

# Programmation assembleur (ASM)

## *Instructions de transfert*

Prof. Daniel Rossier  
Version 1.7 (2017-2018)

## Plan

- Instructions de **transfert de données** (*x86 & ARM*)
- Modes d'adressage (*x86 & ARM*)
- Pseudo-instructions (*ARM*)

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.

## Instructions de transfert ARM (1/5)

- Transfert entre *registre(s)* et *mémoire*
- L'exécution peut s'opérer en plusieurs cycles machine.
  - Transfert *simple* registre-mémoire
    - Exécution en 2 cycles
  - Transfert *multiple* registres-mémoire
    - Exécution en  $n+1$  cycles

L'instruction de transfert possède une ou plusieurs opérands, 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.

## Instructions de transfert *ARM* (2/5)

- Deux opérandes
  - Destination, source
- Différents mnémoniques en **fonction de la taille** de la donnée:
  - Mot (*Word*)                    **32 bits**                    (sans suffixe)
  - Demi-mot (*Half-word*)       **16 bits**                    (suffixe **h**)
  - *Byte*                                **8 bits**                      (suffixe **b**)
- L'utilisation de **[R<sub>x</sub>]** permet l'accès d'une donnée stockée à l'adresse contenu dans **R<sub>x</sub>**
  - Equivalent du symbole \* en C
  - **Adressage indirect**

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.

## Instructions de transfert ARM (3/5)

- Transfert de la mémoire vers un registre (*load*)
  - `ldr rx, [ry]` Copie du *word* stocké en mémoire à l'adresse  $r_y$  dans le registre  $r_x$
  - `ldrb rx, [ry]` Copie d'un *byte* stocké en mémoire à l'adresse  $r_y$  dans le registre  $r_x$
- Transfert d'un registre vers la mémoire (*store*)
  - `str rx, [ry]` Copie d'un *word* stocké dans le registre  $r_x$  vers la mémoire à l'adresse  $r_y$
  - `strb rx, [ry]` Copie d'un *byte* stocké dans le registre  $r_x$  vers la mémoire à l'adresse  $r_y$

5

Cours ASM - Institut REDS/HEIG-VD

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 opérands 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.

## Instructions de transfert ARM (4/5)

- Un autre mnémotique permet d'utiliser un *label* (étiquette) à la place du registre d'adresse.
  - (Pseudo-)adressage direct

```
1
2 ldr r0, pos_mem    @ r0 = 0xc4534123
3 ldrb r4, val       @ r4 = 'g'
4
5
6 pos_mem:
7 | .word 0xc4534123
8
9 val:
10 | .byte 'g'
```

6

Cours ASM - Institut REDS/HEIG-VD

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.

$$@ z = x + y + 4$$

$$@ z = 3x - 5$$

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

$$@ z = 4x + 3$$

## Modes d'adressage ARM (1/6)

- **Différents modes d'adressage**
  - Sans indexage
  - **Pré-indexé**
  - **Post-indexé**
  - **Auto-indexé**

8

Cours ASM - Institut REDS/HEIG-VD

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 décrétementée), 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.



## Modes d'adressage ARM (2/6)

- **Pré-indexé** (*pré-indexage*)
  - $[r_n, \#imm]$ 
    - $\#imm$  est rajoutée à l'adresse **avant** l'accès mémoire
      - $r_n + \#imm$
    - L'adresse reste **inchangée**.
  - $[r_i, r_j]$ 
    - Accès à un élément d'un tableau de type (base + *index*)
    - Possibilité de décalage sur  $r_j$  :  $[r_i, r_j, lsl \#3]$

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.

## Modes d'adressage ARM (3/6)

- **Post-indexé** (*post-indexage*)

- $[r_n], \#imm$ 
  - L'accès mémoire est effectué à l'adresse  $r_n$
  - Puis, l'adresse est modifiée:  $r_n = r_n + \#imm$
- $[r_i], r_j$ 
  - Accès à un élément d'un tableau de type (base + *index*)
  - Possibilité de décalage sur  $r_j$  :  $[r_i], r_j, lsr \#2$
  - L'adresse est modifiée:  $r_i = r_i + r_j$

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.

## Modes d'adressage ARM (4/6)

- **Auto-indexé (auto-indexage)**

- $[r_n, \#imm]!$ 
  - $\#imm$  est rajoutée à l'adresse **avant** l'accès mémoire.
    - $r_n + \#imm$
  - L'adresse est mis à jour :  $r_n = r_n + \#imm$
- $[r_i, r_j]!$ 
  - Accès à un élément d'un tableau de type (base + index)
  - Possibilité de décalage sur  $r_j$  :  $[r_i, r_j, lsr \#2]!$
  - L'adresse est modifiée:  $r_i = r_i + r_j$

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.

## Modes d'adressage ARM (5/6)

- *Sans indexage*
  - `ldr r0, [r1]`            @ r0 = mem32[r1]
- *Pré-indexé*
  - `ldr r0, [r1, #-4]`       @ r0 = mem32[r1-4]
- *Auto-indexé*
  - `ldr r0, [r1, #4]!`       @ r0 = mem32[r1+4]  
                                 @ r1 = r1+4
- *Post-indexé*
  - `ldr r0, [r1], #4`       @ r0 = mem32[r1]  
                                 @ r1 = r1+4

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.

## Modes d'adressage ARM (6/6)

```
1
2  ldr r0, [r1]           @ r0 = mem32[r1]
3
4  ldr r0, [r2, #-3]     @ r0 = mem32[r2 - 3]
5
6  ldr r0, [r1, r2, lsl #2] @ r0 = mem32[r1 + r2*4]
7
8  ldrb r1, [r3, r6]!    @ r1 = mem8[r3 + r6]
9  | | | | | | | |      @ r3 = r3 + r6
10
11 ldr r1, [r3], r6      @ r1 = mem32[r3]
12 | | | | | | | |      @ r3 = r3 + r6
13
14 str r0, [r1]           @ mem32[r1] = r0
15 strb r8, [r4, #0xa]   @ mem8[r4 + 0xa] = r8
16 str r3, [r3, r1, lsr #4] @ mem32[r3 + r1*16]
17
18 strb r5, [r6], #3     @ mem8[r6] = r5
19 | | | | | | | |      @ r6 = r6 + 3
20
```

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

## Instructions de transfert x86 (1/8)

- Transfert entre *registre(s)* et *mémoire*
- Accès à la mémoire via des *adresses*
  - Adressage direct et indirect
  - Différents modes d'adressage
- Adresses *physiques* ou *virtuelles*
  - Accès mémoire via la **MMU** (*Memory Management Unit*)

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).

## Instructions de transfert x86 (2/8)

- **Segmentation x86**

- Registres de segment 16 bits

- *cs* – Code Segment      Accès mémoire via *%eip*
- *ds* – Data Segment      Accès mémoire via *%eax, %ebx, %ecx, %edx, %esi*
- *ss* – Stack Segment      Accès mémoire via *%ebp, %esp*
- *es* – Extra Segment      Accès mémoire via *%edi*

- *fs, gs* – Data Segment

- Références explicites

- *cs:eax*
- *ds:edx*

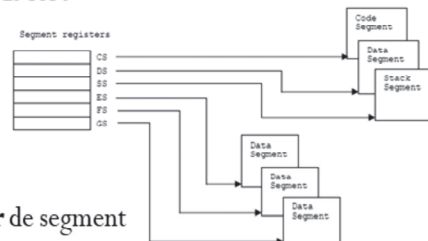
Sur *x86*, les accès mémoire nécessitent l'utilisation de segments, même si, comme nous l'avons déjà évoqué, ceux-ci sont initialisés au démarrage d'un OS ou d'un moniteur pour contenir l'espace d'adressage complet.

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).

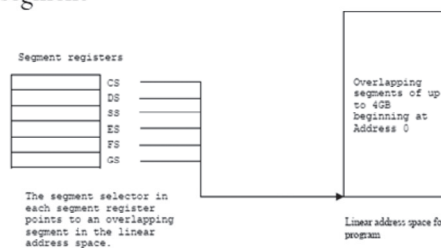
Le sélecteur de segment peut être aussi explicite dans l'utilisation de l'adresse; dans ce cas, le registre de segment préfixera le registre d'adresse avec le signe **:** (exemple: ***cs:eax***, ou ***es:edi***, etc.)

## Instructions de transfert x86 (3/8)

- Mode *réel*
  - Adressage sur **20 bits** ( $2^{20}$  octets = 1 Mo adressable)
  - Registres de segment contenant les bits de poids fort



- Mode *protégé*
  - Registres de segment contenant un **sélecteur** de segment
  - Index dans une table de **descripteurs** de segment



16

Cours ASM - Institut REDS/HEIG-VD

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).



## Instructions de transfert x86 (4/8)

- Instructions de traitement avec opérande **entre parenthèses**

- **Adressage indirect**

- `movl $0xc100ff00, %eax`
- `movl $35, (%eax)` # stockage de la valeur 35 à l'adresse 0xc100ff00 (4 bytes)
- `movl (%eax), %ebx` # lecture de l'entier à l'adresse contenue dans `eax`
- `movb '$x', (%edx)` # stockage de l'octet 'x' à l'adresse contenue dans `edx`
- `movl $0x126, %fs:(%edx)` # considère le segment `fs`

17

Cours ASM - Institut REDS/HEIG-VD

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.

## Instructions de transfert x86 (5/8)

- **xchg** <dest1>, <dest2>
  - Echange <dest1> avec <dest2>
  - xchg %eax, (%ebx)
  - xchg (%ecx), %esi
  - xchg %ebx, %ecx
  
- **cmpxchg** <old>, <new>
  - Instruction **atomique**
  - Teste si la valeur <old> avec %eax
    - Si identique, remplace <old> par <new>
    - Si pas identique, charge <new> dans %eax

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.

## Instructions de transfert x86 (6/8)

- *Move String*

- **movsb**  
(1 byte)

- **movsw**  
(2 bytes)

- **movsd**  
(4 bytes)

- Copie le contenu à l'adresse %ds:%esi vers %es:%edi
- Incrémente ou décrémente %esi, %edi selon le *flag* de direction (cf *eflags*)

- **rep movsb**

- Répète l'instruction **movsb** le nombre de fois indiqué dans %ecx

```
2
3 start:
4   mov $3, %ecx      # Number of bytes to copy
5
6   mov $msg1, %esi   # Source address
7   mov $msg2, %edi   # Destination address
8   cld              # Clear destination flag (increment) / (cld would decrement)
9   rep movsb        # Perform the copy
10
11 msg1:
12   .string "Hello"
13
14 msg2:
15   .space 20
16
```

00000000	<start>:		
0:	b9 03 00 00 00	mov	\$0x3,%ecx
5:	be 12 00 00 00	mov	\$0x12,%esi
a:	bf 18 00 00 00	mov	\$0x18,%edi
f:	fd	std	
10:	f3 a4	rep movsb	%ds:(%esi),%es:(%edi)

19

Cours ASM - Institut REDS/HEIG-VD

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'aide 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.

## Instructions de transfert x86 (7/8)

- `add %eax, (%ebx)`                   #  $\text{mem}_{32}[\text{ebx}] = \text{mem}_{32}[\text{ebx}] + \text{eax}$
- `incb (%edx)`                       #  $\text{mem}_8[\text{edx}] = \text{mem}_8[\text{edx}] + 1$

- **Adressage direct**

```
1 |
2 | movl 0xc342, %edx       # edx = mem32[0xc342]
3 |
4 | movb msg, %cl           # cl = 'x'
5 |
6 | movl $msg, %edx       # edx = addr(msg)
7 |
8 | msg:
9 |     .byte 'x'
10|
```

20

Cours ASM - Institut REDS/HEIG-VD

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.

$$\# z = x + y + 4$$

$$\# z = 3x - 5$$

$$\# z = (x + y) / 4$$

$$\# z = 4x + 3$$

## Modes d'adressage x86 (1/6)

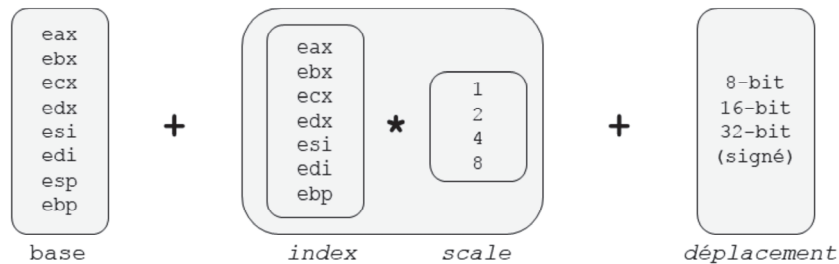
- Adressage **direct**
  - Adresse codée dans l'instruction
  - Utilisation des *labels*
- Adressage **indirect**
  - Adresse stockée dans un registre
    - *Base*
    - *Index*
    - *Facteur d'échelle (scale)*
    - *Déplacement*

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*.

## Modes d'adressage x86 (2/6)

- Adresse effective
  - *esp* ne peut pas servir de registre d'index



- Assembleur *AT&T*
  - *déplacement(base, index, scale)*

23

Cours ASM - Institut REDS/HEIG-VD

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**.

## Modes d'adressage x86 (3/6)

- Variations des modes d'adressage
  - **base + déplacement**
    - Accès de type  $A[i]$  contenant un *byte*
  - **index\*scale + déplacement**
    - Accès de type  $A[i]$  contenant une donnée de taille supérieure au *byte*
  - **base + index\*scale**
    - Accès d'un tableau sur une pile
  - **base + index\*scale + déplacement**
    - Accès de type  $A[i][j]$  ( $A$  est un tableau bidimensionnel de taille  $m*n$ )

24

Cours ASM - Institut REDS/HEIG-VD

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 1<sup>ère</sup> dim.)



## Modes d'adressage x86 (4/6)

- `movl $10, (%ebx)`  $mem_{32}[ebx] = 10$
- `movb (%edi), %ah`  $ah = mem_8[edi]$
  
- `movl 10(%ebx), %edx`  $edx = mem_{32}[ebx+10]$
- `movb $26, -5(%eax, %ecx)`  $mem_8[eax+ecx-5] = 26$
- `movl 0x30(%ebx, %esi, ), %ecx`  $ecx = mem_{32}[ebx+esi+0x30]$
- `movl 0x30(%ebx, %esi, 1), %ecx`  $(idem)$
  
- `movb %al, array(%ebx, %edi, 1)`  $mem_8[array+ebx+edi] = al$
- `movw %ax, array(, %edi, 4)`  $mem_{16}[array+edi*4] = ax$
- `movl %eax, tab(%eax, %esi, 4)`  $mem_{32}[tab+eax+4*esi] = eax$
- `addl 0x1000(%esi, %ebx, ), %eax`  $eax = eax + mem_{32}[esi+ebx+0x1000]$
- `shrl $1, 20(%ecx, %edi)`  $mem_{32}[ecx+20+edi] = mem_{32}[ecx+20+edi] / 2$

Les exemples ci-dessus montrent comment utiliser les différents 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).

## Modes d'adressage x86 (5/6)

- Utilisation du mode d'adressage avec les labels

```
2
3 movl    $0, %eax
4 movl    $3, %esi
5
6 movb    args(%eax), %bl
7 incl    %eax
8 movb    args(%eax), %cl
9 movb    args(%eax, %esi), %dl
10
11 args:
12
13     .string "ls"
14     .byte  '-'
15     .byte  0x26
16
```

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';
```

## Pseudo-instructions (ARM) (1/7)

- **Pseudo-instructions**
- Traduction par le compilateur en instructions de base
  - Adressage direct (`ldr rx, <label>`)
  - `adr`
  - `adrl`
  - `ldr <r>, =`
    - **Attention!** Il existe également une instruction **`ldr`**
  - `nop`

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.

## Pseudo-instructions (ARM) (2/7)

- **adr**{condition} registre, expression
  - Chargement d'une adresse dans un registre.
    - Relative au registre *pc*
    - Traduction en une instructions d'addition (*add*) ou soustraction (*sub*)
  - Déplacement limité à  $\pm 1024$  octets **maximum**.
  - Exemple:

```
3 start:  
4   mov r0, #10  
5   adr r4, start
```

```
00000000 <start>:  
0: e3a0000a   mov    r0, #10  
4: e24f400c   sub    r4, pc, #12
```

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.

## Pseudo-instructions (ARM) (3/7)

- **adr1**{condition} registre, expression
  - Semblable à *adr*
  - Traduction vers deux instructions de soustraction ou d'addition
  - Déplacement limité à **±256 KB maximum**
  - Exemple:

```
2 start:
3   mov   r0,#10
4   adr1  r4, start+60000 @ 0xea54
5
```

```
00000000 <start>:
0:  e3a0000a      mov   r0, #10
4:  e28f4054      add  r4, pc, #84 ; 0x54
8:  e2844cea      add  r4, r4, #59904 ; 0xea00
```

La pseudo-instruction *adr1* 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.

## Pseudo-instructions (ARM) (4/7)

- **ldr** {cond} register, =[expr|label-expr]
- Chargement d'une valeur quelconque (32 bits) dans un registre
  - Constante de 32 bits
- **Utilisation de la mémoire**
  - Stockage de la valeur en mémoire
  - Stockage dans un registre si la valeur le permet
  - Déterminé automatiquement par le compilateur

31

Cours ASM - Institut REDS/HEIG-VD

La pseudo-instruction `ldr rn, =<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`.

## Pseudo-instructions (ARM) (5/7)

- Exemples

```
ldr r3, =0xff00    @ charge 0xff00 -> r3
                   @ => mov r3,#0xff00 (0xff * 28)

ldr r1, =0xfff     @ charge 0xfff -> r1
                   @ => ldr r1, [pc, offset vers Mm]
offset {          @
           ...
           @ Mm: .word 0xfff

ldr r1, =Place     @ charge la const. Place -> r1
                   @ => ldr r1, [pc, offset vers Mm]
                   @
                   @ Mm: .word Place
```

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.



## Pseudo-instructions (ARM) (6/7)

- **nop** (*no operation*)
- Instruction sans effet, traduite de la manière suivante :
  - `mov r0, r0`
- Cas d'utilisation
  - Implémentation d'un délai (consommation de cycles d'horloge)
  - Alignement
  - Réservation d'un espace mémoire pour des instructions
  - Mesure du temps d'exécution le plus petit possible
  - Remplissage du *pipeline* pour éviter un décodage erroné
  - ...

33

Cours ASM - Institut REDS/HEIG-VD

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 mesurer 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)



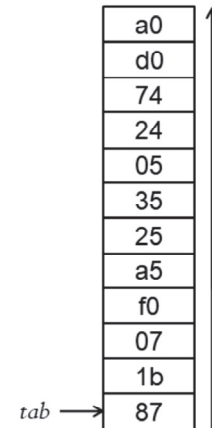
⇒ Indiquez les valeurs de r1 et r2.  
(le processeur est configuré en *little-endian*.)

```

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
```

R1

R2



## 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.