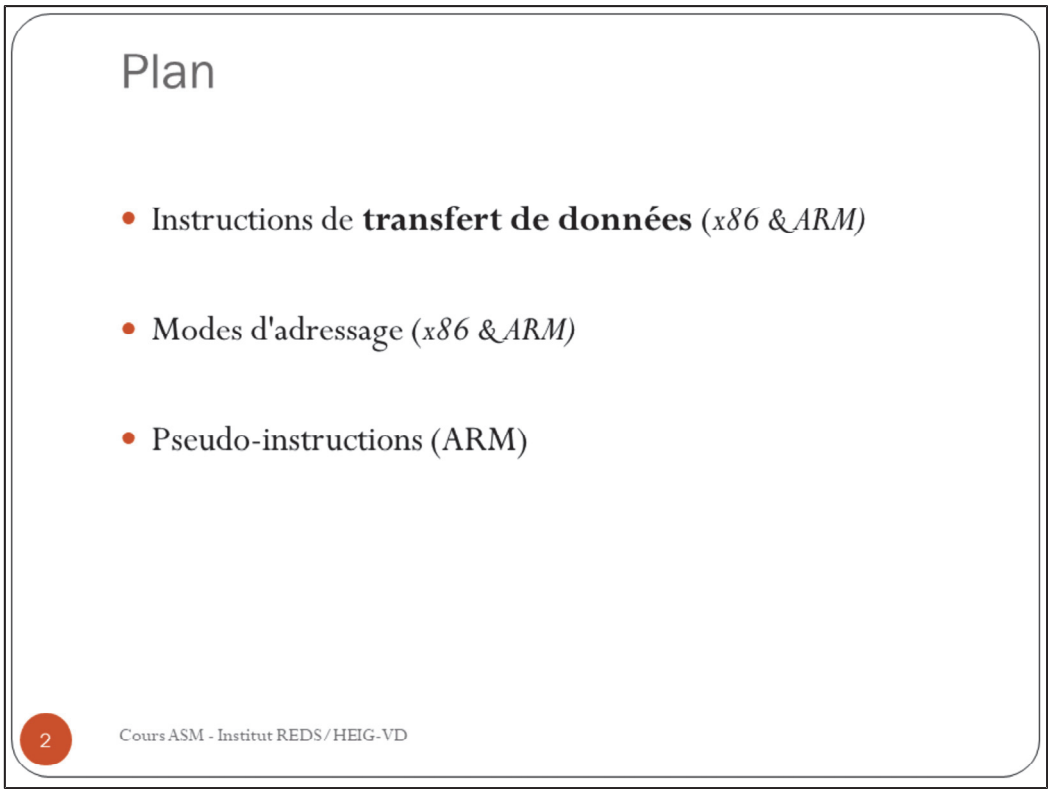
REDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Programmation assembleur (ASM)
Instructions de transfert

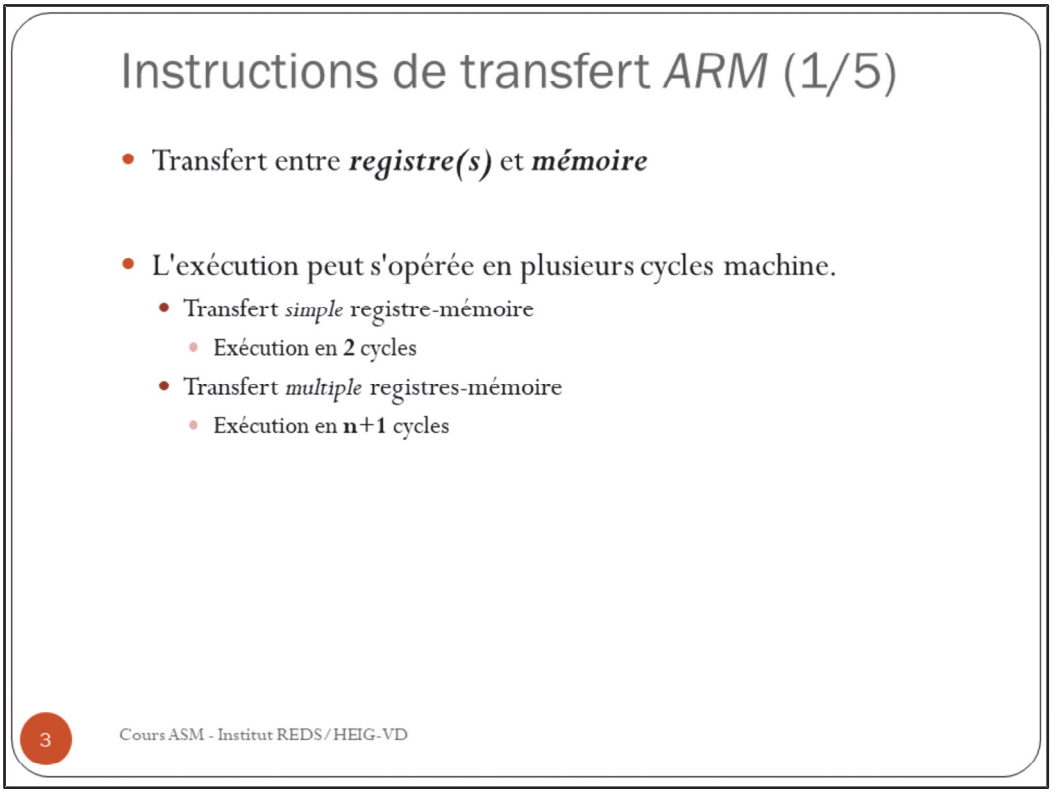
Prof. Daniel Rossier

Version 1.7 (2017-2018)



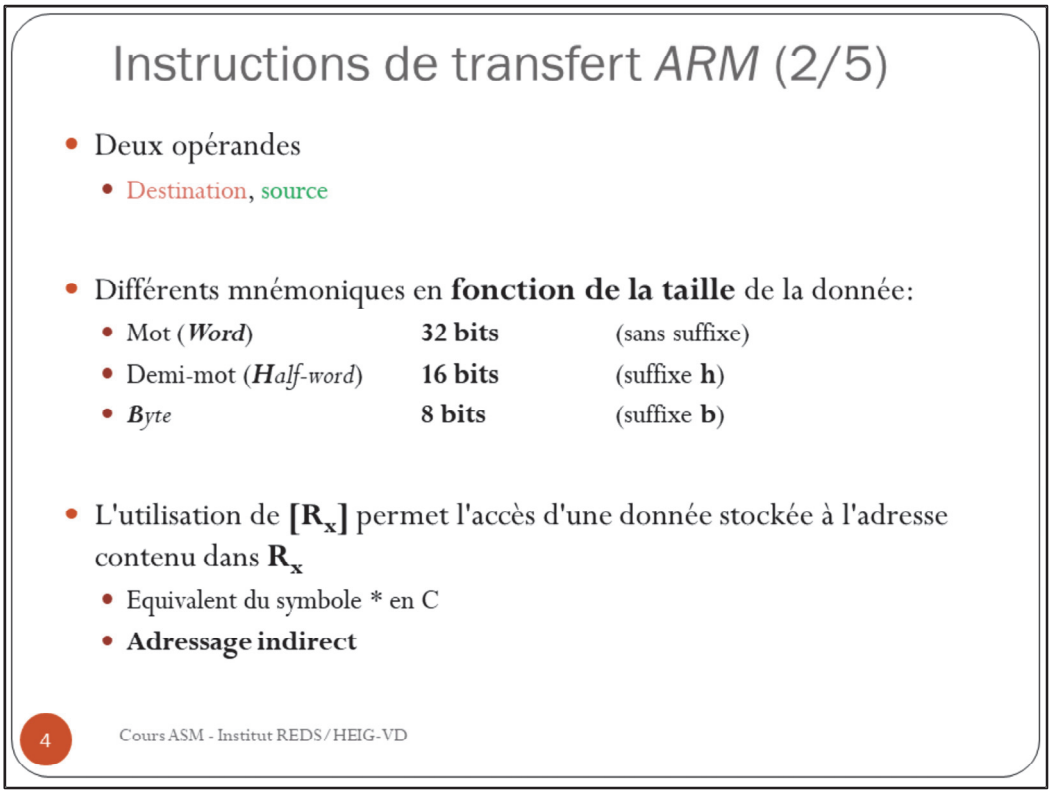
Ce chapitre est consacré à l'étude des instructions de transfert des processeurs *x86* et *ARM.* Cette catégorie d'instructions permet de transférer des données d'une taille d'un à quatre octets typiquement entre la mémoire et les registres du processeur.

La notion de *mode d'adressage* et de *pseudo-instruction* sera également examinée; les modes d'adressage permettent de faciliter l'accès aux données et de réduire ainsi le nombre d'instructions.

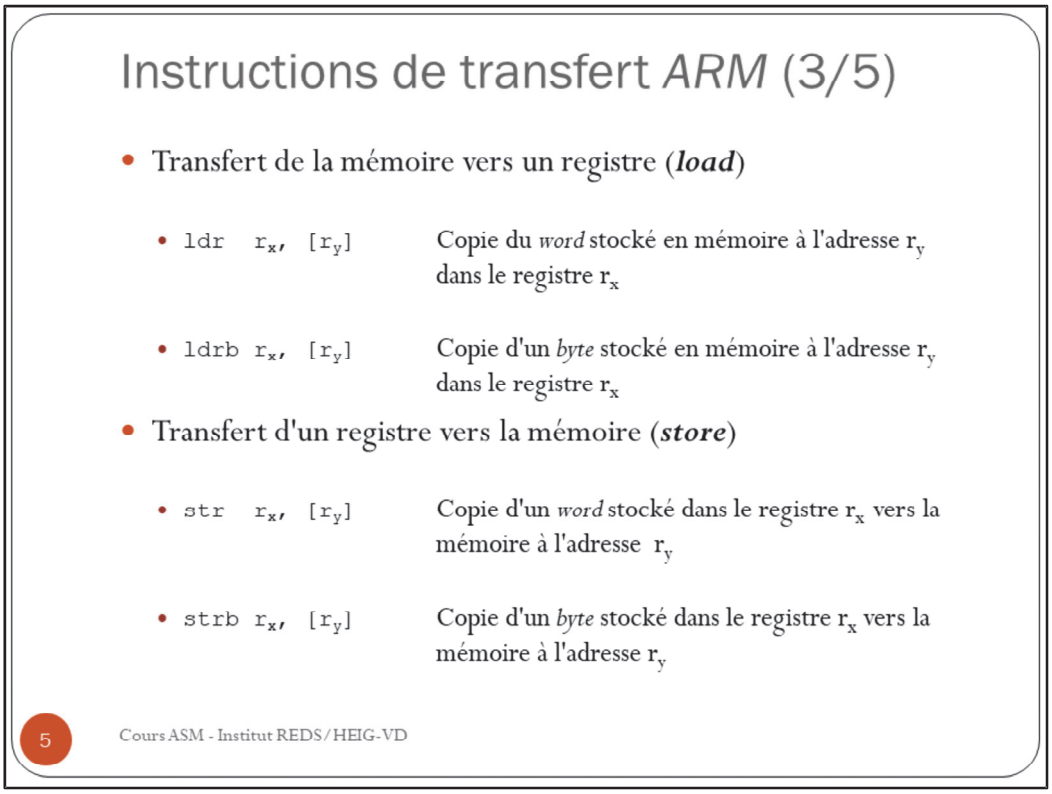


L'instruction de transfert possède une ou plusieurs opérandes, et dans le cas où l'adresse fait partie directement d'une opérande, on parle alors **d'adressage direct**. A l'inverse, si l'opérande d'une instruction est un registre contenant l'adresse, **l'adressage sera indirect**.

Les instructions de transfert sur *ARM* permettent de transférer des données entre registres et mémoire. L'exécution des instructions de transfert peuvent nécessiter plusieurs cycles d'horloge en fonction du nombre de registres impliqués dans le transfert.

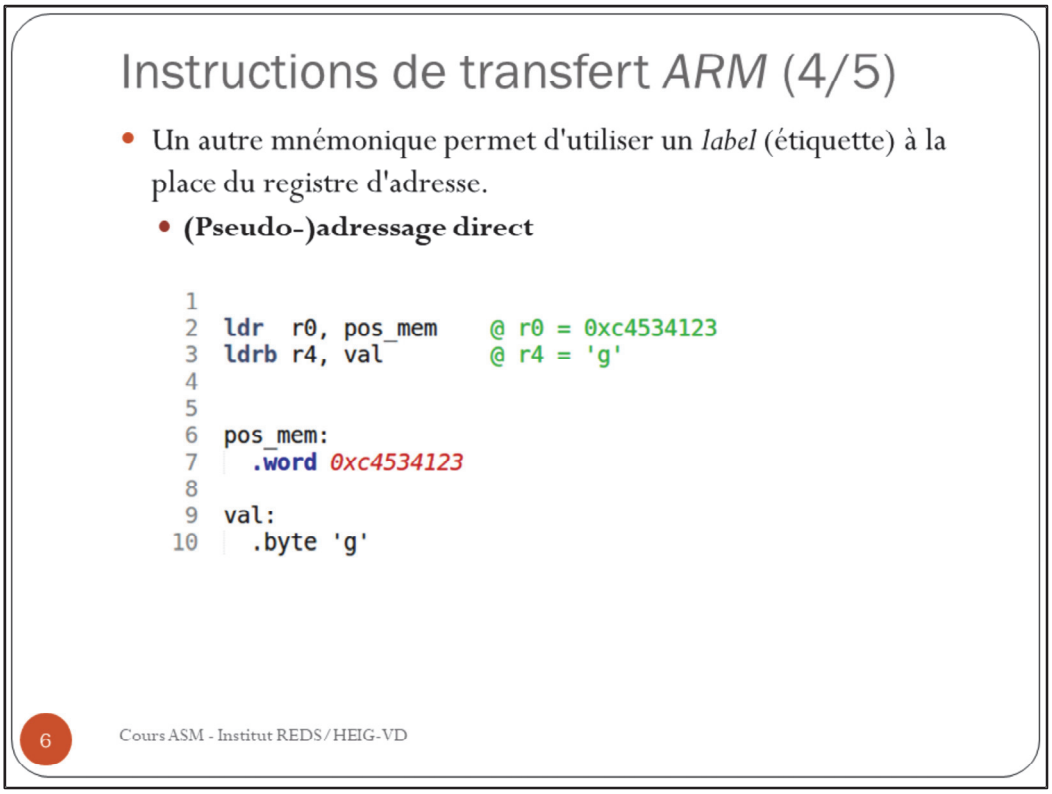


En assembleur *ARM*, c'est le crochet ouvert-fermé encadrant un registre, qui dénote l'accès mémoire. Ce mode d'adressage est de type indirect puisque l'opérande est constituée d'un registre contenant l'adresse.



Les instructions *ldr* et *str* sont les deux instructions principales permettant d'accéder la mémoire, sur *ARM*. Elles sont simples d'utilisation : l'une des opérandes se réfère à un **registre**, l'autre se réfère à une **donnée en mémoire** (via son adresse). Ce mécanisme d'accès mémoire est aussi connu sous le nom de *load-store*.

La taille de la donnée transférée est spécifiée par la présence d'un suffixe aux instructions *ldr* et *str.* L'absence de suffixe signifie un transfert sur 32 bits.



Dans le cas d'un adressage direct, l'opérande peut contenir l'adresse **via un label**; contrairement au *x86*, il n'est pas possible de spécifier une adresse en valeur immédiate.

Nous verrons par la suite que ce type d'instruction, supportée par le compilateur, est traduite en différente(s) instruction(s) de base; on parle dans ce cas de ***pseudo-instruction***. Les pseudo-instructions ne font pas partie du jeu d'instructions de base. En fait, ce dernier ne supporte que le mode d'adressage indirect.

Instructions de transfert *ARM* (5/5)

- Soit les variables entières signées **x, y, z**.
  - **x** dans l'adresse mémoire **0x010**
  - **y** dans l'adresse mémoire **0x014**
  - **z** dans l'adresse mémoire **0x018**

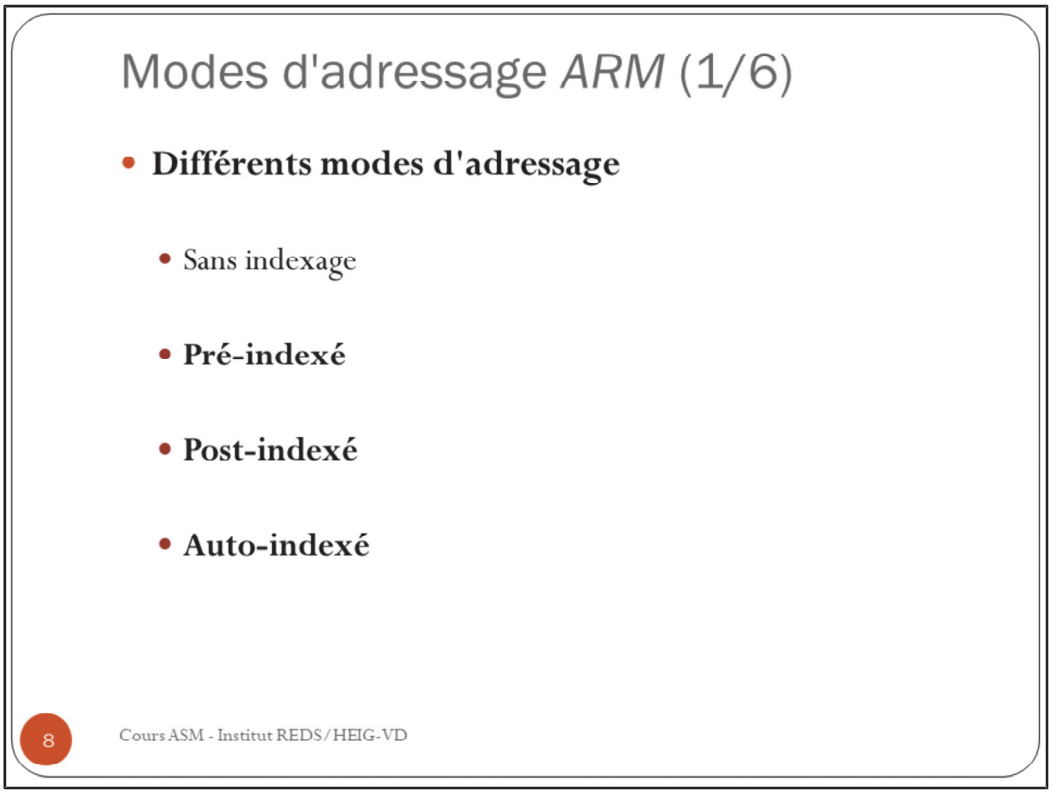
⇒ Codez en assembleur l'une ou l'autre des opérations ci-dessous.

```
@ z = x + y + 4
```

```
@ z = 3x – 5
```

```
@ z = (x + y) / 4
```

```
@ z = 4x + 3
```



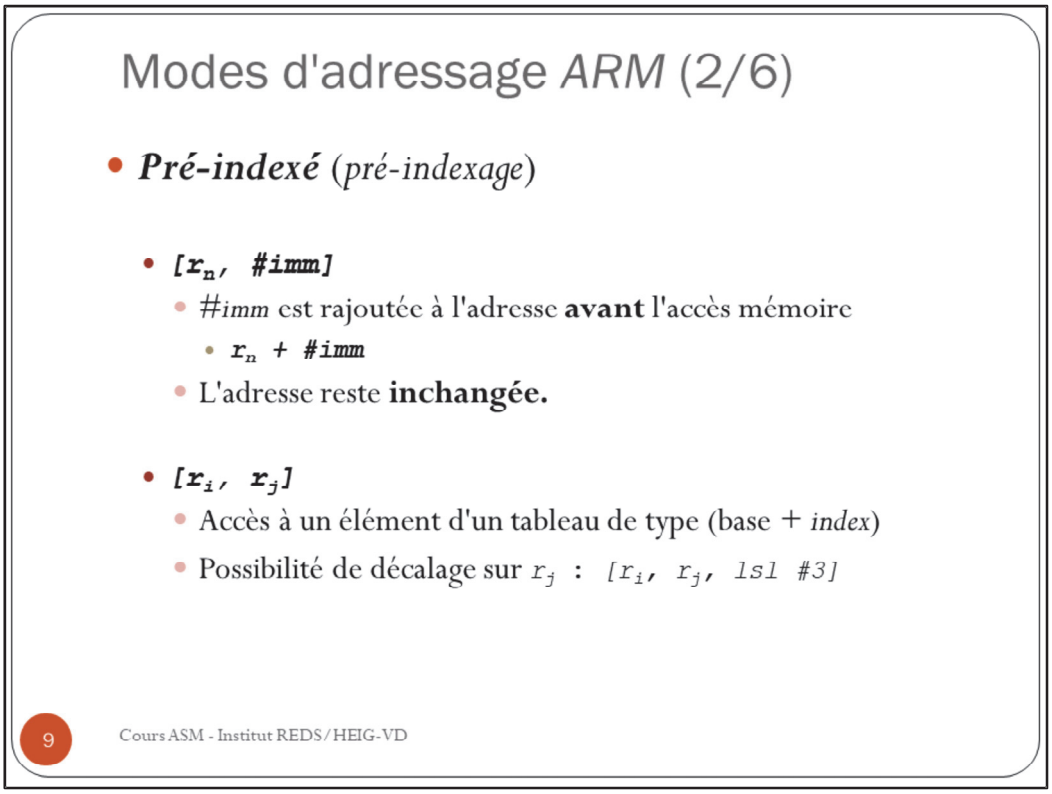
Le transfert d'une valeur de la mémoire vers un registre, ou vice-versa, s'effectue à l'aide d'une adresse mémoire indiquant la position de la donnée en mémoire.

Généralement, un programme devra transférer plusieurs données contiguës, représentant un tableau par exemple. Dans ce cas, l'instruction de transfert est identique, seule l'adresse varie. A cet effet, l'adresse devra être incrémentée (ou décrémentée), ce qui nécessite l'utilisation d'une instruction d'addition (ou de soustraction). C'est pourquoi, les instructions de transfert *ARM* autorisent différents modes d'indexage ayant pour effet de manipuler l'adresse **dans le même cycle d'instruction** que l'instruction de transfert.

En fonction du déplacement en mémoire, l'utilisation judicieuse d'un mode d'indexage permettra ainsi l'addition ou la soustraction, avant ou après l'exécution du transfert.

Comme nous allons le découvrir, une troisième opérande aux instructions de transfert permettra de contrôler l'indexage.

Ces différents modes sont maintenant examinés en détail.

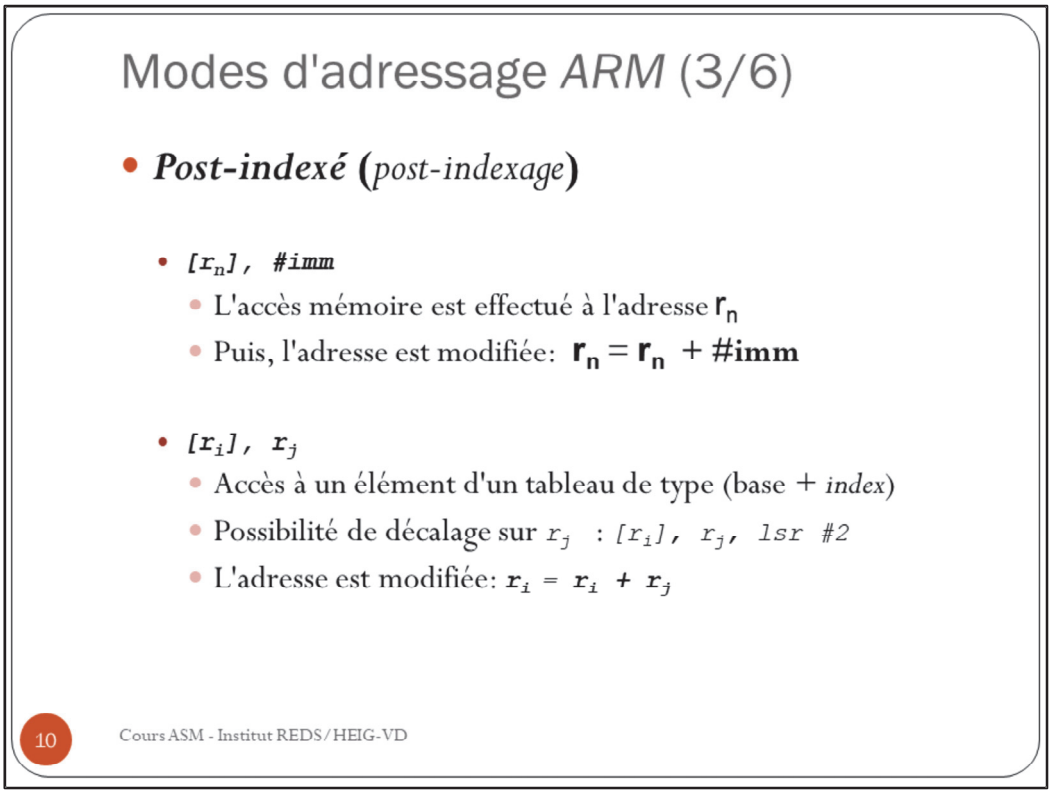


La troisième opérande – dans les crochets – permet d'indiquer la valeur du déplacement. Cette opérande peut être soit une valeur immédiate, soit un registre.

Le mode d'indexage *pré-indexé* additionne la troisième opérande à la seconde.

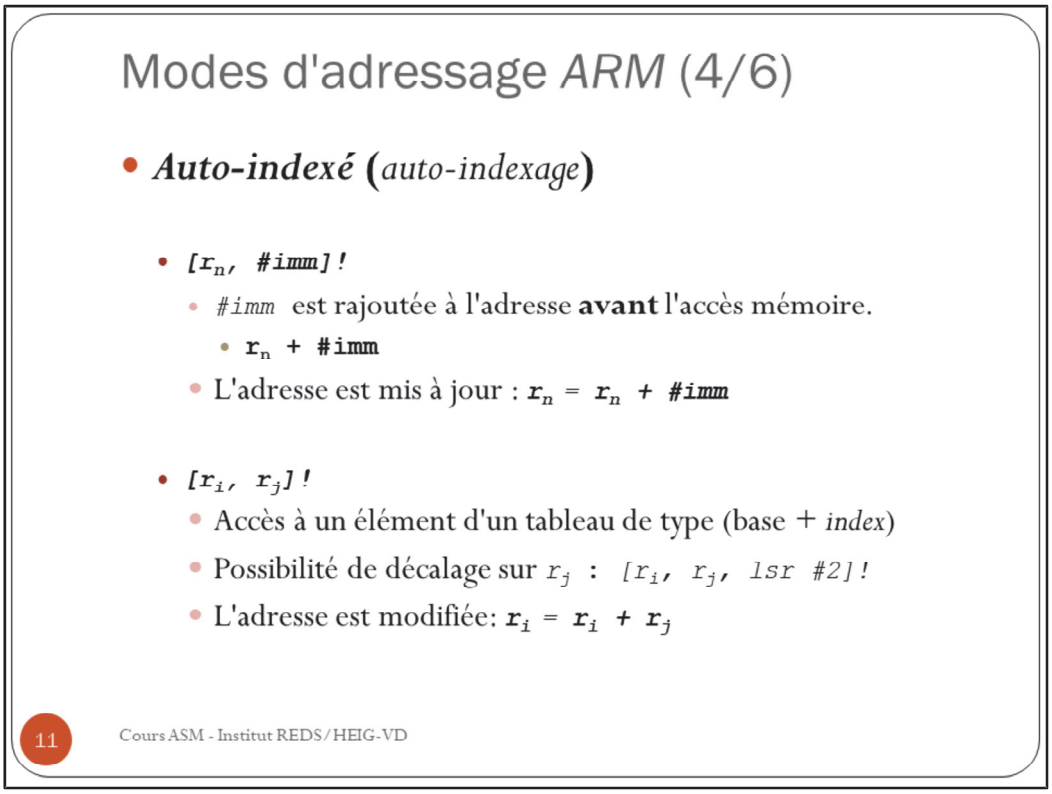
Il est également possible de rajouter une opération de décalage sur la troisième opérande de telle manière à pouvoir utiliser l'effet d'un facteur multiplicatif sur celle-ci.

Ce mode ne modifie pas l'adresse contenu dans le registre.

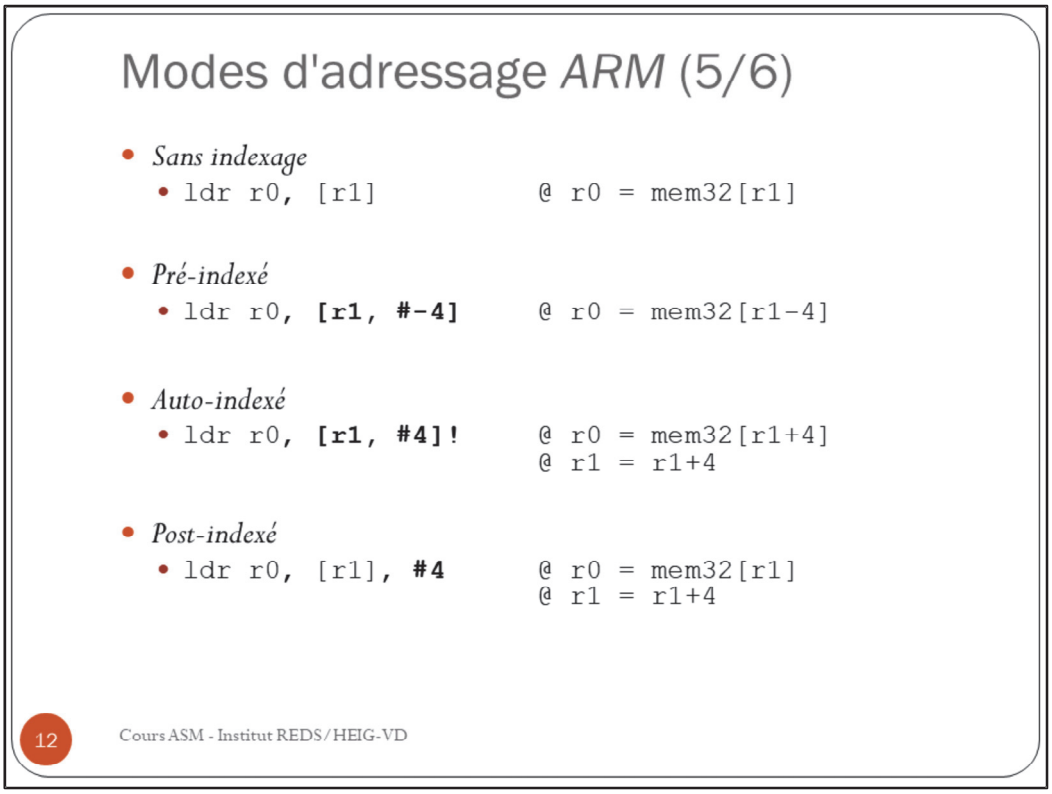


Le mode d'adressage *post-indexé* effectue l'adresse mémoire **avant de mettre à jour** le registre de la seconde opérande.

Comme les modes précédents, un registre à décalage peut être utilisé dans la troisième opérande.

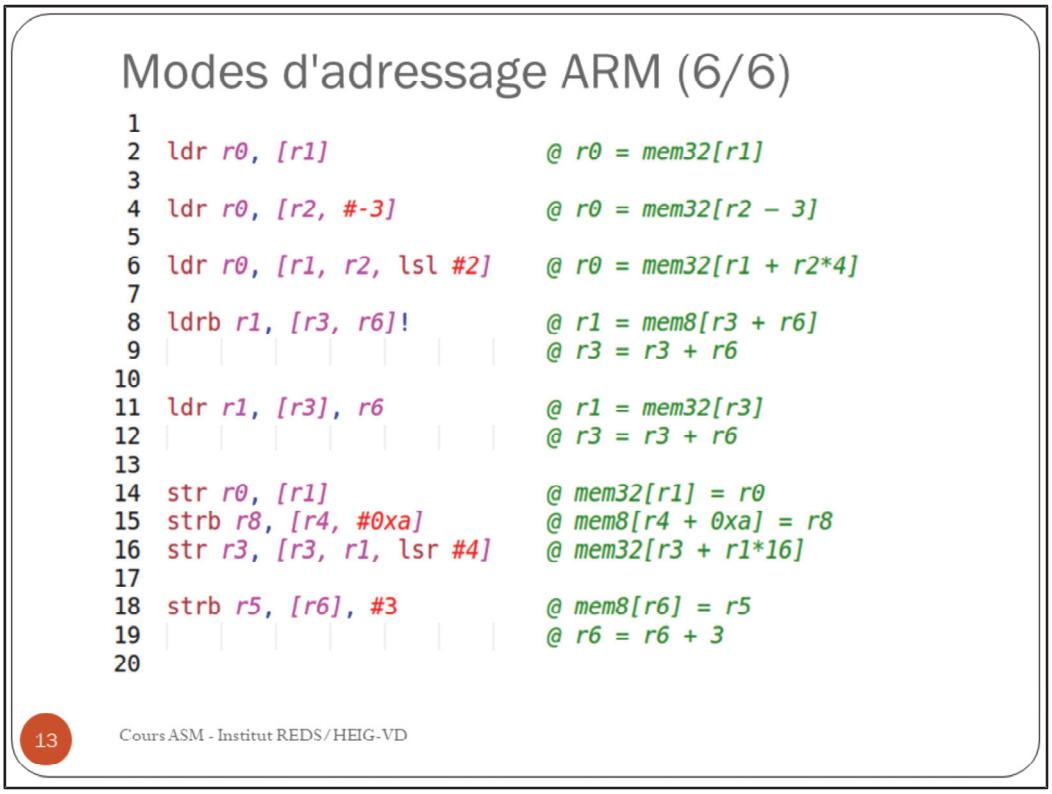


Le mode d'indexage *auto-indexé* est identique à celui *pré-indexé* avec la particularité que le registre de la seconde opérande est **mis à jour automatiquement après** l'opération de transfert.

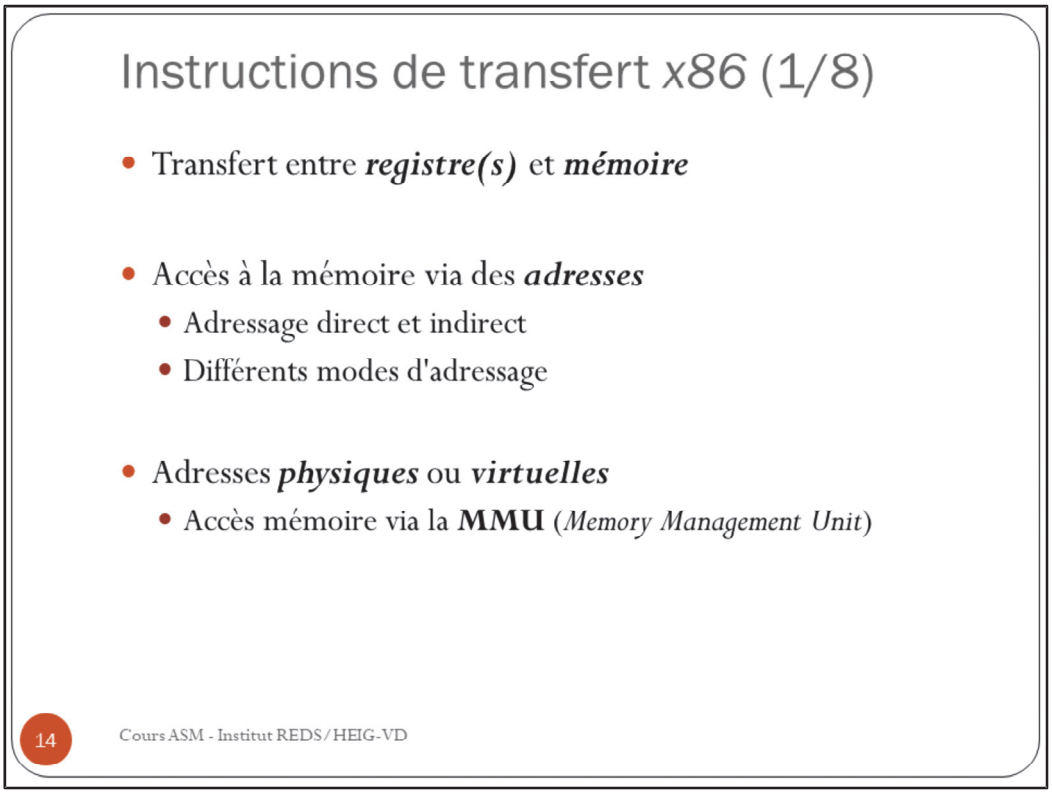


Les exemples ci-dessus montrent l'utilisation des différents modes d'adressage.

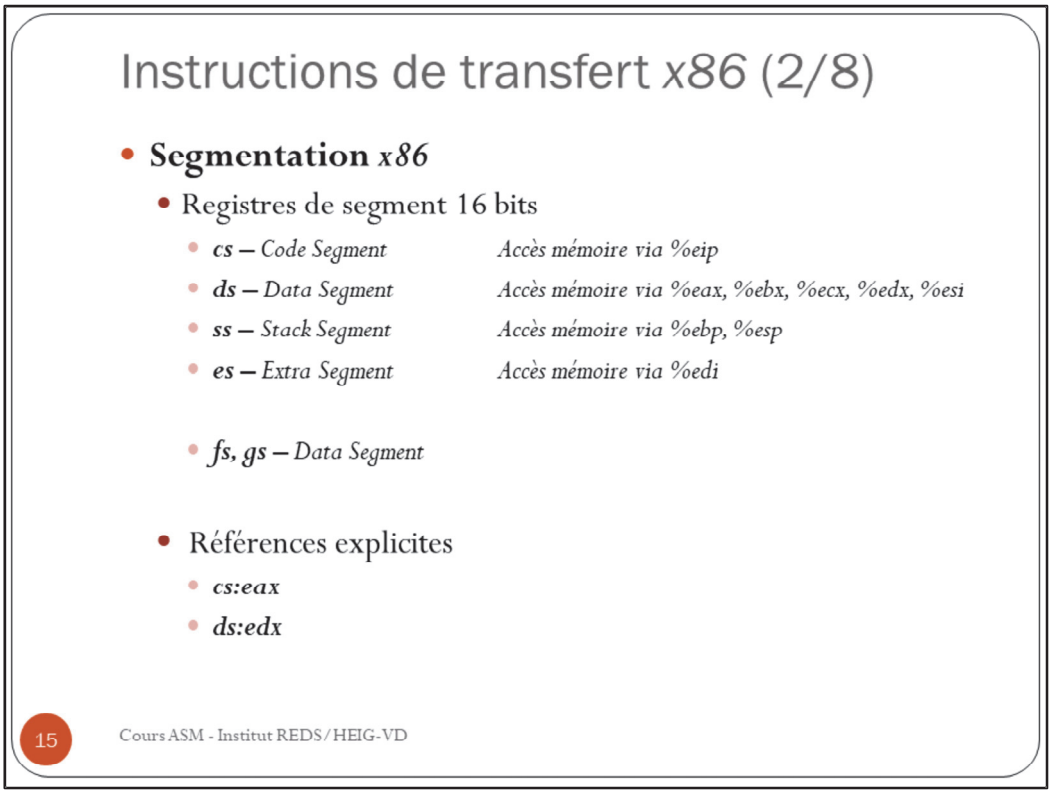
L'index peut être négatif. L'opération agissant sur la troisième opérande est toujours une addition.



D'autres exemples d'utilisation des instructions *ldr* et *str* sont montrés ci-dessus.



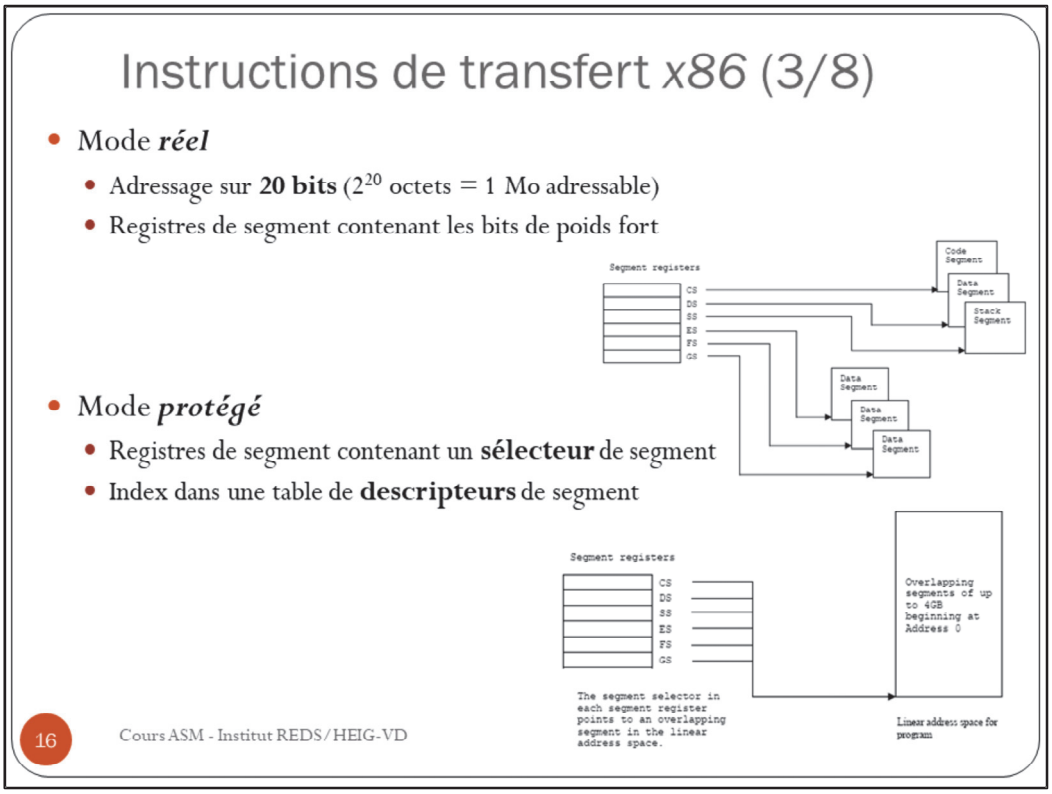
Lorsque la *MMU* est activée, l'adresse manipulée par l'instruction est obligatoirement une adresse **virtuelle**. L'instruction de transfert dans ce cas, lors de son exécution, sollicitera la *MMU* à chaque accès qui devra traduire l'adresse virtuelle en adresse physique au travers des tables de traduction (tables de segment, tables de pages).



Sur *x86*, les accès mémoire nécessitent l'utilisation de segments, même si, comme nous l'avons déjà évoqué, ceux-ci sont initialisés au démarrage d'un OS ou d'un moniteur pour contenir l'espace d'adressage complet.

Les registres de segment sont utilisés pour représenter la **base** de l'adresse; le registre *eip*, par exemple, utilisé lors de la récupération d'une instruction, constitue le *déplacement* à l'intérieur du segment dont la base est la valeur contenue dans le registre *cs.* C'est le cas en mode *réel*, mais en en mode protégé, le registre de segment contient une adresse – ou ***sélecteur de segment*** – vers une structure de données appelée ***descripteur de segment*** qui contient notamment l'adresse de base d'un segment. C'est la *MMU* qui gère les accès aux descripteurs lors d'un accès mémoire. Lors d'un accès mémoire, le processeur utilise le sélecteur de segment contenu dans l'un des registres de segment, choisi de manière implicite en fonction de l'opération. Par exemple, le *fetch* d'une instruction conduit à l'utilisation du registre *cs*, les opérations faisant intervenir la pile (registres *sp* et *bp*) conduit à l'utilisation du registre *ss*; le registre *ds* sera considéré pour toute opération sur les autres types de données, et le registre *es* sera utilisé par les instructions qui manipulent les blocs mémoire (chaînes de caractères par exemple).

Le sélecteur de segment peut être aussi explicite dans l'utilisation de l'adresse; dans ce cas, le registre de segment préfixera le registre d'adresse avec le signe *:* (exemple: ***cs:eax***, ou ***es:edi***, etc.)

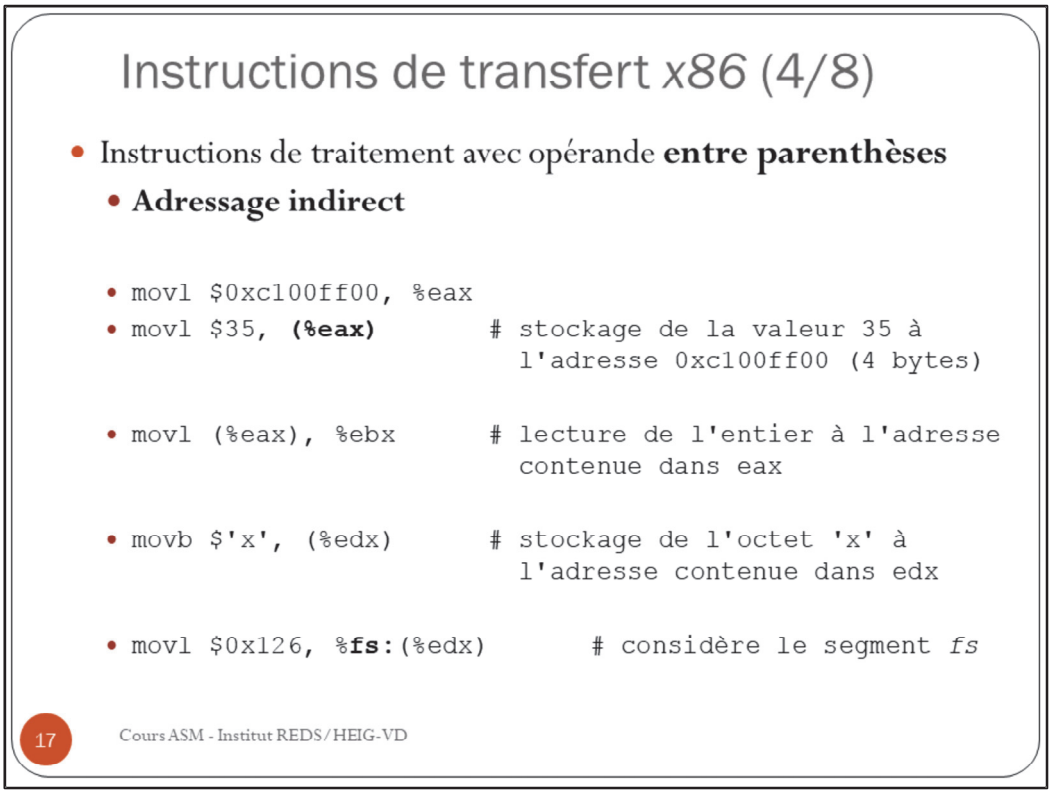


Dans le mode initial du *x86* (mode réel), le processeur ne peut adresser plus d'un Mégaoctet d'adresses, pour des raisons "historiques". Ce mode est malgré tout toujours présent et le processeur doit être configuré en mode protégé afin de pouvoir adresser tout l'espace mémoire. Il est à noter aussi que ce mode est parfois utilisé pour des systèmes temps-réels critiques où la mémoire est relativement limitée.

Le mode réel – comme les autres modes – repose sur la segmentation mémoire; les registres de segment sont utilisés, dans ce cas, pour stocker l'adresse de base du segment (en réalité, l'adresse de base est calculée à partir de la valeur du registre décalée de 4 bits à gauche).

Comme les segments peuvent avoir une taille maximale de 64 Ko, ceux-ci peuvent se chevaucher; c'est une particularité de ce mode d'exécution.

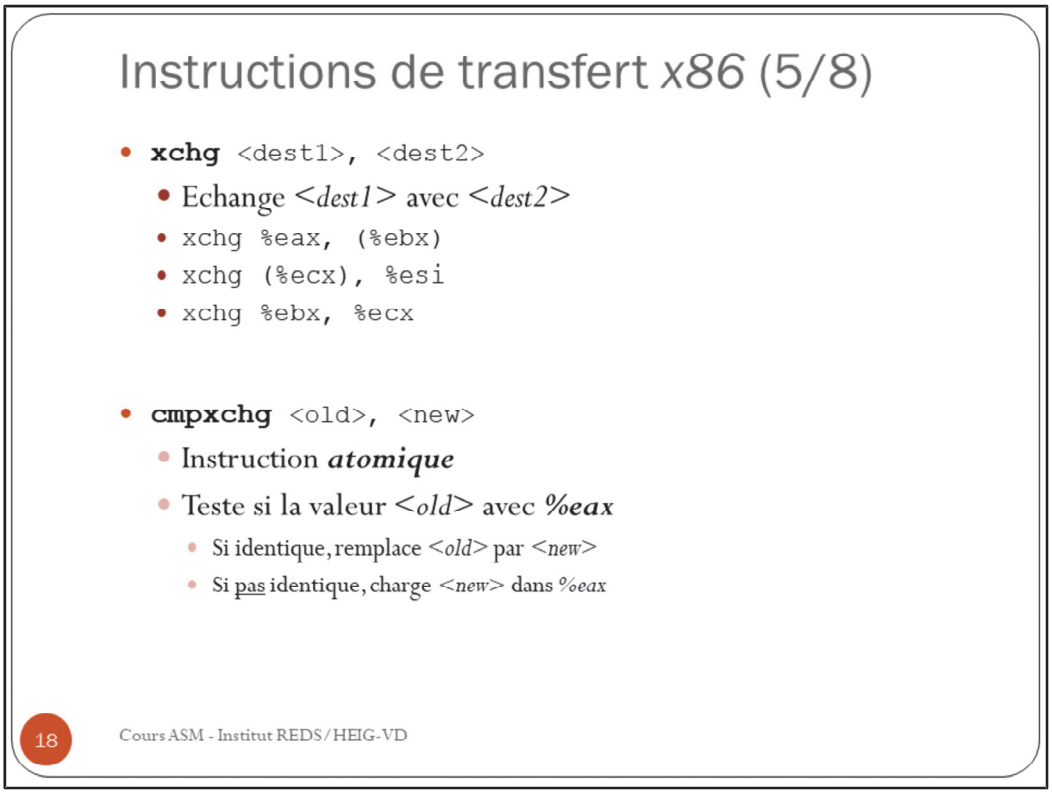
Le mode protégé utilise une table de segments – ou table de descripteurs de segment – contenant les informations relatives aux segments. Cette table est typiquement initialisée au démarrage de l'OS (ou du *bootloader*). La mémoire n'étant plus gérée par segmentation mais par pagination, les segments seront définis comme étant l'espace d'adressage complet (4 Go).



L'accès mémoire est caractérisé par une opérande entre parenthèses (). Comme illustré ci-dessus, le registre contenant l'adresse mémoire peut être l'opérande de gauche (source) ou de droite (destination).

Le nombre de *bytes* transmis dépendra du suffixe de l'instruction. Dans le cas d'un transfert mémoire, il est habituel de spécifier explicitement le suffixe, le compilateur ne pouvant pas le déterminer dans la plupart des cas.

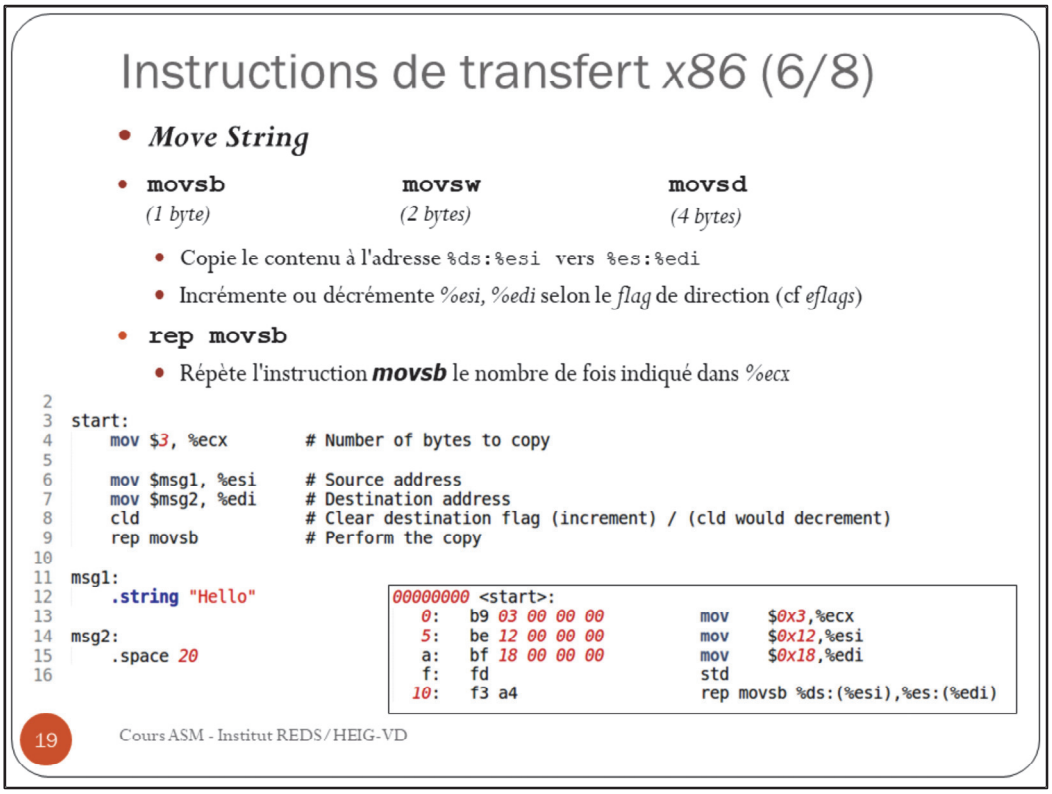
Le registre de segment peut être aussi spécifié de manière explicite; dans ce cas, il précédera le registre contenant l'adresse, mais sans être inclus dans les parenthèses.



Tout comme les autres instructions, l'instruction *xchg* utilise une opérande sous forme de registre, et l'autre sous forme d'adresse utilisée pour l'accès mémoire; il n'est pas possible d'utiliser deux opérandes contenant des adresses mémoire, les instructions de transfert nécessitant – obligatoirement – un registre du processeur pour stocker la valeur transférée.

L'instruction *cmpxchg* permet de tester la valeur de la première opérande (généralement un accès mémoire) avec le contenu du registre *%eax* et d'effectuer l'échange si et seulement si les deux valeurs sont identiques. Si elles ne sont pas identiques, la valeur de la seconde opérande est transférée dans le registre *%eax*.

Le test de la valeur et l'échange sont effectués de manière ***atomique***, c-à-d que le contenu en mémoire **ne peut pas** être modifié entre les deux opérations. Ce type d'instruction – aussi connu sous le nom *Test-and-Set* – est nécessaire afin de pouvoir implémenter efficacement des mécanismes de verrou en programmation concurrente.

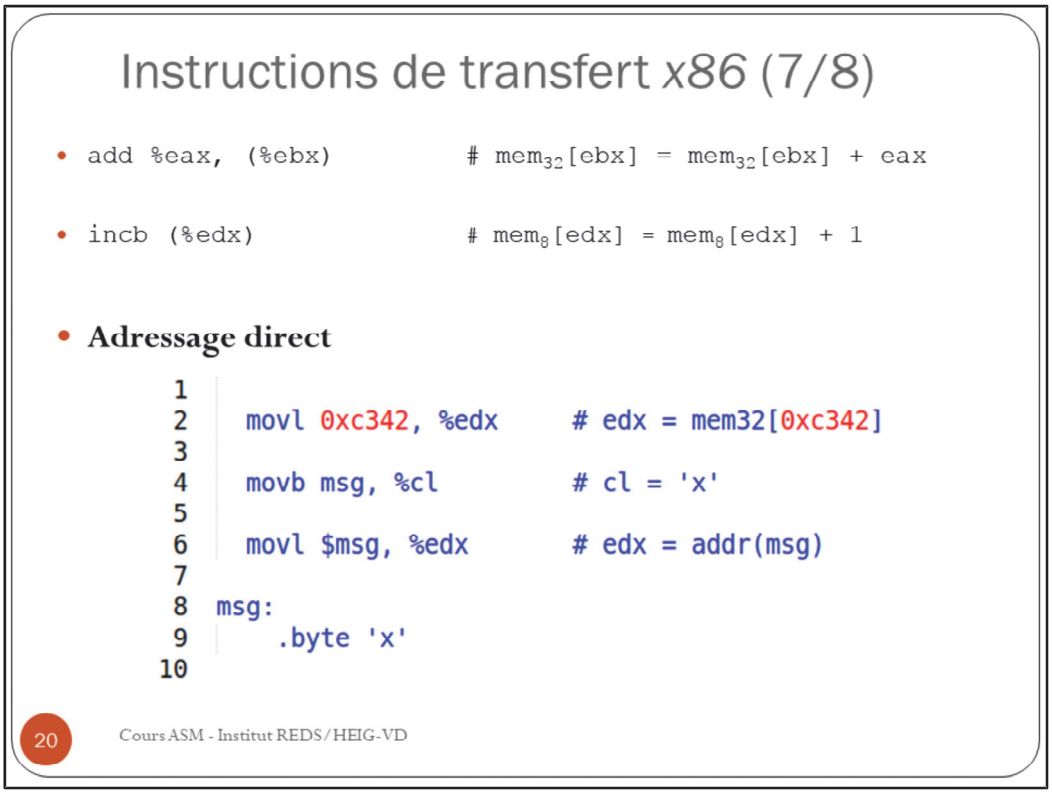


Les instructions mentionnées ci-dessus sont des exemples d'instructions complexes : elles permettent de transférer un bloc mémoire (comme une chaîne de caractères par exemple) d'un endroit vers un autre, en une seule instruction (exécutée plusieurs fois automatiquement).

L'instruction *movsb* utilise le bit **D** du registre *eflags*. Il s'agit du bit de *direction*; si celui-ci est à 0, cela signifie que les registres *esi* et *edi* seront incrémentés, si le bit D est à 1, les registres sont décrémentés. Ce bit de direction est contrôlé à l'aides des instructions ***cld*** (*clear direction flag*) et ***std*** (*set direction flag*).

C'est également un type d'instruction qui fait intervenir des segments différents : le segment *ds* contient les données d'origine alors que le segment *es* contient la destination. On se rappelle que les registres de segment peuvent se référer au même espace d'adressage.

Le mot-clé ***rep*** fonctionne avec le registre *ecx*. Il *répète* l'instruction qui suit, le nombre de fois correspondant à la valeur stockée dans *ecx*. A priori, le préfixe ne fonctionne que pour un ensemble limité d'instructions dont celles présentées ci-dessus.



La plupart des instructions de traitement peuvent effectuer un accès mémoire dans l'une de leurs opérandes. Ce mécanisme permet de simplifier grandement la manipulation de données en mémoire; contrairement à une architecture *RISC* où les instructions de traitement ne permettent pas de mélanger les types d'opérande, les instructions *CISC* de type *x86* donnent ainsi beaucoup de flexibilité au programmeur.

Dans le cadre d'un adressage direct, l'adresse est directement spécifiée dans l'une des opérandes. Cette adresse peut être explicite ou être caractérisée par l'utilisation d'un *label*. Dans les deux cas, on **n'utilisera pas** le symbole **$** comme préfixe.

Instructions de transfert *x86* (8/8)

- Soit les variables entières signées ***x, y, z***.
  - **x** dans l'adresse mémoire **0x010**
  - **y** dans l'adresse mémoire **0x014**
  - **z** dans l'adresse mémoire **0x018**

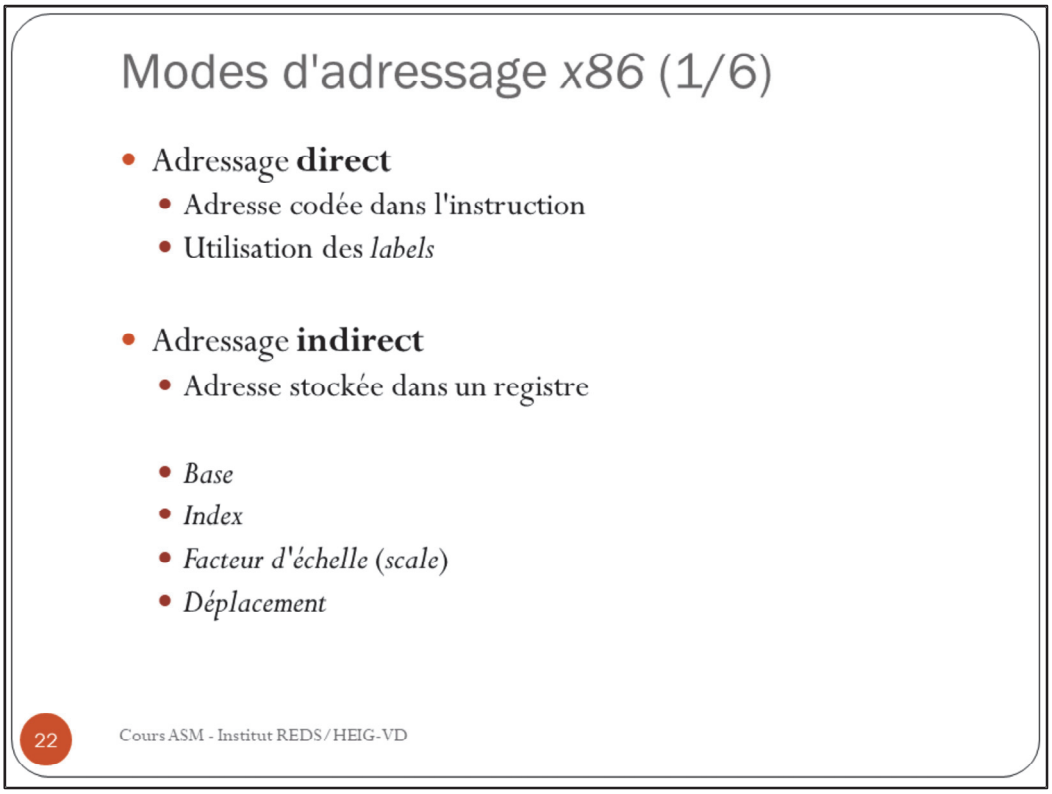
⇒ Codez en assembleur l'une ou l'autre des opérations ci-dessous.

```
# z = x + y + 4
```

```
# z = 3x – 5
```

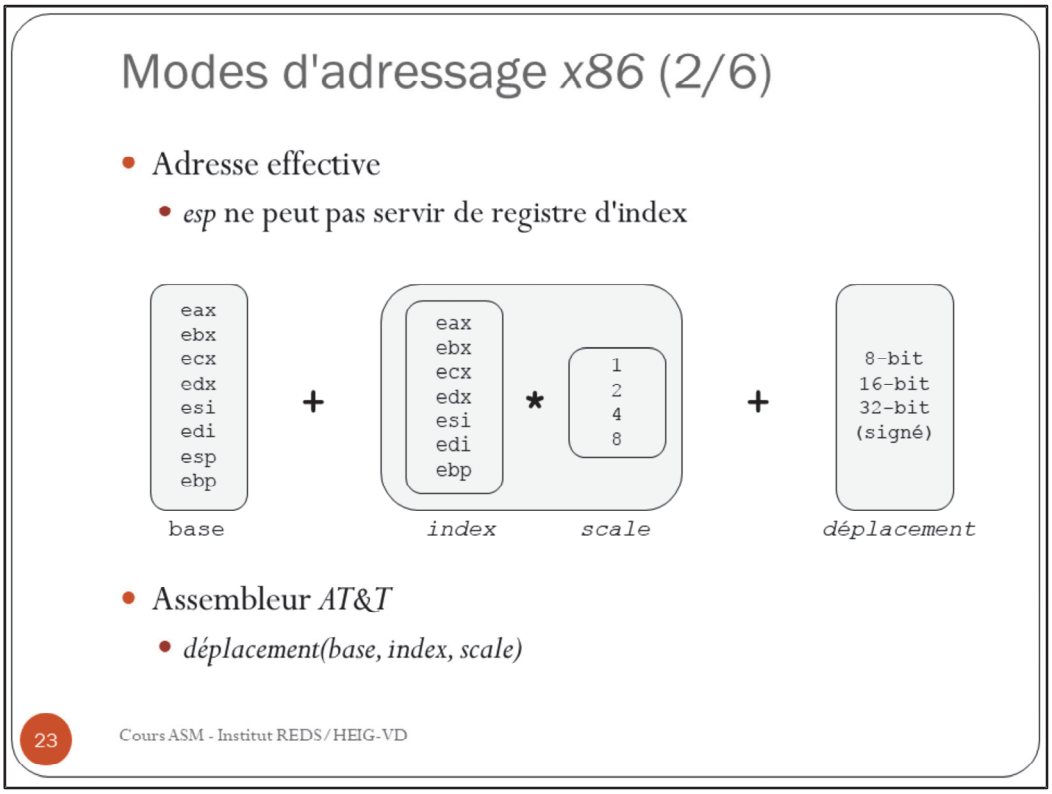
```
# z = (x + y) / 4
```

```
# z = 4x + 3
```



Les modes d'adressage sont utilisés dans le cadre d'un adressage indirect. Ils facilitent grandement les accès mémoire grâce à l'utilisation d'index/*offset* permettant au programmeur de se déplacer facilement dans des régions mémoire telles que tableaux multidimensionnels, chaînes de caractères, etc.

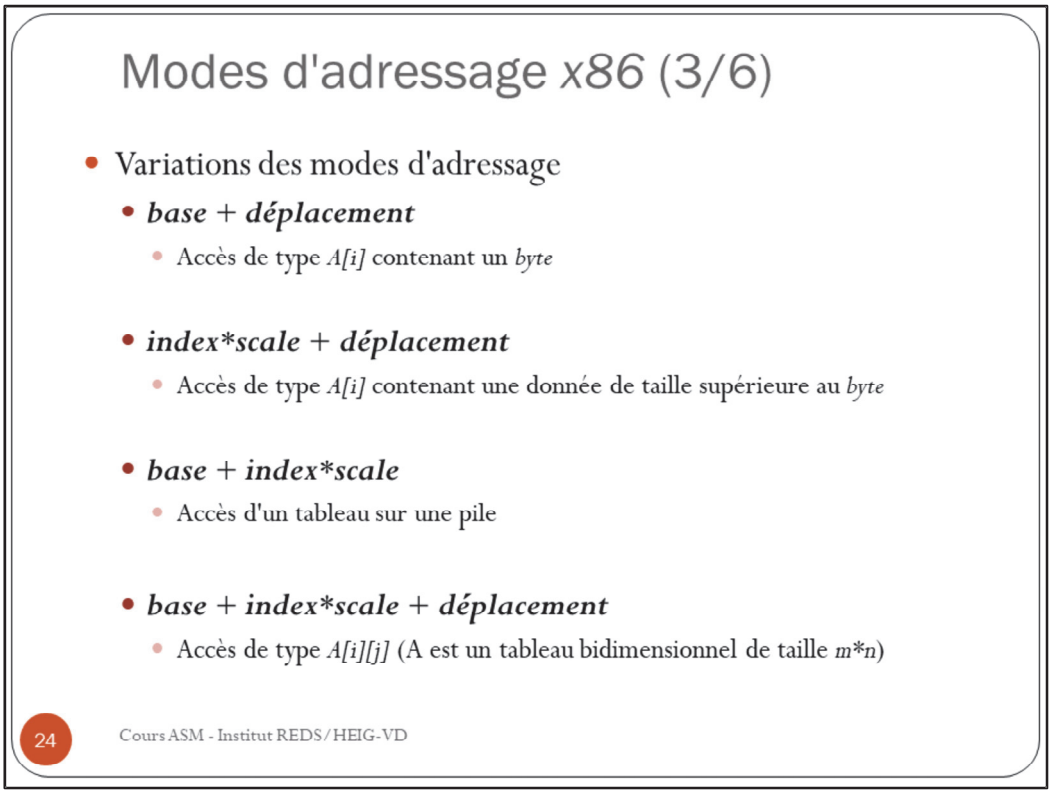
Sur *x86*, les modes d'adressage sont essentiellement basés sur les notions de *base, index, scale* et *déplacement*.



Le modèle général d'adressage sous *x86* est représenté sous la forme *déplacement(base, index, scale)* en assembleur *AT&T*.

Les conditions d'utilisation sont les suivantes :

- Le déplacement est exprimé sous forme d'une valeur immédiate (sans $) ou d'un *label*.
- La base doit être stockée dans un registre général.
- L'index doit être un registre général, **sauf *esp*.**
- Le facteur *scale* peut prendre la valeur **1, 2, 4 ou 8**.



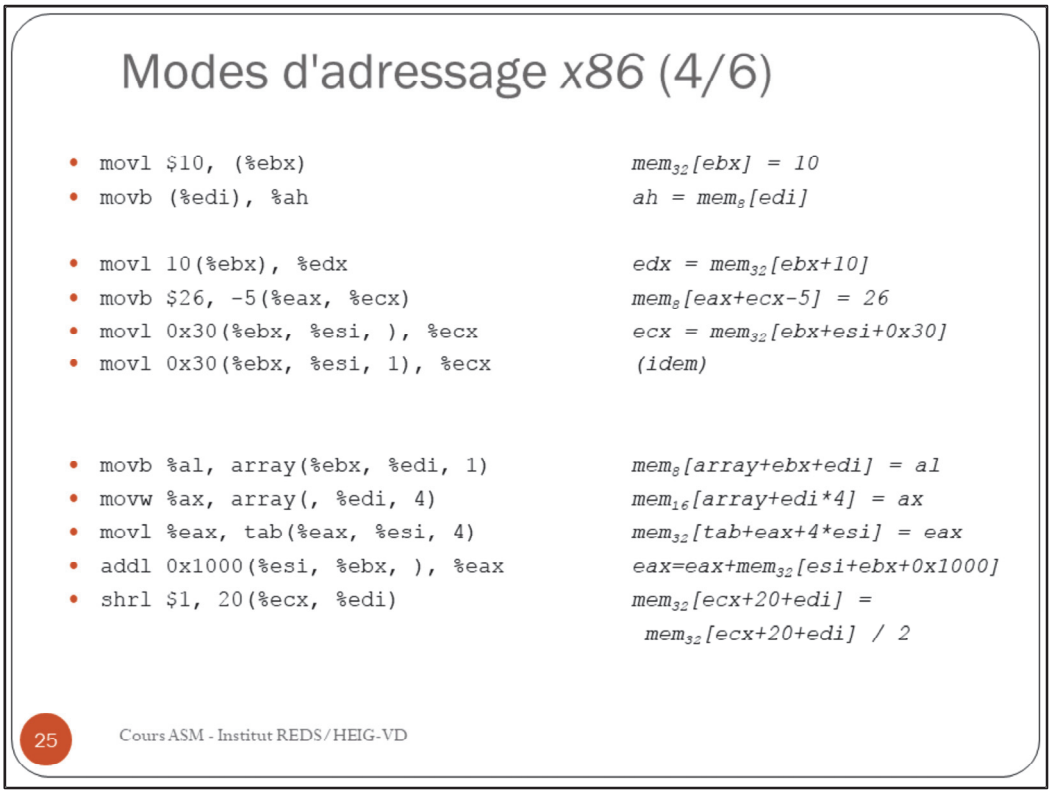
Basées sur le modèle général, différentes variantes sont possibles en fonction de la zone mémoire référencée.

La forme ***base + déplacement*** est typiquement utilisée pour se déplacer dans un tableau d'octets. Un incrément de un à la base permet de référer le *byte* suivant (attention à utiliser le bon élément du mode d'adressage). Par exemple :
- *10(%eax)*
- *-5(%ebp)*

La forme ***index*scale + déplacement*** permet de se déplacer de plusieurs *bytes* à la fois. Il pourrait s'agir alors d'un tableau d'entiers ou de demi-mots (16 bits).
- *5(, %eax, 2)*
- *array(, %esi, 4)* (*array* représente un *label* donnant la position d'une zone mémoire).

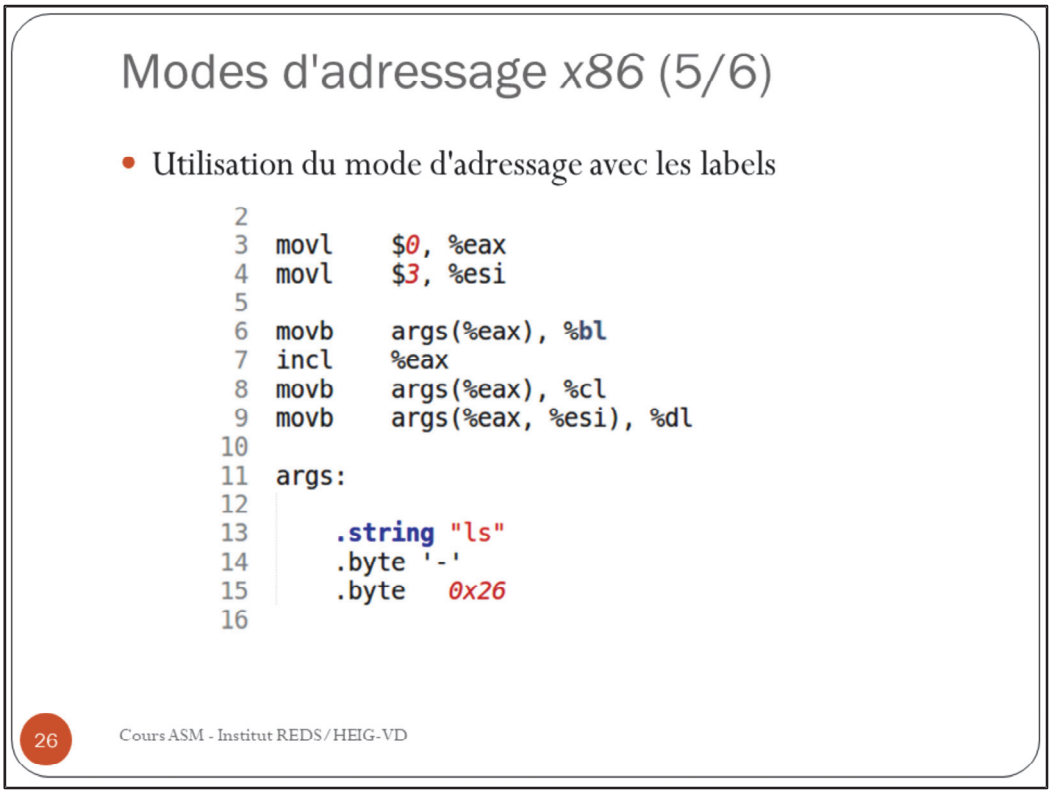
La **forme *base + index*scale*** peut être utilisée pour référer le contenu d'une pile par exemple, mais peut tout aussi bien être considérée également pour indexer un tableau.
- *(%ebx, %esi)*
- *(%ebx, %esi, 1)*
- *(%ebx, %esi,)*

Finalement, la forme générale permettrait d'accéder un tableau multidimensionnel. Si *array[i][j] est* un tableau d'éléments de taille *s*, et l'on réfère l'élément [i][j], alors le forme est la suivante:
- *array(%ebx, %esi, 4)* avec *ebx = i*n*s* et *esi = j* (n : # éléments de la 1ère dim.)



Les exemples ci-dessus montrent comment utiliser les différentes modes d'adressage. Le suffixe exprimant la taille devrait être systématiquement utilisé afin de faciliter la lecture, et d'informer le compilateur de la taille exacte.

On notera que l'absence du facteur *scale* équivaut à un facteur unitaire (de valeur 1).

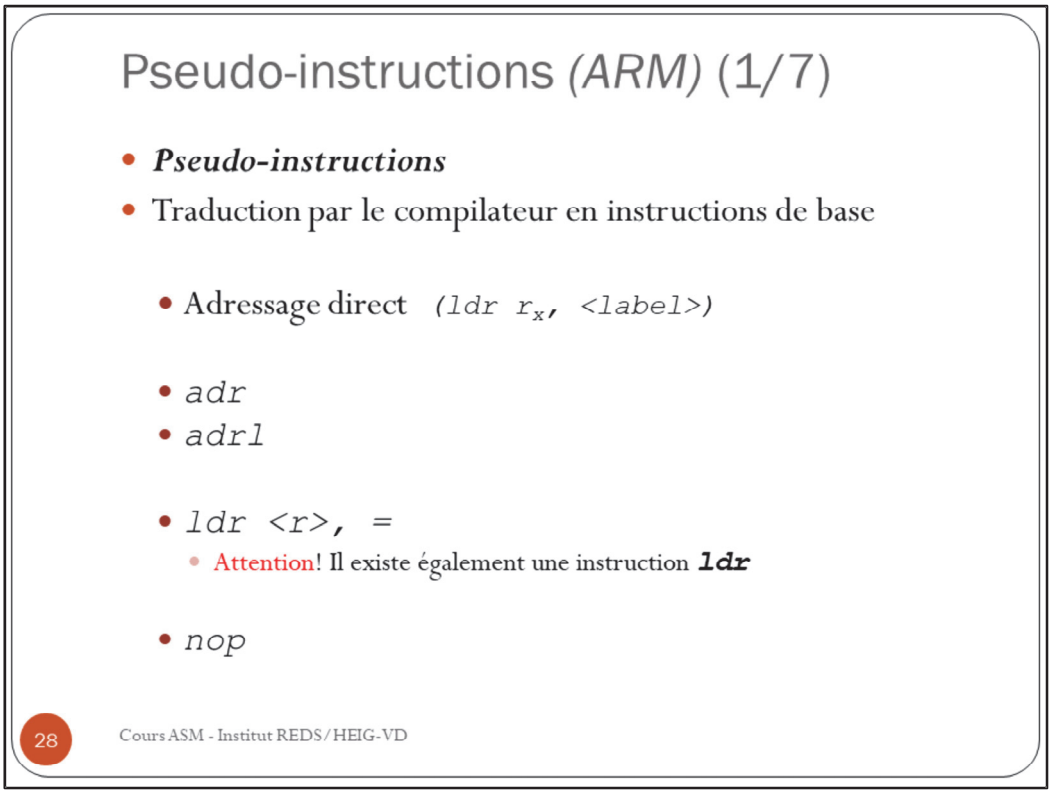


L'exemple ci-dessus montre l'utilisation du mode d'adressage avec un label (*args*). Le label prend la place du déplacement et se substitue à une valeur immédiate. Le compilateur et le *linker* se chargeront de déterminer l'adresse correspondante.

Modes d'adressage *x86* (6/6)

Réalisez un code assembleur *x86* qui corresponde au code C ci-dessous.

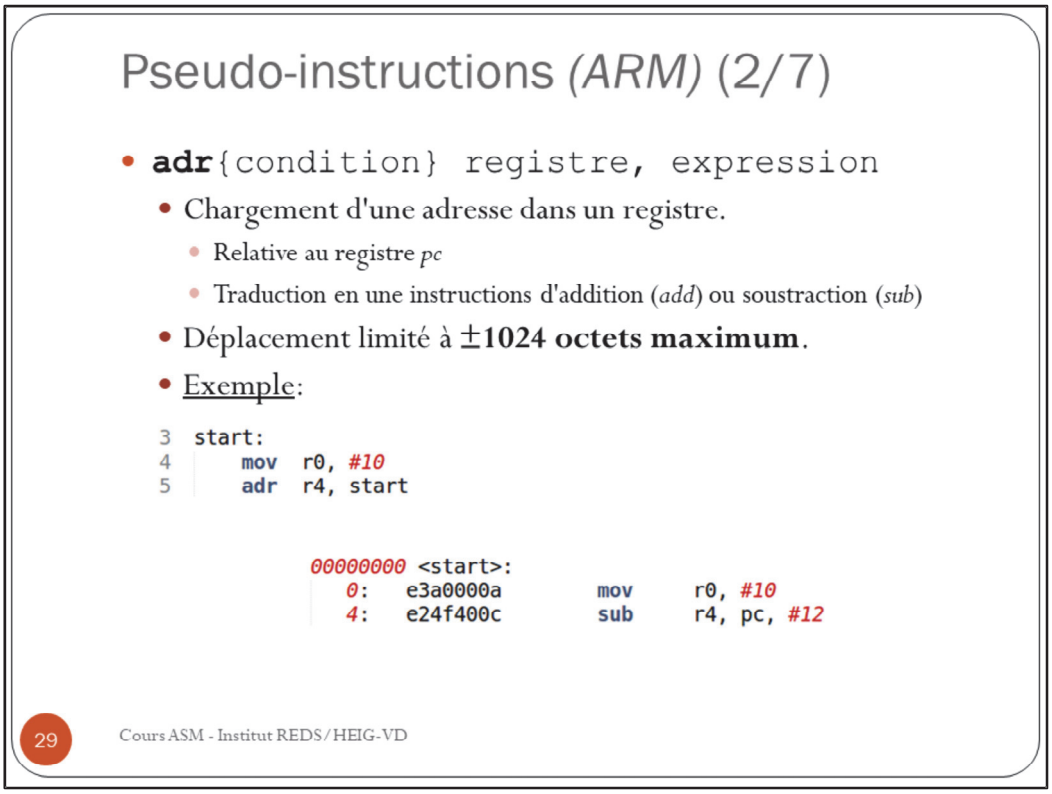
```
int tab[5][10];
unsigned char *strings[5];
unsigned char args[5][10];

tab[3][2] = 0x1324;
tab[0][3] = tab[3][0];

strings[2] = "Hello world\n";
args[3][9] = strings[2][5];
args[3][2] = 'c';
```



Une pseudo-instruction est un élément du langage assembleur; elle apparaît comme une instruction *ARM* du point de vue du programmeur, mais doit être traduite par le compilateur en instructions de base.

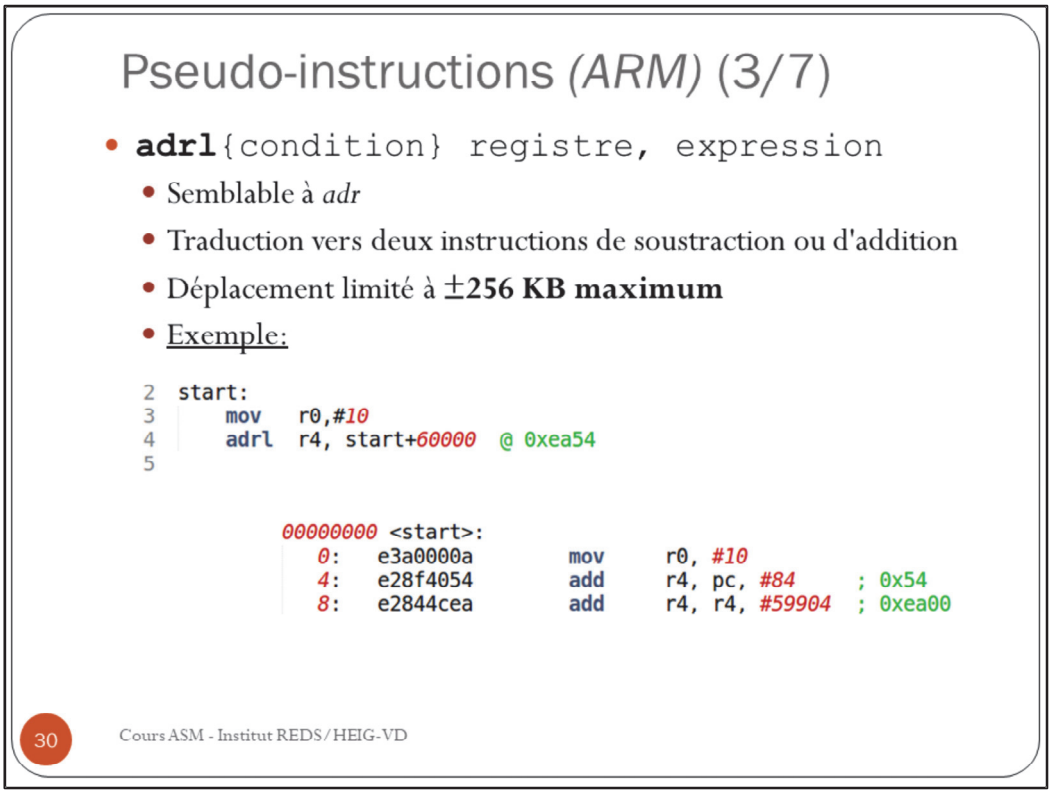
Les pseudo-instructions les plus courantes sont *adr, adrl, "ldr =",* et *nop.*

De plus, nous avons vu précédemment que l'adressage direct consiste à donner un *label* comme seconde opérande de l'instruction *ldr*. Cependant, le compilateur traduit cette forme d'instruction en une instruction de transfert *ldr* avec adressage indirect, en calculant la position du *label* par rapport au registre *pc* et en passant cette valeur comme adresse.

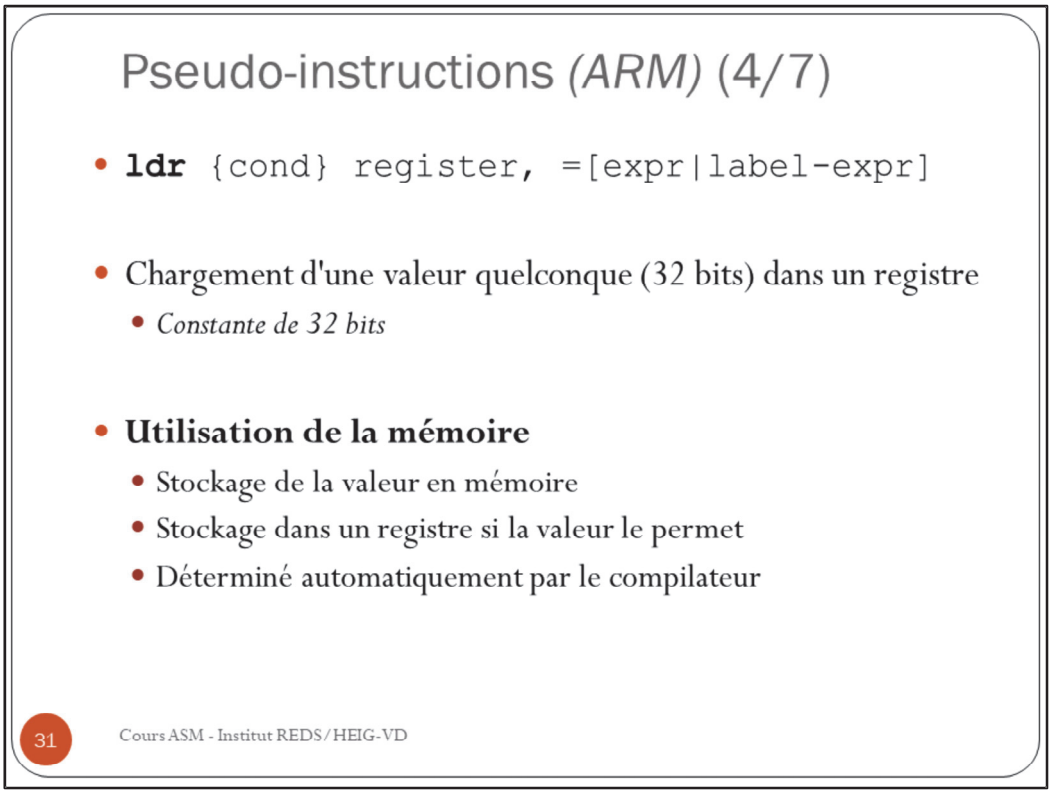


La pseudo-instruction *adr* est utile pour récupérer l'adresse d'un *label*. Elle a la particularité qu'elle est traduite par de simples additions ou soustractions en fonction de la position du *label* par rapport à la position courante (registre *pc*).

Comme l'instruction d'addition ou de soustraction ne peut contenir une valeur immédiate quelconque, le déplacement correspondant ne peut dépasser une certaine limite; la position du *label* **ne peut dépasser 1024 octets** par rapport à la position courante.



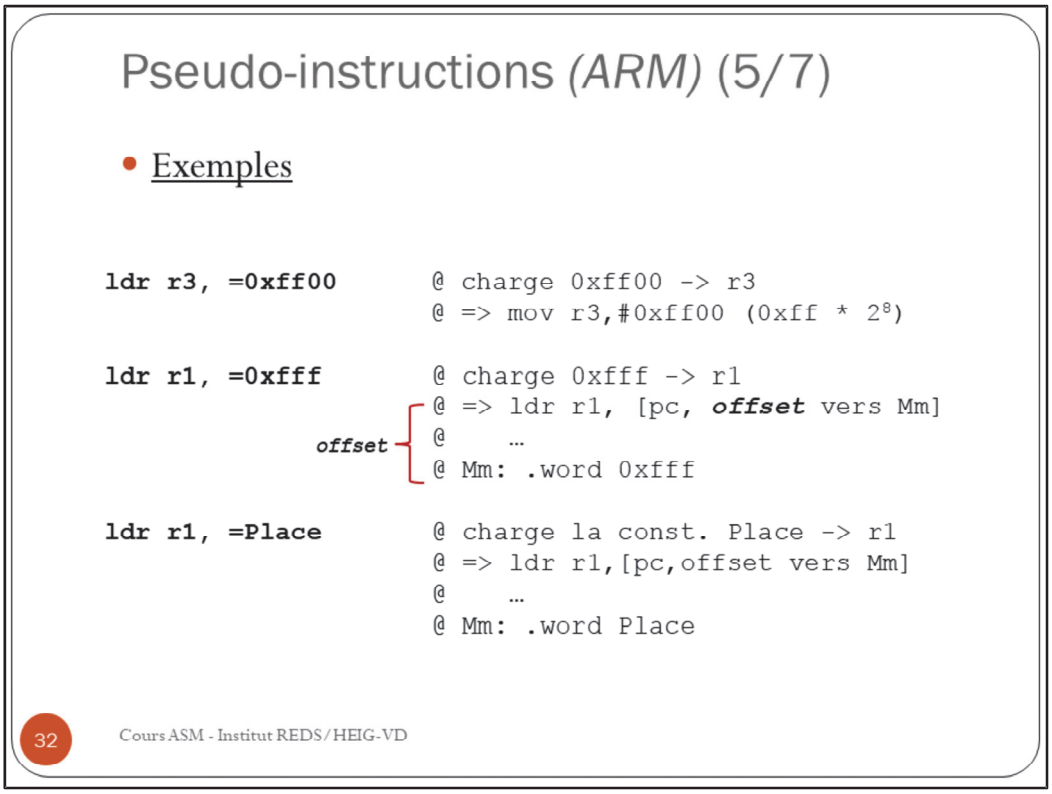
La pseudo-instruction *adrl* est identique à *adr,* sauf qu'elle accepte un déplacement plus grand; la traduction s'effectuera alors avec deux instructions d'addition ou de soustraction, nécessaires pour réaliser le déplacement souhaité.



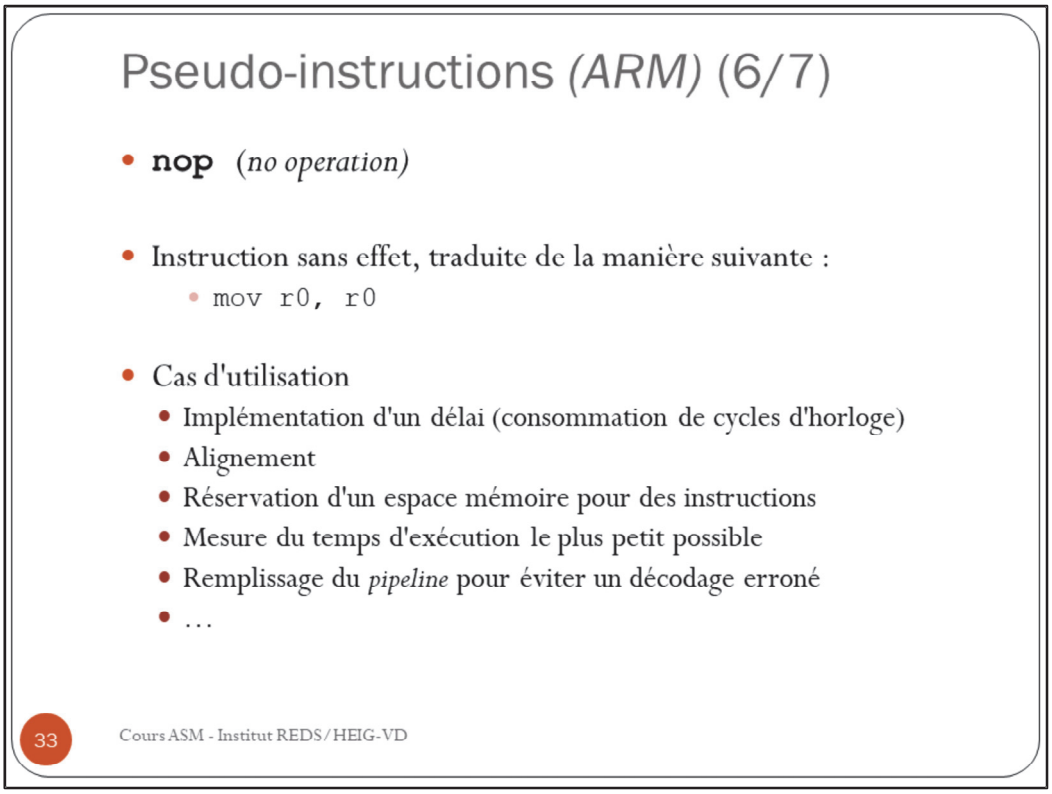
La pseudo-instruction `ldr` r_n `,=<value>` est très souvent utilisée puisqu'elle autorise le chargement de n'importe quelle valeur 32 bits dans un registre. En revanche, il faut être conscient qu'elle implique un transfert mémoire dans la plupart des cas. Le mécanisme consiste à stocker la valeur 32 bits quelque part en mémoire, lors de la compilation, puis à déterminer l'adresse de son emplacement, et à effectuer ainsi le transfert mémoire via cette adresse avec une instruction `ldr` conventionnelle.

L'emplacement mémoire déterminée lors de la compilation porte le nom de "*pool de literal*". Le compilateur place généralement cette zone mémoire proche de l'instruction de transfert, quelque part dans la section du code (*text*).

Le développeur a aussi la possibilité de forcer l'emplacement du *pool de literal* à une position déterminée (via un *label* par exemple) en utilisant la directive de compilation *.ltorg*.



La pseudo-instruction "*ldr =*" permet également le stockage de l'adresse d'un *label* situé n'importe où dans le programme, sans limitation par rapport à la position courante. L'adresse du *label* est déterminée lors de la compilation, et stockée dans un emplacement mémoire (*pool de literal*) défini par le compilateur. Ainsi, tout comme une valeur quelconque, l'adresse n'est pas codée dans l'instruction, mais récupérée via un transfert mémoire normal.



Sur *ARM,* il n'existe pas d'instruction *nop* explicite. En effet, la pseudo-instruction *nop* est traduite en une instruction de traitement n'ayant aucun effet. Comme l'instruction de traitement ne nécessite qu'un cycle d'horloge, l'effet sera équivalent à une "vraie" instruction *nop*.

Bien qu'il peut paraître superflu d'avoir une telle instruction, elle est toutefois utilisée dans de nombreux contextes. Dans les systèmes embarqués, il peut s'avérer utile d'introduire de petites temporisations afin de permettre à certains périphériques (contrôleurs mémoire par exemple) de s'initialiser correctement. On peut également effectuer une série de *nop* afin de forcer un alignement si l'application l'exige. L'instruction *nop* peut être aussi utilisée afin de mesure le temps d'exécution le plus petit.

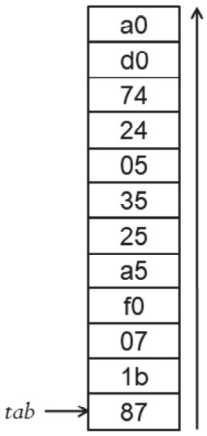
On évoquera finalement l'utilisation de cette pseudo-instruction pour remplir le *pipeline* et éviter ainsi des problèmes potentiels de décodage d'instruction; un tel cas peut se produire si une instruction de saut est immédiatement suivie par une zone de données. L'organisation en *pipeline* conduit au décodage de l'instruction suivante durant le même cycle d'exécution de l'instruction courante. Si le décodeur d'instruction ne reconnaît pas une instruction valide, le processeur lèvera une exception de type *undefined.*

Pseudo-instructions *(ARM)* (7/7)

⇒ Indiquez les valeurs de r1 et r2.
(le processeur est configuré en *little-endian*.)

R1 **R2**

```
3   adr     r1, tab
4   ldrb    r2, [r1]
5   ldr     r2, [r1]
6   ldr     r2, [r1], #4
7
8   ldrb    r2, [r1]
9   ldr     r2, [r1, #4]!
10
11  ldr     r2, [r1, #-2]
12
13  @ 0xc0009000
14  tab:
15      .word  0xf0071b87
16
```



Références

- ARM Infocenter. Site ARM principal : *http://infocenter.arm.com*
- Intel x86 32 & 64 bits.
  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

- Rajat Moona, *Assembly Language Programming in GNU/Linux for IA32 Architectures*, Eastern Economy Edition, New Dehli, 2009.
- Andrew N.Sloss, Dominic Symes, *ARM System Developer's Guide, Designing and Optimizing System Software*, Elsevier, 2004.