. Pourtant ils équipent pratiquement les trois quarts des appareils électroniques portables ou des équipements informatiques tels que :

- les téléphones mobiles (Nokia, Motorola, Sony-Ericson, ...)
- les consoles de jeux (Nintendo, PlayStation, Game Boy, ...)
- les baladeurs MP3 (iPod d'Apple, ...)
- les assistants personnels (Palm, Pocket PC,
- les navigateurs GPS (TomTom, Navtman, ...)
- les périphériques informatiques (disques durs externes, scanners, imprimantes, ...)
- les équipements réseaux (routeurs, modem-routeurs, commutateurs, points d'accès Wifi, ..)
- les équipements d'accès Internet multimédia (FreeBox3, AliceBox, ...)
- etc ....

Le succès des ARM est dû aux performances élevées qu'ils offrent pour une faible consommation et un coût relativement bon marché. Un autre atout important, est de pouvoir faire tourner des systèmes d'exploitations libres tels que Linux ou µClinux sur des équipements à base d'ARM.

C' est sûrement pour ces raisons que l'Assembleur des microprocesseur de la famille ARM est pris comme exemple dans les enseignements d'Architecture des Ordinateurs de très nombreuses filières Informatique

CHAPITRE 1 : Architecture des microprocesseurs ARM

1. Les principaux microprocesseurs de la famille ARM

microprocesseurs 32 bits de la 3ème génération (1ère génération → µP 4bits ; 2ème génération → µP 8bits)

1983-85 ▪ l' **ARM1** est développé par la société britannique **ACORN**. ARM signifie Acorn RISC Machine.

1987 ▪ ACORN qui à été rachetée par OLIVETTI, sort son premier ordinateur personnel ARCHIMEDES fonctionnant sur l'**ARM2** à 8 Mhz puis 12 MHz. ACORN est alors l'un des leaders du marché de l'informatique personnelle et éducative en Grande-Bretagne.

1989 ▪ ACORN en partenariat avec le fondeur VLSI, sort l'**ARM3** qui fonctionne à 25 Mhz puis 33 MHz et qui dispose d'un cache unique de 4 Ko.
De nombreux fabricants commencent à intégrer des ARM dans leurs produits.
ACORN, VLSI et APPLE créent la société ARM (Advanced RISC Machine) chargée de poursuivre le développement de la technologie ARM.

1990 ▪ sortie de la famille **ARM6** cache unique de 4 Ko
coprocesseur arithmétique flottant (FPU) et contrôleur graphique amélioré.

199? ▪ sortie de la famille **ARM7** 3 niveaux de pipeline, cache unique de 8 Ko
contrôleur de mémoire virtuelle (MMU)
l' ARM7TDMI dispose d'un second jeu d'instructions appelé THUMB permettant le codage des instructions sur 16 bits (au lieu de 32) et de réaliser ainsi un gain de place mémoire important notamment pour les applications embarquées.

199? ▪ sortie de la famille **ARM9** qui fonctionne de 130 Mhz à 200 MHz , 5 niveaux de pipeline
architecture Hardward (cache instructions et cache données séparés de 8 Ko), MMU

1999 ▪ sortie de la famille **ARM10** qui fonctionne de 200 Mhz à 400 MHz , 6 niveaux de pipeline
architecture Hardward (cache instructions et cache données séparés de 16 Ko), MMU
coprocesseur virgule flottante VFP10 simple et double précision à la norme IEEE 754

2001-03▪ sortie de la famille **ARM11** qui fonctionne de 200 Mhz à 400 MHz , 6 niveaux de pipeline
architecture Hardward (cache instructions et cache données séparés), MMU
coprocesseur virgule flottante VFP10 simple et double précision à la norme IEEE 754

2. Caractéristiques de l'ARM

Il existe 2 types d'architectures :

- celles du type **RISC** (Reduced Instruction Set)
- celles du type **CISC** (Complex Instruction Set)

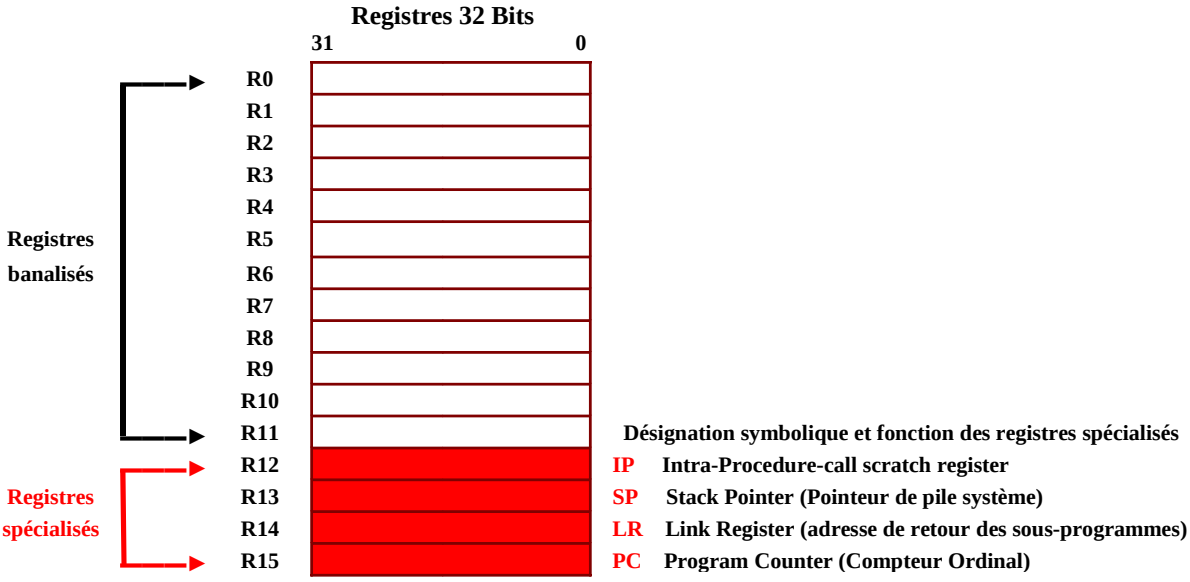
L'ARM possède les caractéristiques standard des architectures du type **RISC** (Reduced Instruction Set) :

- un banc de registres internes important (16 registres)
- un format d'instruction fixe : codage sur un mot (32 bits) pour être lue en 1 seul cycle mémoire.
- tous les traitements se font uniquement sur des opérandes stockés dans les registres internes et non en mémoire dont le temps d'accès est beaucoup plus long que celui des registres internes.
  L'accès à la mémoire est limité à 2 types d'instructions:
  - chargement de registres internes (**load**) avec des données lues en mémoire.
  - écriture en mémoire (**store**) de données contenues dans des registres internes.
- des modes d'adressage standards simples
- la quasi totalité des instructions s'exécutent en un seul cycle d'horloge.

Il possède en plus quelques caractéristiques particulières qui améliorent ses performances comme :

- l'exécution conditionnelle de toutes les instructions.
- des modes d'adressage spécifiques complexes.

3. Modèle simplifié de programmation



Register d'état appelé CPSR (Current Program Status Register)

- **N** (Negative): Indicateur de résultat **négatif** après une opération arithmétique sur des entiers relatifs en complément vrai. N est égal au bit de plus fort poids du résultat.
- **Z** (Zero): Indicateur de résultat **nul** après une opération arithmétique, logique, de décalage ou de comparaison. Z=1 si tous les bits du résultat sont égaux à 0.
- **C** (Carry): Indicateur de **débordement** après une opération arithmétique sur des entiers naturels ou dernier bit sortant dans une opération de décalage.
  La Carry aussi est modifiée par de nombreuses autres instructions.
- **V** (oVerflow): Indicateur de **débordement** après une opération arithmétique sur des **entiers relatifs en complément vrai.** V=1 si les 2 dernières retenues du résultat sont différentes.
- **mode**: ces 5 bits définissent le mode de fonctionnement du processeur.
- **T (Thumb)**: T=0 si le processeur utilise le jeu standard d'instructions. Le processeur peut aussi utiliser un jeu réduit d'instructions, dans ce cas T=1 (fonctionnement en mode Thumb).
- **IF (IRQ et FIRQ)**: ces 2 bits servent à gérer les interruptions matérielles IRQ et FIRQ.

► 3 formats possibles pour les données traitées dans les **registres banalisés R0 – R11** :

- 8 bits (**Byte**)
- 16 bits (**Halfword**)
- 32 bits (**Word**)

Attention !

Les données au format 8 ou 16 bits sont chargées dans **les bits de faibles poids des registres** et complétées à gauche par des **0** sur les **bits de forts poids**, excepté pour les **relatifs négatifs en CV** qui sont complétées à gauche par des **1**.

Ceci permet de ne pas modifier la valeur des entiers naturels ou relatifs lorsqu'ils passent d'une représentation 8 ou 16 bits à une représentation 32 bits (cf. cours Arc1).

► formats 32 bits pour les adresses chargées dans les **registres banalisés R0 – R11**, lorsqu'ils sont utilisés comme **pointeurs** pour lire ou écrire en mémoire

► formats 32 bits pour les adresses chargées dans les **registres spécialisés R12– R15**

4. Représentation interne des informations

4.1. valeurs numériques

- naturels binaires sur 8, 16 ou 32 bits
- relatifs binaires en CV sur 8, 16 ou 32 bits
- réels en virgule flottante IEEE 754 sur 32, 64 ou 96 bits s'il y a un coprocesseur
- caractères ASCII sur 8 bits traités comme des naturels dans l'intervalle [0-255]

4.2. adresses mémoire

adresses absolues ou relatives (déplacements) sur 8, 16 ou 32 bits représentant :

- des pointeurs sur des données en mémoire
- des étiquettes d'instructions
- des adresses de registres circuits contrôleurs de périphériques

4.3. entités binaires

mots binaires 8, 16 ou 32 bits représentant :

- des constantes binaires dans les opérations logiques et de décalages
- des mots d'état ou de contrôle du microprocesseur ou de circuits contrôleurs de périphériques

5. Langage interne

5.1. instructions

Nous utiliserons uniquement le jeu d'instructions standard.
Il est constitué classiquement des différentes catégories d'instructions suivantes :

- instructions de transferts
- instructions de traitements (arithmétiques, logiques et de décalages)
- instructions de tests et comparaisons
- instructions de branchements
- instructions de contrôle du microprocesseur

5.2. format des instructions

une instruction machine est codée sur 32 bits et est composée de 2 parties :

CODEOPER	ADRESSES

- un champ CODEOPER contenant le **codage en binaire** de l'opération que doit réaliser l'instruction et d'indicateurs binaires qui peuvent conditionner l'exécution de l'instruction.

- un champ ADRESSES contenant la **désignation des opérandes** de l'opération.
le champ ADRESSES est constitué de 1 à 4 champs adresses suivant le nombre d'opérandes de l'opération.

5.3. modes d'adressages

- Dans les instructions autres que les branchements, un **champ adresse** sert à désigner l'emplacement de l'opérande qui peut être stocké dans un **registre** ou en **mémoire**.

  S'il est en mémoire on peut y accéder **directement** en indiquant son adresse mémoire ou **indirectement** par un **pointeur**.

  Un champ adresse peut donc contenir un nom de registre, une adresse mémoire absolue ou relative ou une combinaison de nom de registre et d'adresse utilisés pour calculer la valeur du pointeur.

  Si l'opérande est une **constante**, sa valeur est codée directement dans le champ adresse (**adressage immédiat**).

- Dans les instructions de branchements, un **champ adresse** sert à désigner l'étiquette de l'instruction à laquelle devra se faire le branchement.

  Un champ adresse peut donc contenir une adresse mémoire d'instruction (**adressage direct**) ou un déplacement relatif à la position courante (**adressage relatif** au PC) ou un nom de registre contenant une adresse mémoire d'instruction (**adressage indirect**)

Ces différentes méthodes de codage des informations placées dans un champ adresse sont appelées **modes d'adressages**.

Les principaux modes d'adressages de l'ARM et leur syntaxe en langage d'assemblage seront vus dans la suite du cours.

CHAPITRE 2 : Langage d'assemblage

Le langage d'assemblage qui sera utilisé est celui de l'Assembleur **as** de **GNU**.

1. Eléments du langage

1.1. Alphabet

Il est basé sur le code ASCII (codes compris entre 0 et 127).

1.2. Littéraux

littéraux numériques entiers

les différentes bases autorisées sont les bases **2(binaire)**, **8(octal)**, **10(décimal)** et **16(hexadécimal)**

le signe **+** est facultatif pour les entiers relatifs

si la base est différente de la base 10, le littéral doit être préfixée de la façon suivante :

0b ou **0B** pour la base **2** (ex : 0b11001011)

0 pour la base **8** (ex : -03257)

0x ou 0X pour la base **16** (ex : 0xF6E9)

littéraux caractères

ce sont des caractères **ASCII affichables** préfixés par une apostrophe

ex : 'A

littéraux chaînes de caractères

ce sont des **chaînes de caractères ASCII affichables** délimitées par un couple de guillemets

ex : "voulez-vous continuer ? (o,n): "

1.3. Codes opérations

Les codes opérations des instructions sont des **mots réservés** de 1 à 4 lettres qui correspondent à une abréviation anglaise de l'opération. Ils peuvent être écrits en majuscule ou en minuscule.

Exemples :

ADD ou **add** pour l'addition (add)

CMP ou **cmp** pour la comparaison (compare)

B ou **b** pour le branchement inconditionnel (branch)

1.4. Symboles

Les symboles sont des chaînes de caractères définies par le programmeur pour représenter :

- les identificateurs de variables
- les étiquettes d'instructions
- les constantes symboliques

Un symbole est une chaîne de caractères commençant par une lettre majuscule ou minuscule ou par un point. Les caractères suivants peuvent être des lettres majuscules ou minuscules, des chiffres, le point, le dollar ou le souligne.

1.5. Expressions

Expressions arithmétiques

composition de littéraux numériques et/ou de symboles liés par les opérateurs arithmétiques + - * et /
l'expression peut être parenthésée
l'expression est évaluée à l'assemblage et peut servir à faire des calculs de constantes ou d'adresses

Expressions logiques et de décalages

composition de littéraux numériques et/ou de symboles liés par les opérateurs logiques ! & | et ^
et de décalages << et >>
l'expression peut être parenthésée
l'expression est évaluée à l'assemblage et peut servir à faire des calculs de constantes

1.6. Modes d'adressages

Les principaux modes d'adressages et leur syntaxe seront vus au paragraphe 2.

1.7. Directives d'assemblage

Les directives d'assemblages sont des commandes pour l'assembleur.

Exemple : allocation mémoire et association d'un identificateur à une variable

```
compteur:   .int   0    ( variable simple entière au format 32 bits initialisée à 0 )
```

Les principales directives d'assemblages seront vues au paragraphe 3.

1.8. Autres directives

Il existe d'autres directives que ne seront pas abordées dans ce cours, telles que :

- définitions de macro-opérations (macro-assembleur)
- directives d'assemblage conditionnel
- directives de contrôles structurés

2. Codage symbolique des instructions

2.1. Format des instructions

Une instruction est codée sur une ligne de texte composée de plusieurs champs.
Les champs doivent être séparés par un ou plusieurs **espaces** ou **tabulations**.
Le format général d'une ligne est le suivant :

<étiqette> ***<code opération>*** ***<opérandes>*** ***<commentaire>***

champ *étiquette*

Ce champ est utilisé obligatoirement pour les instructions sur lesquelles sont fait des branchements.
On peut aussi l'utiliser pour des instructions sur lesquelles on ne fait pas des branchements pour marquer des structures de contrôle (par exemple le début d'une structure **si ...alors** ...).

Une étiquette doit respecter la syntaxe d'écriture des symboles (cf § 1.4).
Elle doit commencer à la première position de la ligne sinon elle doit être suivie du caractère **deux points** (**:**).

Exemples :

TQ4 CMP r1, r5

TQ4: CMP r1, r5

Si l'étiquette est précédée par un ou plusieurs **espaces** ou **tabulations** et n'est pas suivie du caractère deux points, l'assembleur considérera que le champ étiquette est **vide** et l'étiquette sera prise comme code opération.
Pour éviter tout problème, il est conseillé de faire suivre une étiquette par le caractère deux points.

Si l'instruction n'est pas étiquetée, le champ code opération doit être précédé par un ou plusieurs **espaces** ou **tabulations**, sinon le code opération sera considéré comme une étiquette.

Une étiquette peut apparaître **seule** sur une ligne et dans ce cas elle est considérée comme étiquette de l'instruction codée sur la ligne suivante.

Exemple :

```
TQ4:
        CMP     r1, r5
```

champ *code opération*

Ce champ contient le code opération symbolique de l'instruction et **ne peut donc pas être vide**.

Si l'instruction n'est pas étiquetée, le champ code opération doit être précédé par un ou plusieurs **espaces** ou **tabulations**, sinon le code opération sera considéré comme une étiquette.

Exemples :

```
        CMP     ……
        beq     ……
```

champ *opérandes*

Dans les instructions à 1, 2 ,3 ou 4 adresses, ce champ permet de désigner les opérandes de l'instruction (constante, variable, registre ou étiquette).

Dans le cas de plusieurs opérandes, ils doivent être séparés par le caractère **virgule** .

La désignation d'un opérande se fera en spécifiant le **mode d'adressage** choisi par le programmeur parmi ceux autorisés pour cette instruction.

Les noms de registres peuvent être codés en lettres majuscules ou minuscules.

Exemple :

ADD r2, r5, #1 **r2** reçoit la somme du contenu de **r5** et de la constante **1**

Les modes d'adressages et les formats autorisés pour chaque instruction seront décrits dans le fascicule de TD.

La syntaxe et la sémantique des principaux modes d'adressages seront décrites au paragraphe suivant.

champ *commentaire*

Ce champ est en général vide. Il peut être utilisé pour associer un commentaire à l'instruction.
Dans ce cas le texte du commentaire doit être précédé d'un **@**

Exemple :

ADD r2, r5, #1 @ r2 ← r5 + #1

2.2. Principaux modes d'adressages

Remarques préliminaires:

1°) La lecture ou l'écriture d'une donnée en mémoire n'est possible qu'avec les 2 instructions :

- **LDR** (load register) qui stocke dans un registre la donnée lue en mémoire.
- **STR** (store register) qui écrit en mémoire la donnée contenue dans un registre.

Donc, toutes les instructions de traitements, de comparaisons et de transferts (sauf LDR et STR) opèrent sur des données stockées dans des registres.

Toutes les instructions étant codées sur 32 bits, le champ ADRESSE des instructions LDR et STR à une longueur de 20 bits à laquelle il faut encore retrancher 4 bits pour coder le numéro du registre. Il n'est donc pas possible d'utiliser une adresse **directe absolue 32 bits** pour accéder à la mémoire.

Il faut utiliser un mode d'**adressage indirect par pointeur**.

2°) Pour les instructions de branchements B et BL, le champ ADRESSE à une longueur de 24 bits. Il n'est donc pas possible d'utiliser une adresse **directe absolue 32 bits** pour désigner l'adresse de l'instruction sur laquelle on doit faire le branchement.

Le champ ADRESSE contiendra un entier relatif en CV, égal au déplacement à effectuer pour atteindre l'instruction sur laquelle on doit faire le branchement.

3°) Il existe plusieurs autres modes d'adressages plus complexes qui ne sont pas abordés dans ce cours et qui ne seront donc pas utilisés dans les TD et les TP.

Mode d'adressage	Syntaxe	Signification
Adressage **direct** d'un registre	**Rn**	L'opérande est contenu dans le registre **Rn** Ex : CMP r2, r7 @ r2 :: r7
Adressage **indirect** de la mémoire par un registre de base	**[Rn]**	L'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn** Ex : LDR r0, [r4] @ r0 ← r4↑
Adressage **indirect** de la mémoire par un registre de base avec déplacement **constant**	**[Rn, # +/- depl]** **cf. note 1**	L'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn**, augmentée ou diminuée du déplacement **depl** Ex : LDR r0, [r4, # 8] @ r0 ← (r4+8)↑
Adressage **indirect** de la mémoire par un registre de base avec déplacement **variable** (contenu dans un registre)	**[Rn, +/- Rm]** **cf. note2**	L'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn**, augmentée ou diminuée du déplacement contenu dans le registre **Rm** Ex : LDR r0, [r4, – r5] @ r0 ← (r4 - r5)↑

Mode d'adressage	Syntaxe	Signification
Adressage **indirect** de la mémoire par un registre de base avec déplacement **constant** et mise à jour du registre de base **postérieure** à l'accès mémoire	**[Rn], # +/- depl** **cf. note1**	L'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn**. Après l'accès mémoire, **Rn** est augmentée ou diminuée du déplacement **depl** Ex : LDR r0, [r4], # 8 @ r0 ← (r4)↑ @ puis r4 ← r4+8
Adressage **indirect** de la mémoire par un registre de base avec déplacement **variable** et mise à jour du registre de base **postérieure** à l'accès mémoire	**[Rn], +/- Rm** **cf. note2**	L'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn** Après l'accès mémoire, **Rn** est augmentée ou diminuée du déplacement contenu dans **Rm** Ex : LDR r0, [r4], – r5 @ r0 ← (r4)↑ @ puis r4 ← r4 – r5

Mode d'adressage	Syntaxe	Signification
Adressage **indirect** de la mémoire par un registre de base avec déplacement **constant** et mise à jour du registre de base **antérieure** à l'accès mémoire	**[Rn, # +/- depl]!** **cf. note1**	Avant l'accès mémoire, **Rn** est augmentée ou diminuée du déplacement **depl** Ensuite l'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn** Ex : LDR r0, [r4, # 8]! @ r4 ← r4+8 @ puis r0 ← (r4)↑
Adressage **indirect** de la mémoire par un registre de base avec déplacement **variable** et mise à jour du registre de base **antérieure** à l'accès mémoire	**[Rn, +/- Rm]!** **cf. note2**	Avant l'accès mémoire, **Rn** est augmentée ou diminuée du déplacement contenu dans **Rm** Puis l'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn** Ex : LDR r0, [r4, – r5]! @ r4 ← r4 – r5 @ puis r0 ← (r4)↑

Mode d'adressage	Syntaxe	Signification
Adressage **indirect** de la mémoire par un registre de base avec déplacement variable mis au facteur d'échelle	**[Rn, +/- Rm, LSL # p]** **cf. note3**	L'opérande est rangé dans la mémoire à l'adresse contenue dans le registre **Rn**, augmentée ou diminuée du déplacement contenu dans le registre **Rm** préalablement décalé de **p** positions à gauche. Ex : LDR r0, [r4, – r5, LSL # 2] @ r0 ← (r4 - r5*4)↑
Adressage **relatif** à la position courante	***etiquette***	Utilisé uniquement dans les instructions de branchement L'opérande est un déplacement relatif égal à la différence entre l'adresse de l'étiquette ***etiquette*** et l'adresse de l'instruction suivante (contenu du PC) Ex : BNE FSI2
Adressage **immédiat**	***#expression***	L'opérande est une **constante** égale à la valeur de *expression* Ex : LDR r0, #1 @ r0 ← #1

Mode d'adressage	Syntaxe	Signification
Sauvegarde ou restauration de plusieurs registres en mémoire	**Rp!, {R_i, R_j, R_k, ...}** si le registre Rp n'est pas suivi du caractère **!** il reprend sa valeur initiale quand tous les transferts sont terminés sinon il conserve la valeur qui a été calculée lors du dernier transfert	Les registres de la liste sont sauvegardés (STM) par numéro croissant ou restaurés (LDM) par numéro décroissant à partir de l'adresse mémoire contenue dans le registre **Rp** le registre **Rp** qui est utilisé comme pointeur, peut être incrémenté ou décrémenté et ceci avant ou après l'accès mémoire. Ces modes de transferts sont codés par les 4 suffixes suivants qu'on ajoute aux instructions LDM ou STM : – **IA** ≡ **incrémenter** Rp **après** l'accès mémoire – **IB** ≡ " " **avant** " – **DA** ≡ **décrémenter** Rp **après** l'accès mémoire – **DB** ≡ " " **avant** " On verra au chapitre 3, l'utilisation des instructions LDM et STM et de ces différents modes de transferts pour gérer une structure de pile en mémoire.

Note 1 :

- ce mode d'adressage est utilisé uniquement dans les instructions **load** et **store**.
- si l'opérande en mémoire est un mot de **32** bits (LDR ou STR) ou un octet **non signé** (LDRB ou STRB), le déplacement **depl** est un relatif **12** bits ($-2^{11} \leq$ **depl** $\leq +2^{11}-1$).
- si l'opérande en mémoire est un mot de **16** bits (LDRH, LDRSH ou STRH) ou un octet **signé** (LDRSB), le déplacement **depl** est un relatif **8** bits ($-2^{7} \leq$ **depl** $\leq +2^{7}-1$).

Note 2 :

- ce mode d'adressage est utilisé uniquement dans les instructions **load** et **store**.
- le déplacement variable contenu dans le registre est un relatif 3**2** bits ($-2^{31} \leq$ **depl** $\leq +2^{31}-1$).

Note 3 :

- ce mode d'adressage est utilisé uniquement dans les instructions **load** et **store** pour les formats 32 bits et 8 bits non signé.
- le nombre de positions de décalages **p** est un naturel **5** bits ($0 \leq$ **p** ≤ 31).
- tous les types de décalages sont possibles, à droite ou à gauche (LSL, LSR, ASR, ROR, ROX).

3. Spécification des principales instructions

La spécification des principales instructions sera présentée en TD avec des exercices de mise en œuvre.

4. Directives d'assemblage

4.1. Directives d'allocation de la mémoire

Ces directives permettent d'allouer de la mémoire pour les variables et les constantes, de leur associer un identificateur et éventuellement d'initialiser des variables.

Remarques préliminaires

1°) Dans les machines qui gèrent plusieurs formats de données, plusieurs contraintes doivent être respectées pour que l'accès mémoire à une donnée soient faits en un seul cycle :

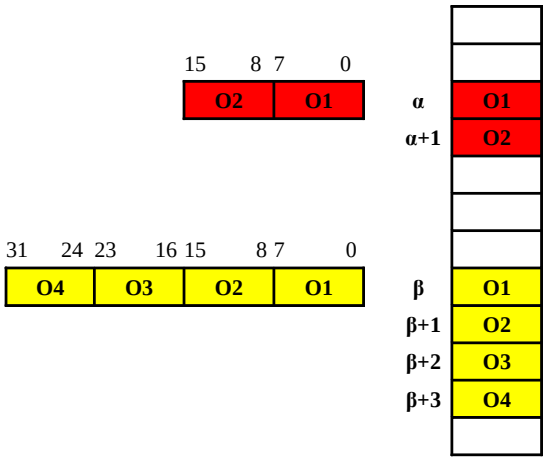
- les formats des données doivent être des puissances de 2 fois un octet (8, 16, 32, 64, ... bits)
- la mémoire doit être organisée en octets et donc les adresses sont des **adresses d'octets**
- les données codées sur **16 bits** (**2** octets) doivent être rangées en mémoire à des **adresses multiples de 2** (on dit que la donnée est **alignée** sur une limite de 2 octets)
- les données codées sur **32 bits** (**4** octets) doivent être rangées en mémoire à des **adresses multiples de 4** (on dit que la donnée est **alignée** sur une limite de 4 octets)
- etc ...

2°) Il existe deux modes d'organisation en mémoire des données constituées de plusieurs octets :

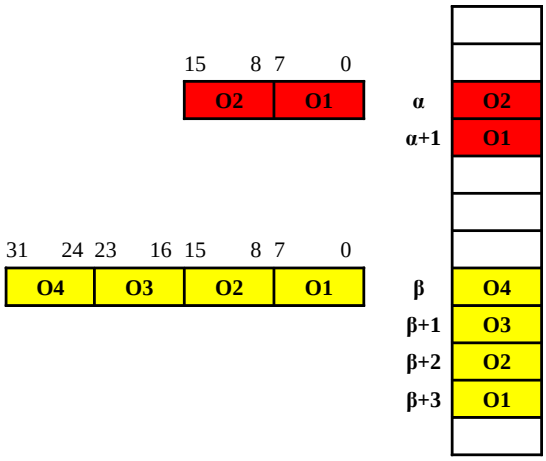
- le mode "**little endian** " dans lequel les octets sont stockés en séquence par **adresses croissantes** en commençant par l'octet de poids faible (octet le moins significatif)
- le mode "**big endian** " dans lequel les octets sont stockés en séquence par **adresses croissantes** en commençant par l'octet de poids fort (octet le plus significatif)

L'ARM peut fonctionner dans chacun de ces deux modes. Le choix est fixé par l'état du signal d'entrée de la broche **CFCBIGEND** du microprocesseur (1 pour "**big endian** ", 0 pour "**little endian** ").

L'émulateur utilisé pour les TP, émule une machine Linux, basée sur un microprocesseur **ARM7** en mode "**little endian** ".



Ex : mode "**little endian** "



Ex : mode "**big endian** "

Alignement de l'adresse mémoire courante

Syntaxe :

.align ***n***

Fonction :

aligne l'adresse mémoire courante sur le multiple de 2^n immédiatement supérieur.

Exemple :

.align **2** **@** aligne l'adresse mémoire courante sur le multiple de 2^2 (4)
@ immédiatement supérieur

Allocation mémoire de variables

Syntaxe :

nomVar **.space** ***taille***

Fonction :

Cette directive alloue un bloc de ***taille*** octets en mémoire à l'adresse courante.

Allocation mémoire de variables avec initialisation

Syntaxe :

nomVar **.byte** ***expression_1, expression_2, ..., expression_n***
nomVar **.hword** ***expression_1, expression_2, ..., expression_n***
nomVar **.word** ***expression_1, expression_2, ..., expression_n***

Fonction :

Ces directives allouent respectivement un bloc de mots de 8 bits (**.byte**), 16 bits (**.hword**) ou 32 bits(**.word**), en mémoire à l'adresse courante et initialise séquentiellement ces mots avec les valeurs ***expression_1, expression_2, ..., expression_n***.

S'il n'y a qu'une expression, ***nomVar*** est une variable élémentaire sinon c'est un tableau.

On peut utiliser **.short** à la place de **.hword** et **.long** ou **.int** à la place de **.word**.

Chaque ***expression_i*** doit être une expression arithmétique entière dont la valeur est comprise entre :

- **0** et $\mathbf{2^n-1}$ pour un **entier naturel**.
- $\mathbf{-2^{n-1}}$ et $\mathbf{+2^{n-1}-1}$ pour un **entier relatif**.

Pour n=8, 16 ou 32 suivant le format des mots de 8 bits (**.byte**), 16 bits (**.hword**) ou 32 bits (**.word**)

Exemples :

tab **.int** **123, -456, +789, 10*(45 - 8) + 23**
liste **.byte** **0, 1, 2, 255**

Syntaxe :

nomVar **.fill** ***nombre, taille, expression***

Fonction :

Cette directive alloue ***nombre*** blocs de ***taille*** octets en mémoire à l'adresse courante et initialise chaque bloc avec la valeur ***expression***.

Le paramètre ***expression*** peut être omis, dans cas chaque bloc est initialisé avec la valeur 0.
Le paramètre ***taille*** ne peut pas être supérieur à **8** (64 bits), s'il est omis, la taille des blocs est de **1** octet.

Exemple :

tab: **.fill** **100, 4, 0** **@** allocation d'un tableau de 100 entiers 32 bits initialisés à 0;

Allocation mémoire d'une chaîne de caractères ASCII

Syntaxe :

nomChaine **.ascii** ***"chaîne_de_caractères_ASCII"***
nomChaine **.asciz** ***"chaîne_de_caractères_ASCII"***

Fonction :

Ces directives allouent un bloc d'octets en mémoire à l'adresse courante et initialisent séquentiellement chaque octet avec chaque caractère de ***chaîne_de_caractères_ASCII***.

La directive **.asciz** rajoute en plus le code ASCII <NUL> (0) en fin de chaîne.

Exemples :

ERR_PARAM **.asciz** **"Erreur de parametre"** **@** chaîne de caractères de type langage C

4.2. Déclaration de constantes symboliques

Syntaxe :

.equiv ***symbole,expression***
.equ ***symbole,expression***
.set ***symbole,expression***

Fonction :

Ces directives affectent la valeur de ***expression*** au symbole ***symbole***.
Avec la directive **.equiv** la valeur du symbole ne peut plus être modifiée.
Les directives **.equ** et **.set** sont équivalentes et permettent de modifier la valeur du symbole ailleurs dans le fichier par d'autres directives **.equ** ou **.set**.

Exemples :

.equ **LG_MAX, 100** **@** définition d'une constante symbolique égale à 100.

.equiv **LF, 0x0D** **@** définition d'une constante symbolique égale au caractère ASCII <LF>
@ équivalent à **\n** en C .

…… utilisation ……

tab: **.space** **LG_MAX * 4** **@** allocation d'un tableau de 100 entiers au format 32 bits.

cmp **r5, #LF** **@** comparer le contenu de r5 avec le caractère ASCII <LF>

4.3. Directives pour l'édition de liens

Syntaxe :

.text
.bss
.data
.rodata

Fonction :

Un programme exécutable est constitué d'instructions et de données.

La directive **.text** indique à l'assembleur de ranger la séquence d'instructions qui suit, une fois assemblées, dans une zone du fichier objet relogeable nommée **section text**.

Dans le système Unix/Linux, les fichiers objets relogeables ont le suffixe "**.o**" .

Les directives **.bss**, **.data** ou **.rodata** indiquent à l'assembleur de ranger les blocs d'octets alloués aux données par la séquence de directives d'allocation qui suit, dans une zone du fichier objet relogeable nommée respectivement **section bss, data** ou **rodata**.

La directive **.bss** est utilisée pour les variables non initialisées, la directive **.data** est utilisée pour les variables initialisées et la directive **.rodata** (read-only data) est utilisée pour les constantes.

Dans la plupart de cas, l'**éditeur de lien** (linker) doit fusionner plusieurs fichiers **objets relogeables** (**.o**) pour générer le fichier **objet exécutable** (par ex: le fichier **.o** du programme principal et les différents fichiers **.o** des fonctions ou procédures associées).

L'**éditeur de lien** fusionnera toutes les sections **text** des fichiers **.o** dans la section **text** du fichier **objet exécutable**, toutes les sections **data** des fichiers **.o** dans la section **data** du fichier **objet exécutable** et toutes les sections **rodata** des fichiers **.o** dans la section **rodata** du fichier **objet exécutable**

Dans le système Unix/Linux, pour lancer l'exécution d'un programme, les instructions stockées dans la **section text** du fichier **objet exécutable** seront chargées dans un segment de mémoire appelé "**text**". Ce segment "**text**" est **protégé en écriture**, pour interdire de modifier les instructions d'un programme en cours d'éxécution.

Les données stockées dans la **section data** du fichier **objet exécutable** seront chargées dans un segment de mémoire appelé "**data**".

On verra au chapitre suivant, qu'un programme a aussi besoin d'un segment de mémoire structuré et géré comme une **pile** (stack en anglais).

Le système Unix/Linux comme d'autres systèmes permet l'**allocation dynamique** de variables. Donc un programme aura aussi besoin d'un segment de mémoire pour stocker les variables dynamiques. Ce segment est appelé **tas** (heap en anglais).

Remarques :

- Le segment **text** ne contient normalement que des instructions. Toutefois il est possible, en prenant certaines précautions, d'allouer des **constantes** dans ce segment.

- Les modes d'adressages de l'ARM imposent d'allouer des **pointeurs constants** dans le segment **text** pour accéder aux variables allouées dans le segment **data**.

Syntaxe :

.global ***liste-de-symboles***

Fonction :

Cette directive indique à **l'éditeur de liens** que les symboles figurant dans la liste ***liste-de-symboles*** pourront être utilisés dans d'autres unités de compilation.
Ils sont alors **globaux** à toutes les unités de compilation.

Ces symboles sont en général des identificateurs de variables ou des étiquettes d'instructions.

Les symboles figurant dans la liste ***liste-de-symboles*** doivent être tous déclarés dans la même unité de compilation que la directive **.global**.

Par contre, les symboles ne figurant dans la liste ***liste-de-symboles*** sont locaux à l'unité de compilation.

Plusieurs directives **.global** peuvent être utilisées suivant les besoins.

Exemple :

Cas d'un programme en C appelant un sous-programme en Assembleur ARM.

Le programme en C est codé dans le fichier **prog.c** et le sous-programme en Assembleur dans le fichier **lireEntier.s**.

Fichier **prog.c**

```
# include   <…….>
……….

extern  lireEntier( int * val ) ;

……..

int main( )
{
int nombre ;

……..

lireEntier( & nombre) ;

……..

}
```

Fichier **lireEntier.s**

```
@ sous-programme lireEntier
      .global      lireEntier
……..
    .text
lireEntier:
…….....
……
……
……
……
……
     .data
.....
val:     .int   0
```

4.4. Directives de contrôle

Syntaxe :

.include ***"désignation-de-fichier"***

Fonction :

Cette directive permet d'insérer dans le fichier source, à partir de la ligne qui la contient, le contenu du fichier nommé ***désignation-de-fichier***.

Ce fichier doit être un fichier texte contenant des directives et/ou des instructions en Assembleur ARM.