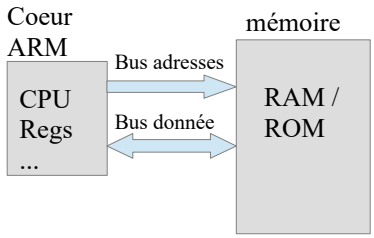
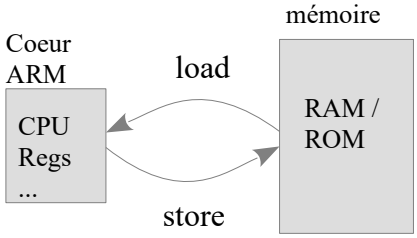
Variables, accès mémoire en asm pour ARM

Généralités

L'architecture ARM est de type **load / store** : la lecture en mémoire se fait par une instruction *ldr*, l'écriture par une instruction *str* (écriture uniquement en RAM).
Par lecture on entend un transfert de donnée en mémoire vers un registre interne. Un écriture est un transfert de donnée dans un registre vers la mémoire.



Architecture très simplifiée



Flux load / store

Le CPU utilise les instructions *ldr* (il en existe plusieurs) pour **aller chercher les données** à manipuler. Il **exploite ces données localement** via son jeu d'instructions. Il **stocke les résultats en mémoire** avec les instructions *str*.

En assembleur, les types de variable (int, char...) **n'existent pas**. On parle en assembleur de données 8 bits, 16 bits ou 32 bits, les données sont donc subdivisée non pas en type, mais en **taille de données**.

L'architecture ARM utilisée permet de lire et d'écrire des données :

- **32** bits avec les instructions ***ldr / str***
- **16** bits avec les instructions ***ldrh/strh*** (il existe aussi ***ldrsh*** pour accéder à une donnée 16 bits signée)
- **8** bits avec les instructions ldrb/strb (il existe aussi ***ldrsb*** pour accéder à une donnée 8 bits signée)

Exemple :
R0 = 0x12ABD5EF
R1 = 0x2000000
str R0,[R1] donne après exécution :

Mémoire (cases de 8 bits)

0x2000 0000	0xEF
	0xD5
	0xAB
	0x12

ldrh R2,[R1] donne après exécution :

R2 | 0x0000D5EF

NB : les 16 bits de poids fort sont forcés à 0.

ldrsb R2,[R1] donne après exécution :

R2 | 0xFFFFFFEF

NB : les 24 bits de poids fort sont positionnés à 1, c'est l'extension de signe (la donnée 32 bits est négative tout comme la donnée 8 bits d'origine)

Les variables en C, en ASM, la réservation mémoire

32 bits (exemple type int)

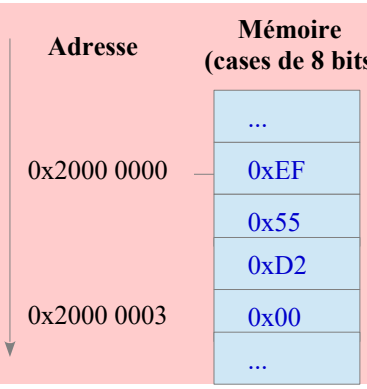
En langage C type int

```
int VarInt = 0xD255EF ;
```

En assembleur taille 32bits

```
area    Ma_RAM,data,readwrite
VarInt dcd 0xD255EF
```

→ l'adresse de la variable VarInt est 0x2000 0000 (voir ci-contre)
→ la valeur de la variable VarInt est 0xD255EF



16 bits (exemple type short int)

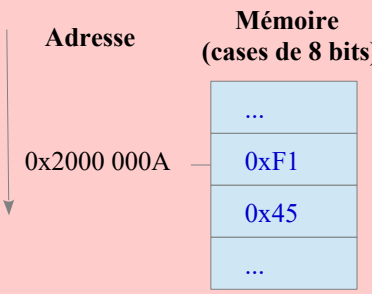
En langage C type short int

```
short int VarShort = 0x45F1 ;
```

En assembleur taille 16bits

```
area    Ma_RAM,data,readwrite
VarShort dcw 0x45F1
```

→ l'adresse de la variable VarShort est 0x2000 000A (voir ci-contre)
→ la valeur de la variable VarShort est 0x45F1



8 bits (exemple type char)

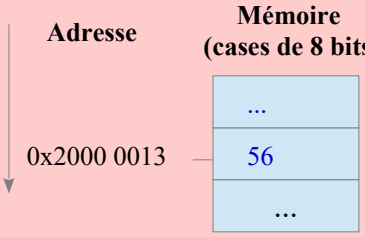
En langage C type char

```
char VarChar = 56 ;
```

En assembleur taille 8bits

```
area    Ma_RAM,data,readwrite
VarChar dcb 56 ; on peut utiliser le décimal
```

→ l'adresse de la variable VarChar est 0x2000 0013 (voir ci-contre)
→ la valeur de la variable VarChar est 56



→ pour les datas 32 bits et 16 bits, **l'octet de poids faible est rangé en premier (little endian)**.

→ L'adresse d'une donnée 32 bits ou 16 bits correspond donc aussi à l'adresse de l'octet de poids faible de la donnée.

→ *area , dcd, dcw, dcb* sont des ***directives*** (≠ instructions asm) voir Moodle pour une liste complète.
→ en asm, une définition de variable (réservation mémoire) se fait dans une zone (area) de type RAM, cad, *data, readwrite*.

VarInt nécessite donc 4 cases mémoire consécutives pour y stocker sa valeur 32 bits.
VarShort nécessitera 2 cases, et VarChar une seule.

Les 3 instructions ASM fondamentales : *mov, ldr, str*

→ l'instruction mov (pas d'accès mémoire!)

mov R0,R1 : la valeur contenue dans R1 est copiée dans R0
mov R0, #758 : R0 prend la valeur 758 (la valeur peut s'étendre sur 16bits)

NB : voir *PM0056.pdf* pour plus d'options de l'instruction mov.

→ l'instruction ldr (load)

ldr R0,[R1] : la valeur 32bits située à l'adresse spécifiée dans R1 est recopiée dans R0
ldr R0,=0xAABBCCDD : R0 prend la valeur 0xAABBCCDD

NB : cette dernière instruction est en fait une « macro instruction » équivalente à un mov R0,#, sauf qu'ici, la valeur s'étend au delà de 16 bits ! Ainsi, ***ldr Ri, = …*** est à penser comme un ***mov*** !
NB : voir *PM0056.pdf* pour plus d'options de l'instruction ldr (notamment pour aller chercher des valeurs 16 bits ou 8 bits, signée ou pas en mémoire, RAM ou ROM)

→ l'instruction str (store)

str R0,[R1] : la valeur 32bits contenue dans R0 est copiée en RAM à l'adresse spécifiée dans R1

NB : voir *PM0056.pdf* pour plus d'options de l'instruction str (notamment pour écrire des valeurs 16 bits ou 8 bits, en mémoire, RAM only !)

Affecter une variable en assembleur, exemple ...

En C, on écrit par exemple :
short int Var ;
Var=0xAABB ;

En assembleur
1- il faut écrire l'adresse de la variable dans une registre, par exemple R1,
2- on utilise l'instruction *strh* puisque la taille de la donnée à écrire est sur 16 bits.

Cela donne :

```
ldr R1,=Var ; Var étant la variable définie, Var est forcément 32bits donc ldr R1, =, pas mov # !
mov R0,# 0xAABB ; si la valeur était sur 32 bits, on utiliserait ldr R0, =. Ici 16 bits donc
                ;     mov R0,#... suffit
strh R0,[R1]
```

→ on note que ***Var*** désigne à la fois **la variable *Var*** mais aussi **son adresse** en assembleur. Ici, ***Var*** est *équivalent* à son adresse, par exemple 0x2000 0000. Dans le code précédent, R1 contiendrait donc 0x2000 0000
→ Quelque soit le type de variable en C, ou sa taille en asm, une adresse est TOUJOURS sur 32bits !

Lire une variable en assembleur ...

1- il faut écrire l'adresse de la variable dans un registre, par exemple R1,
2- on utilise l'instruction *ldr*. ou *ldrh, ldrsh, ldrb, ldrsb* selon le cas...

Cela donne :

```
ldr R1,=Var
ldrb R0,[R1] ; la valeur de Var est stockée dans le registre R0 (ici, on lit un char...)
```