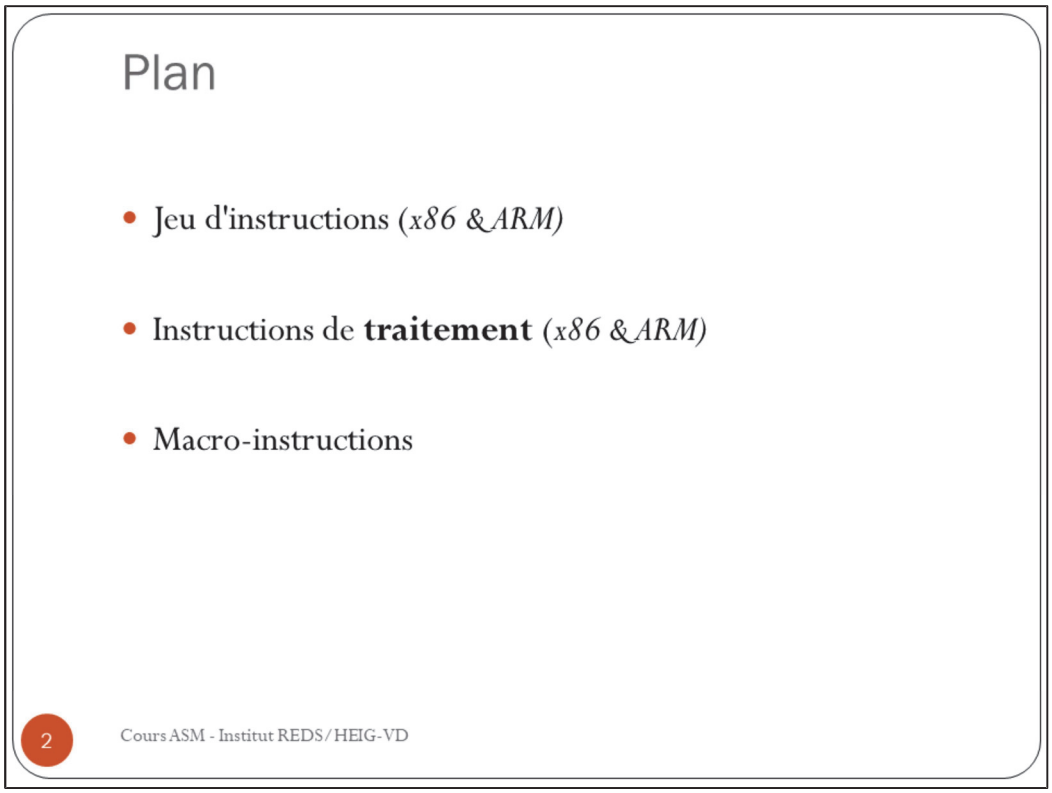
ReDS
Reconfigurable & embedded Digital Systems

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

# Programmation assembleur (ASM)
## *Instructions de traitement*

Prof. Daniel Rossier

Version 1.6 (2017-2018)




Dans les prochains chapitres, nous découvrirons les jeux d'instructions des processeurs *x86* et *ARM*. Il ne s'agira en aucun cas d'une étude exhaustive des instructions, mais il s'agira plutôt d'aborder les instructions fondamentales. Les instructions sont divisés en ***3 groupes fondamentaux:***

- Les instructions de **traitement**

Ce sont les instructions agissant sur les registres uniquement dans le contexte d'opérations arithmétiques ou logiques. Elles comprennent également les instructions de comparaisons et de tests.

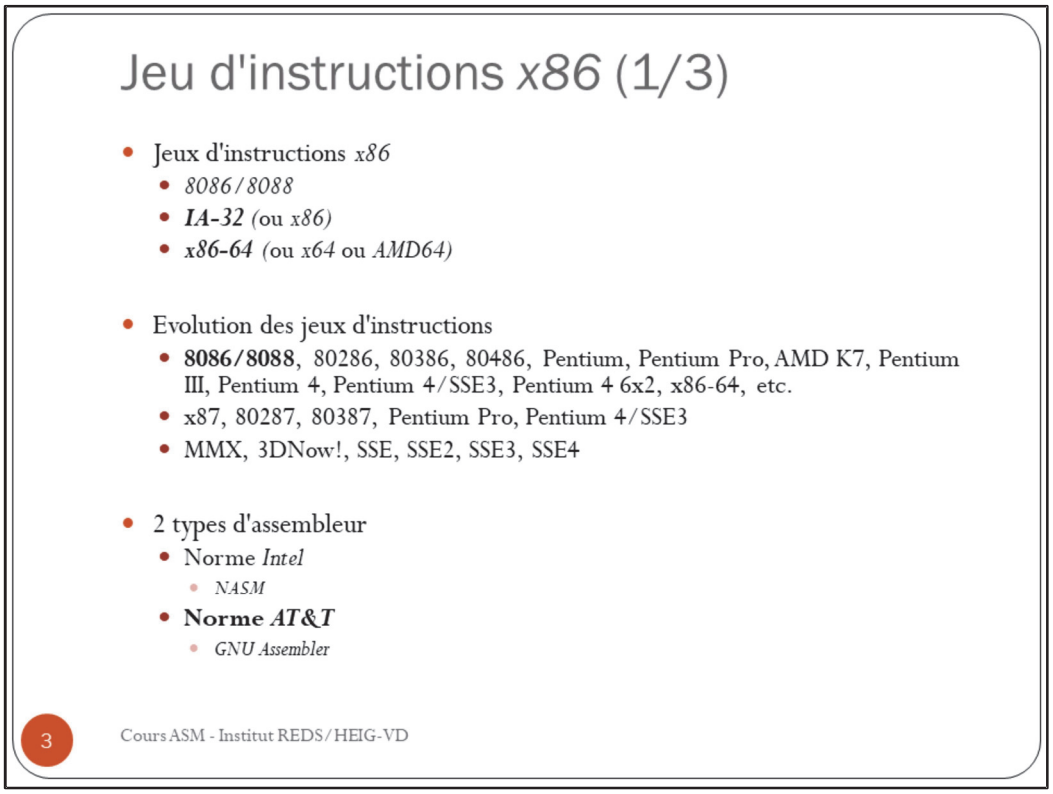
- Les instructions de **transfert de données**

Ce sont les instructions utilisées dans le cadre d'échange de données entre la mémoire et les registres; ces instructions permettent de gérer les interactions entre le cœur de processeur et les périphériques (*RAM, I/O, etc.*). Les instructions de transfert de données sont associées à différents modes d'adressage.

- Les instructions de **branchement**

Finalement, les instructions de branchement permettent de contrôler le déroulement d'un programme durant son exécution.

Ce chapitre se concentre sur les instructions de traitement.

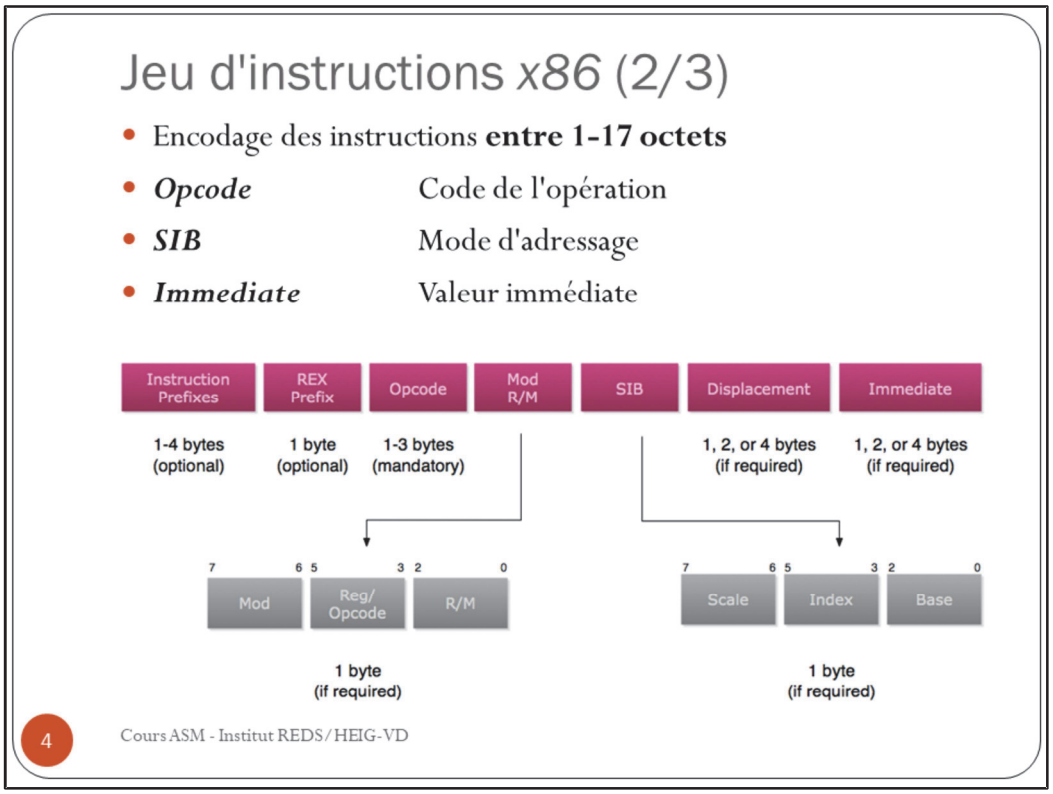



Comme nous l'avons vu dans le chapitre précédent, l'architecture *x86* a été reprise par de nombreux fabricants et a subi d'importantes évolutions matérielles; il en va également de son jeu d'instructions qui n'a sans cesse évolué.

Le jeu d'instructions *IA-32* utilisé par la société *Intel* représente l'une des implémentations pour l'architecture *x86.* C'est un jeu d'instruction largement répandu pour les architectures 32 bits.

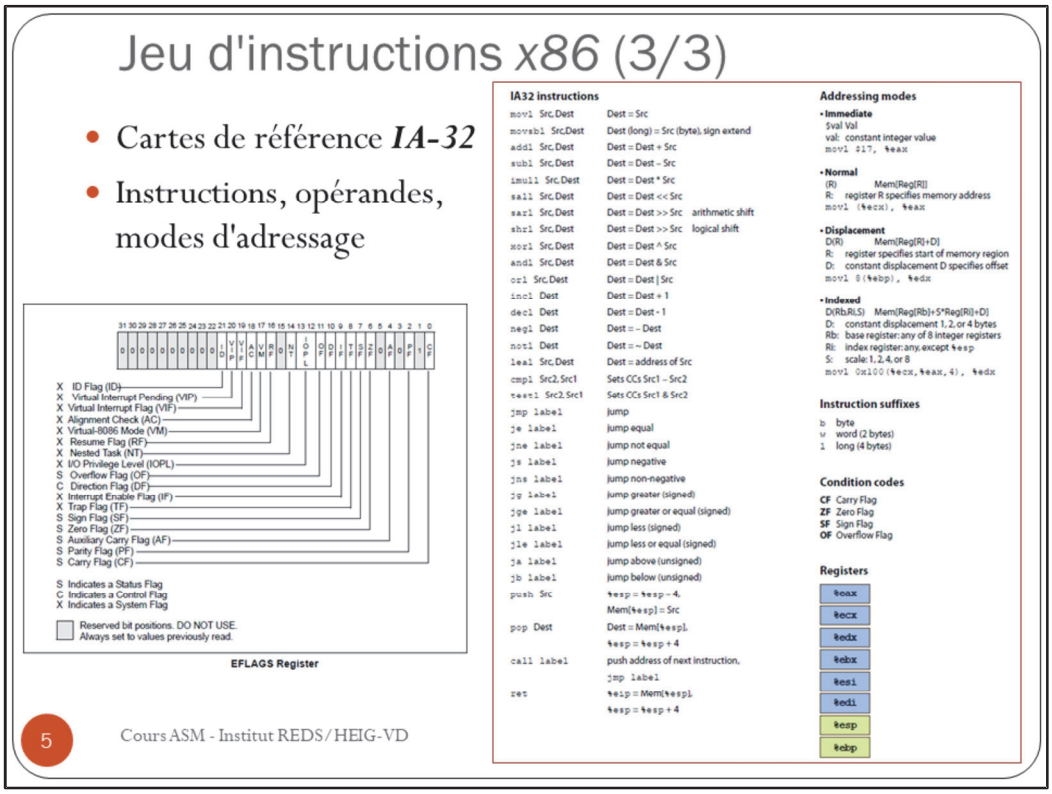
Les compilateurs d'assemblage (ou *assembleurs*) se déclinent principalement en deux familles : ceux suivant la norme d'*Intel* (assembleur *nasm, yasm, etc.*) et ceux suivant la norme *AT&T* (*GNU assembler*), norme qui sera considérée dans ce cours. Ces deux normes différent principalement au niveau de la syntaxe du langage, mais un code *Intel* peut être facilement converti en un code *AT&T*, et vice-versa.

Par la suite, nous utiliserons le terme "*instructions* x86" pour faire référence en fait au jeu d'instructions *IA-32*.




Une instruction représente **l'assemblage** d'une instruction avec ses opérandes et autres valeurs associées (valeur immédiate, *offset*, etc.); du point de vue de la programmation, les opérandes correspondent aux *arguments* de l'instruction. En revanche, du point de vue du processeur, l'instruction exécutée **doit contenir** toutes les informations nécessaires à l'exécution de celle-ci.

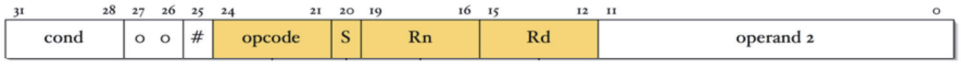
L'encodage d'une instruction *x86* est présenté ci-dessus; la taille d'une instruction peut être variable, entre 1 et 17 *bytes*, et comprend différents champs dont les plus importants sont les suivants: ***rex prefix*** (informations sur le-s registre-s utilisé-s comme opérande-s), ***opcode*** (code opératoire de l'instruction), et ***immediate*** (valeur immédiate). De plus, les champs (*scale, index, base, displacement*) sont liés aux modes d'adressage et donnent une très grande flexibilité lors d'accès mémoire; dans ce cas on se réfère à des instructions de transfert qui seront étudiées ultérieurement.




La carte de référence rapide ci-dessus donne un résumé des instructions *x86* de base ainsi que les différents *flags* du registre d'état et modes d'adressage.

## Jeux d'instructions ARM (1/4)

- Différentes (micro-)architectures et jeux d'instructions
  - **v4** (*ARM-7*), **v5** (*ARM-9*), **v6** (*ARM-11*), **v7** (*Cortex-Ax/Mx/Rx*), **v8** (*ARM 64-bit*)
- Instructions sur **32 bits** (ou 16 bits en mode *Thumb*)



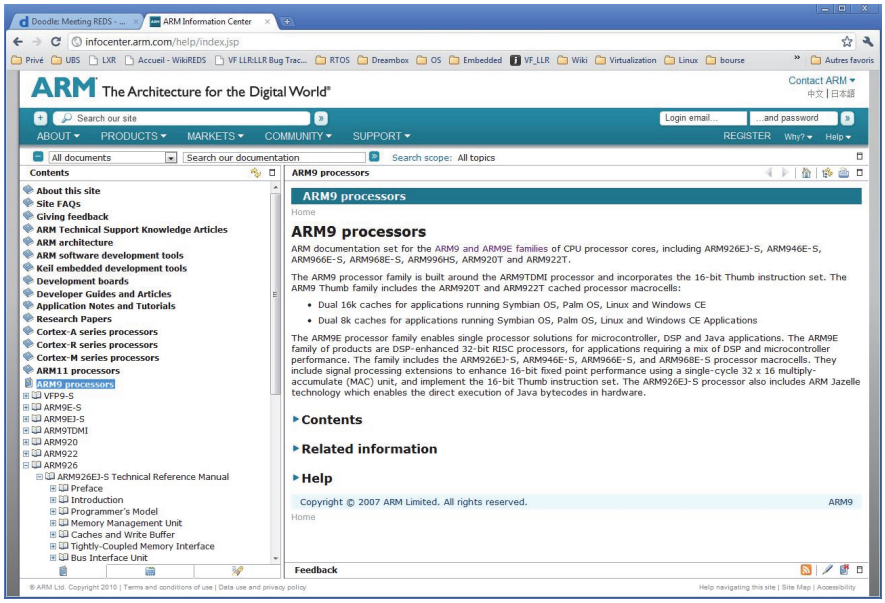

- Exécution des instructions en **un seul cycle machine**
- **Exécution conditionnelle** des instructions
  - Un code d'exécution fait partie de l'instruction.

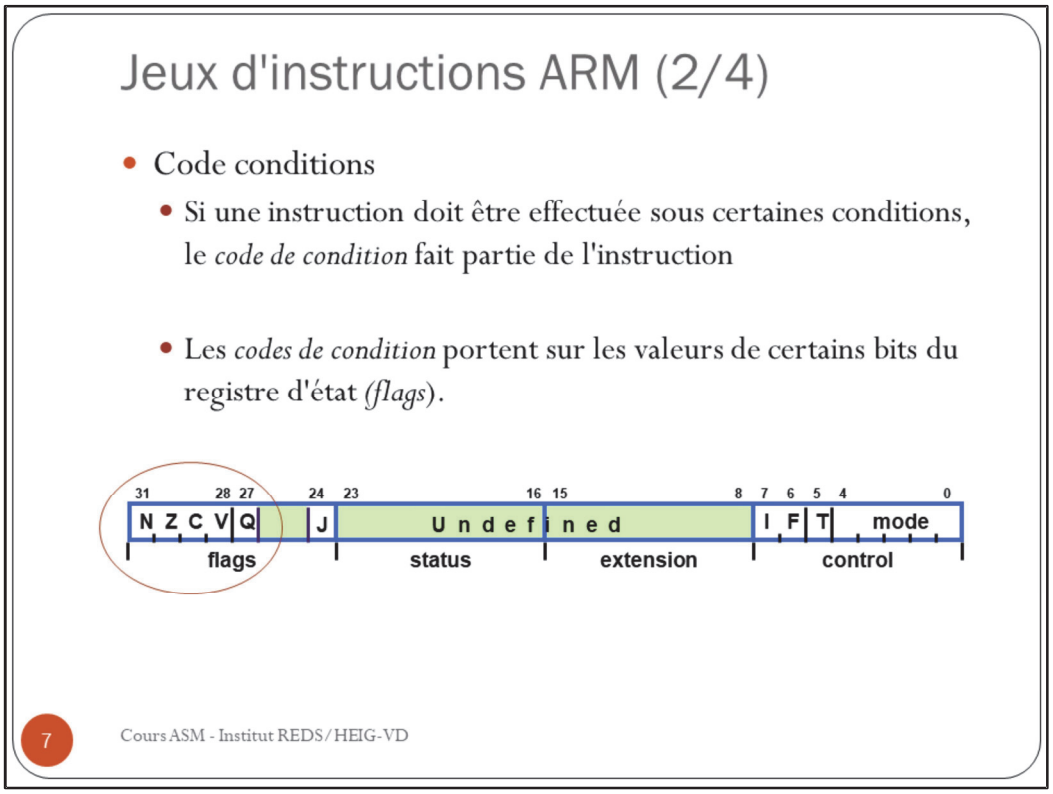
6 Cours ASM - Institut REDS/HEIG-VD

Sur une architecture 32-bit, toutes les instructions ARM sont codées sur 32 bits; cela comprend les différentes parties de l'instruction, à savoir le code opératoire (*opcode*), les opérandes, les *flags*, les valeurs immédiates (constantes), etc.

La majeur partie des instructions sont exécutées en un seul cycle (à l'exception des instructions de transferts et de quelques instructions de traitement particulières) et peuvent être exécutées de manière conditionnelle.

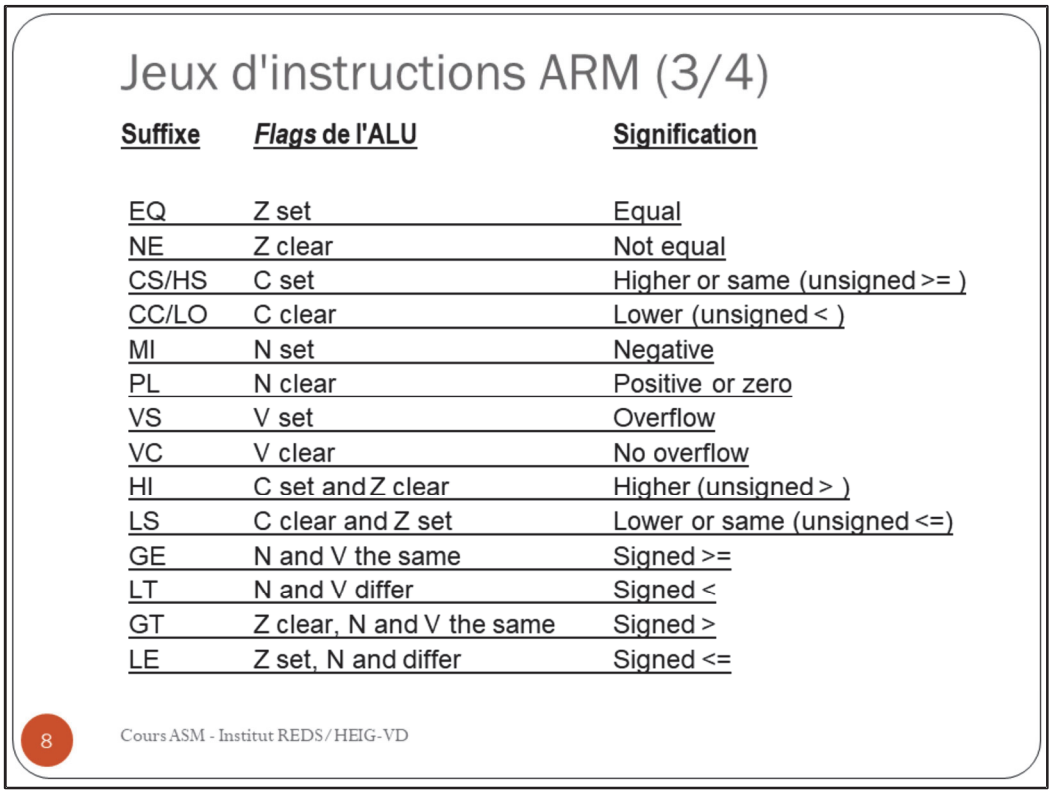
Un site de référence pour les processeurs ARM: ***http://infocenter.arm.com***



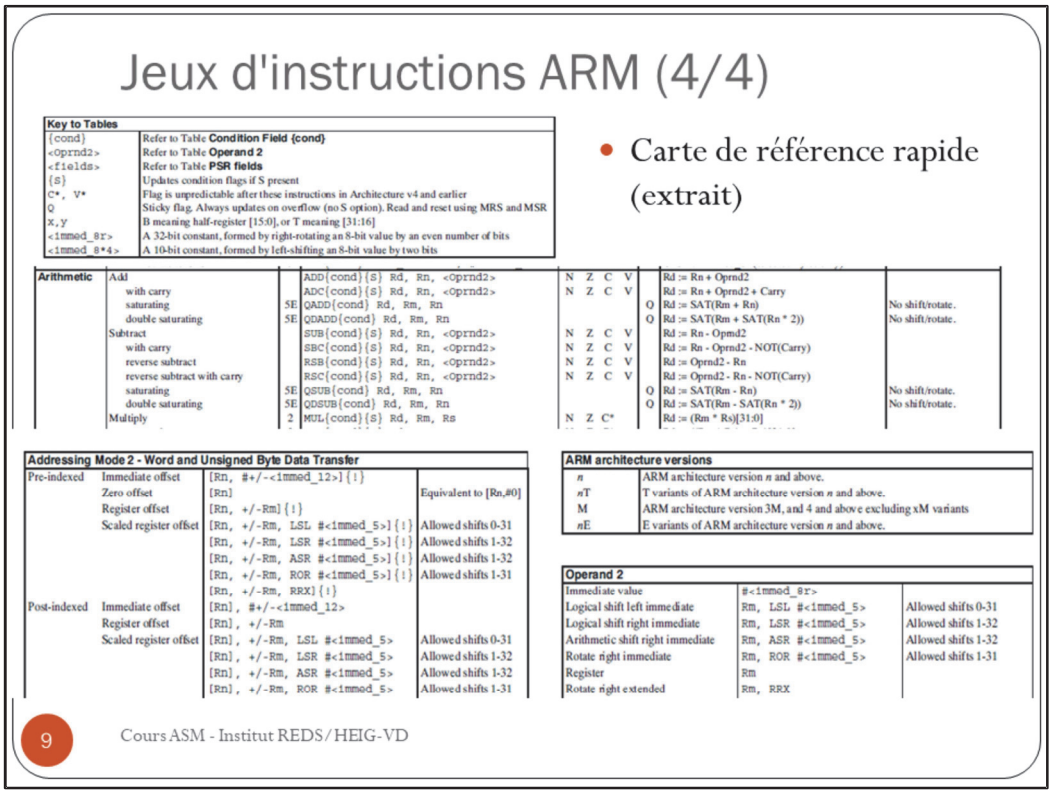



Les *flags* du registre d'état (CPSR) sont mis à jour lors d'une exécution d'une instruction arithmétique par exemple, ou lors d'une comparaison.

Toutefois, comme nous le verrons plus tard, une instruction arithmétique nécessite l'ajout d'un suffixe spécial (lettre '**s**') pour indiquer que la mise à jour doit s'effectuer.

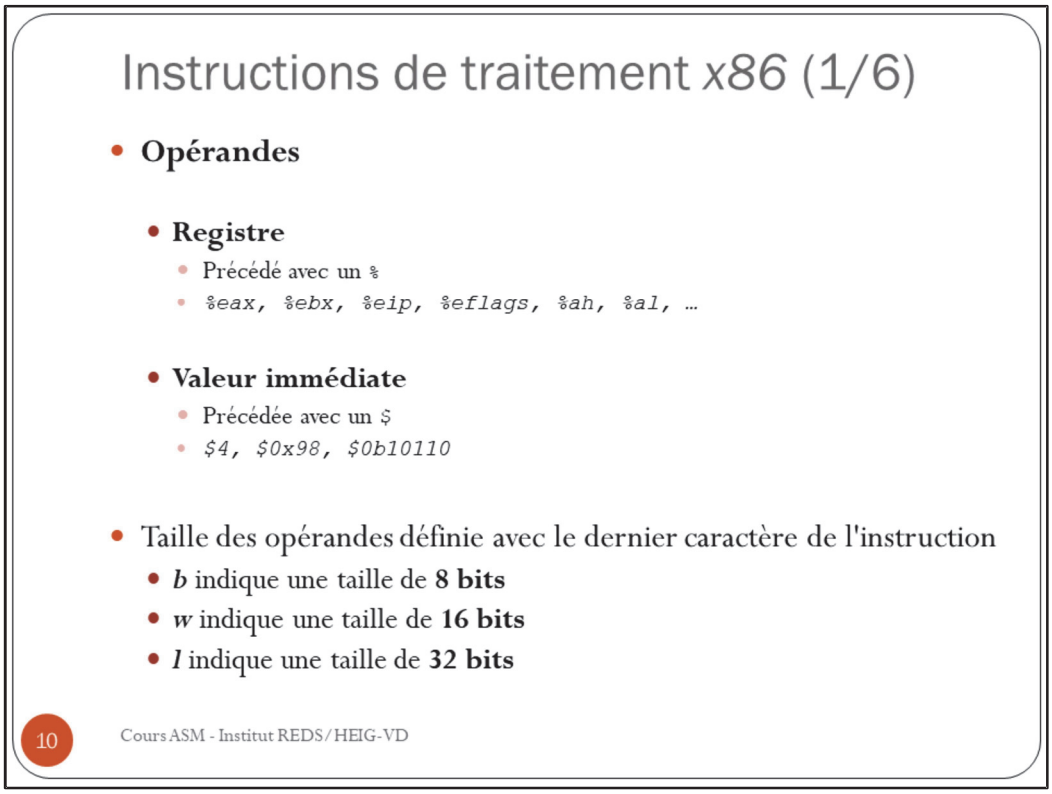



Par exemple, le code de condition ***eq*** est utilisé lorsqu'il y a égalité lors d'une comparaison. L'instruction de comparaison effectuant une soustraction, le résultat de cette soustraction aura pour effet de mettre le flag **Z** à 1 dans le registre CPSR, et donc l'instruction avec ce code de condition teste la valeur de ce flag. La valeur 1 correspond à "**Z set**" dans le tableau ("**Z clear**" signifie que le bit Z est à 0).



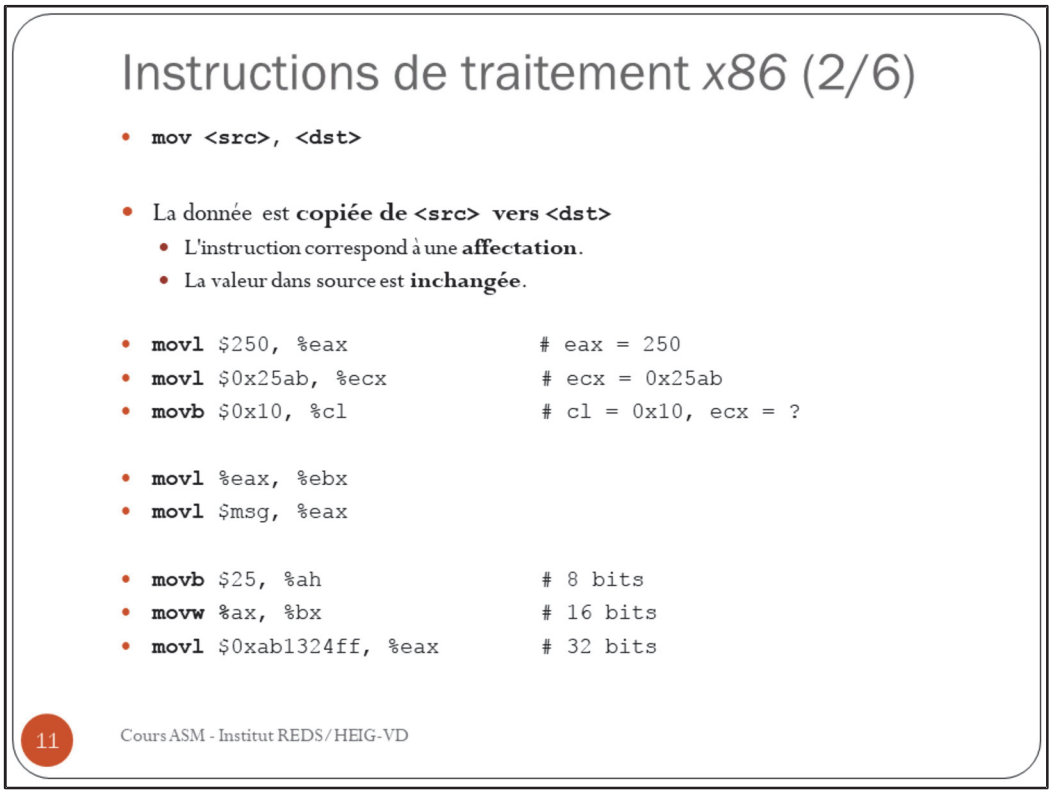

Tout comme pour l'assembleur *x86*, il existe des cartes de référence rapide présentant l'ensemble des instructions les plus importantes. Ces cartes sont très utiles lors du développement, en particulier avec l'assembleur *ARM*, car les instructions sont plutôt rigides au niveau des opérandes qu'elles manipulent. Le tableau indique de manière très rigoureuse le format des opérandes, modes d'adressage et les différentes formes des instructions qui peuvent être validés par le compilateur.

Dans ce sens, l'assembleur *ARM* peut être considéré comme plus contraignant que l'assembleur *x86*.

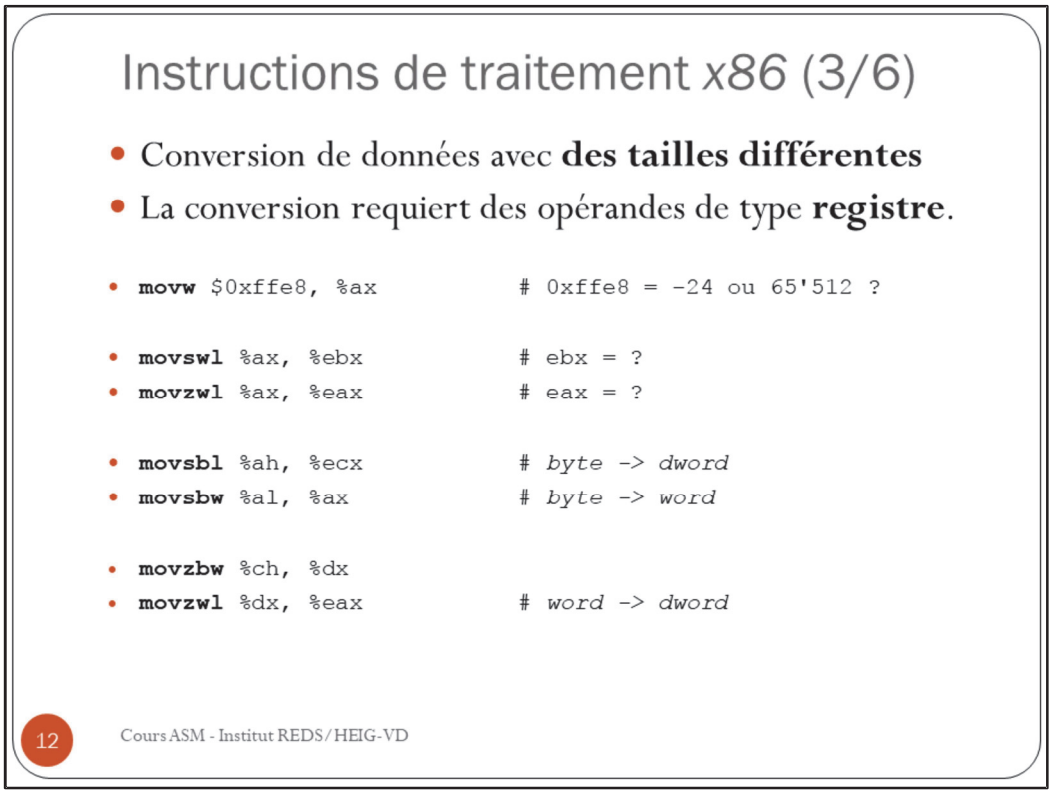



Lorsqu'une instruction peut contenir un registre comme opérande, celui-ci doit être préfixé avec **%**

La valeur immédiate représente une valeur numérique qui sera encodée avec l'instruction. Cette valeur doit être préfixée avec le symbole **$**



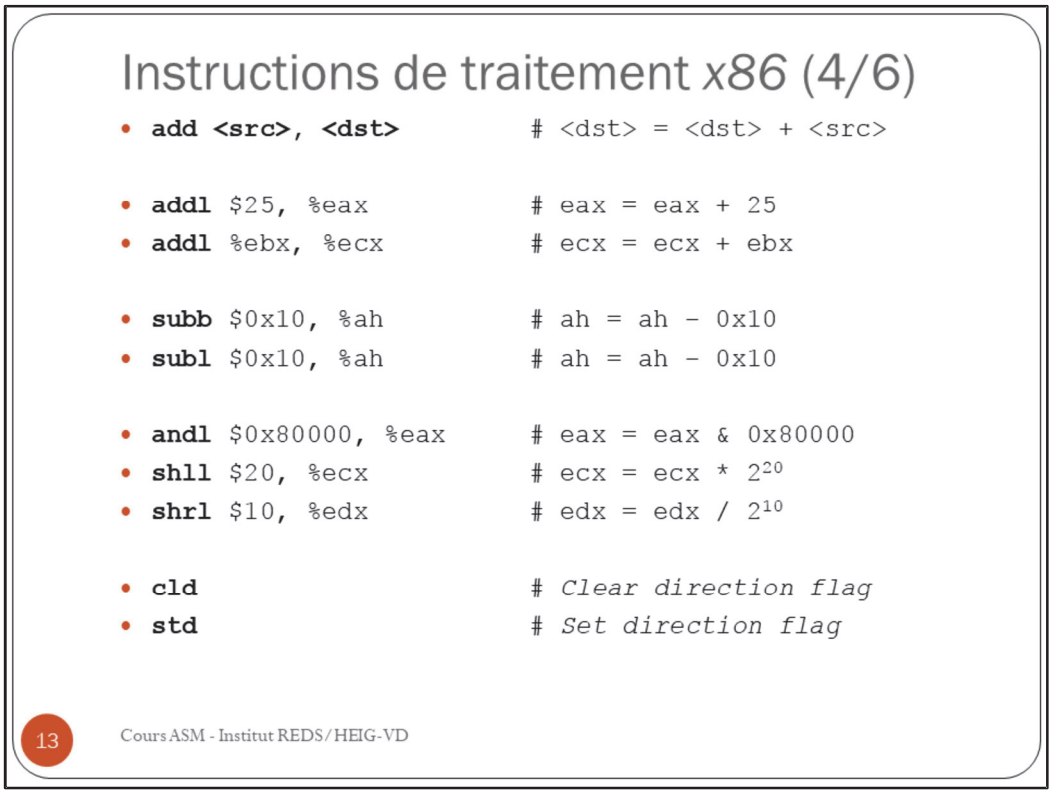

La taille des opérandes peut être intégrée à l'instruction à l'aide d'un suffixe (cf ci-dessus). S'il n'y a pas d'ambiguïté, il n'est en principe pas nécessaire de rajouter le suffixe, bien que le rajout soit vivement recommandé pour faciliter la lecture. Normalement, il n'y a pas d'ambiguïté dans le cas de transfert mémoire (nous verrons les instructions de transfert dans un autre chapitre).




L'instruction *mov* peut être utilisé avec un suffixe composé de trois lettres permettant de convertir la taille d'une donnée à une **taille supérieure**. Ce type d'instruction nécessite l'utilisation de deux opérandes de type registre.

La première lettre du suffixe détermine s'il faut considérer le signe de la valeur source, *i.e.* s'il faut tenir compte de son bit de poids fort ou non, durant sa conversion. Si la valeur est négative, la valeur convertie à la nouvelle taille conservera le signe négatif et sa représentation sera conforme à une valeur négative. Si la valeur est positif, les bits de poids fort de la valeur cible seront nuls. La lettre ***s*** *(signed-extended)* signifie que l'on souhaite préserver le signe, alors que la lettre ***z*** *(zero-extended)* considère une valeur non signée et les bits de poids fort vaudront 0.

Les deux lettres suivantes indiquent le type de conversion; ***bw*** signifie *byte-to-word (8-to-16)*, ***bl*** *byte-to-dword (8-to-32)*, ***wl*** *word-to-dword (16-to-32)*, etc.

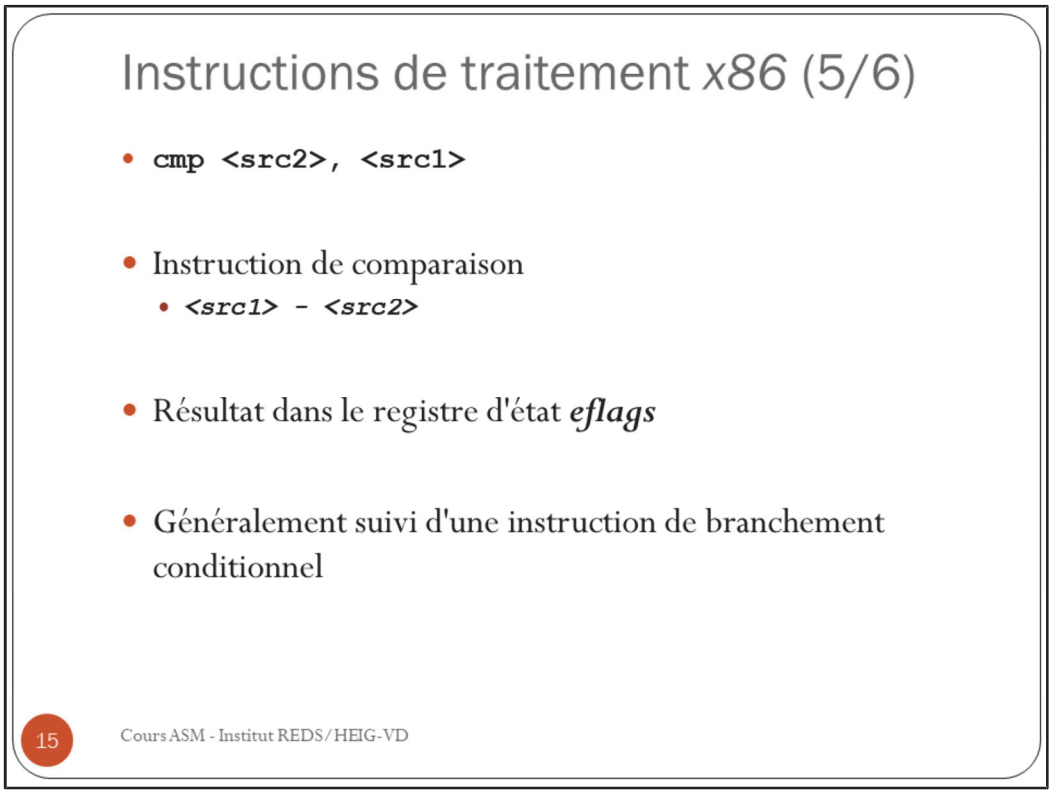



Les exemples ci-dessus montrent l'utilisation d'instructions de traitement logiques. La taille des opérandes peut être également explicite ou implicite.

Les instructions *cld* et *std* sont des exemples d'instructions qui agissent uniquement sur le registre d'état *eflags*. Dans le cas présent, elles touchent uniquement le bit de direction.
Ces deux instructions peuvent donc être utilisées conjointement avec d'autres instructions qui se réfèrent à ce *flag* de direction; nous découvrirons plus tard que c'est le cas, par exemple, pour une instruction permettant la copie rapide d'un bloc mémoire, le *flag* de direction indiquant alors si les adresses doivent être incrémentées ou décrémentées (sens de la copie).

# Instructions de traitement x86 (x/6)

- Mise à jour automatique du registre eflags
- …
- A l'exception de mov




L'instruction de comparaison ***cmp*** permet de comparer les valeurs contenues dans deux registres, qui constituent les deux opérandes de l'instruction. Il s'agit en fait d'une soustraction dont le résultat sera évalué à l'aide du registre *eflags*.

Les codes de condition seront étudiés avec les instructions de branchement, dans le chapitre suivant.

# Instructions de traitement x86 (6/6)

- Soit les variables *x, y, z* associées aux registres *eax, ebx, ecx* respectivement.

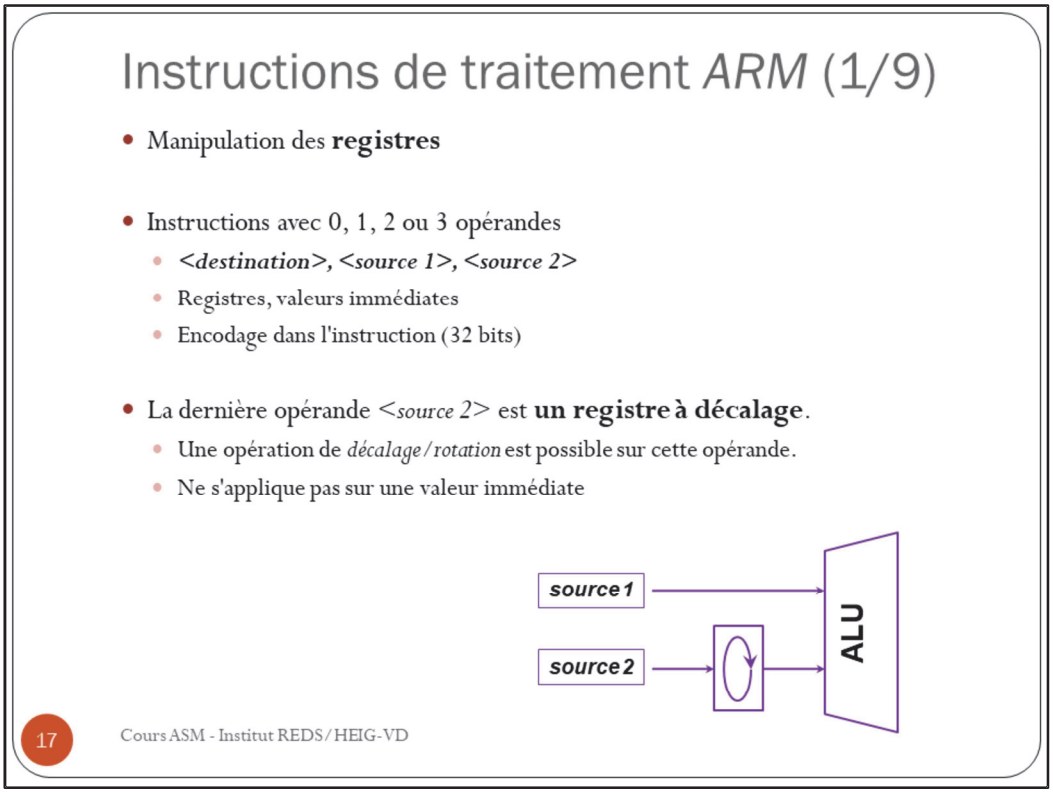
⇒ Codez en assembleur les opérations ci-dessous. Aidez-vous des opérateurs de décalage arithmétique.

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

```
# z = 3x – 5
```

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

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



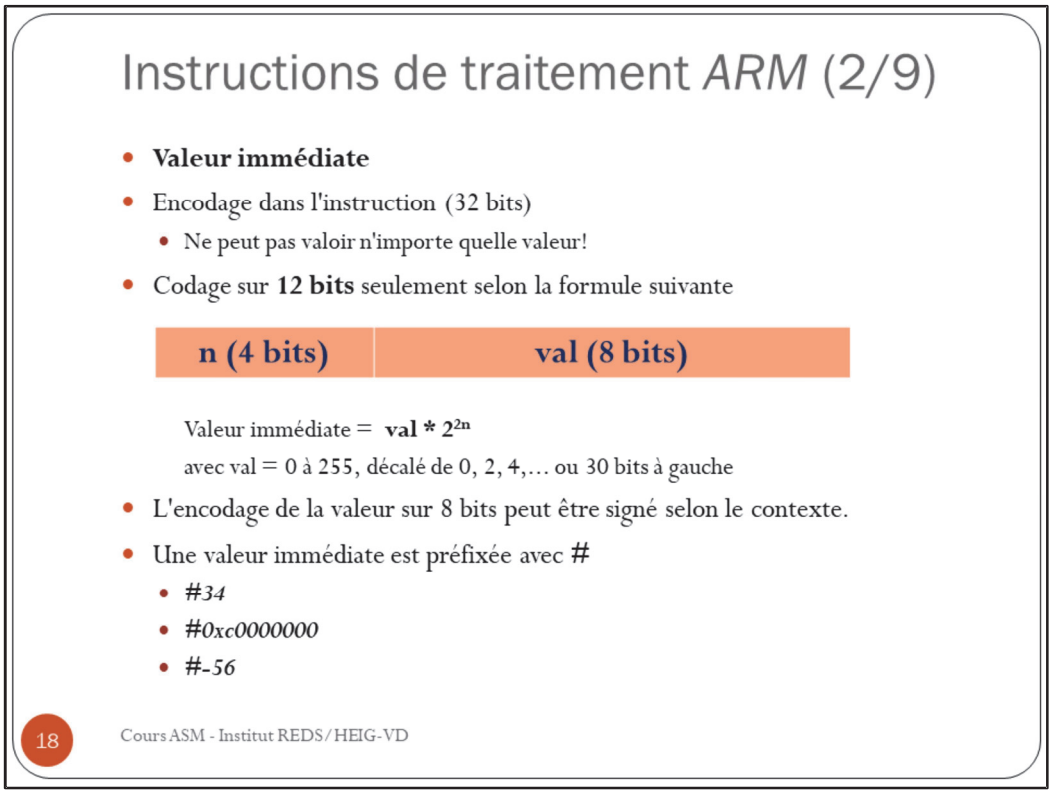

Comme nous l'avons vu précédemment, les instructions de **traitement** permettent de manipuler des données au niveau de **registres**. Alors que sur *x86*, les mnémoniques correspondant aux instructions de traitement pourront aussi être utilisés pour les instructions de transfert, ce n'est pas le cas sur *ARM*. Les instructions de traitement sont uniquement dédiées aux opérandes de type registre.

Une particularité des instructions de traitement sur *ARM* porte sur l'utilisation de la deuxième opérande : cette opérande est associée à un registre de décalage permettant d'effectuer un décalage ou une rotation de bits sur cette opérande.

Une opérande peut être soit un registre, soit une valeur numérique appelée ***valeur immédiate***.

Une valeur immédiate ne peut être utilisée que sur la dernière opérande de l'instruction.

Les opérateurs de décalages **ne s'appliquent pas** sur une valeur immédiate.



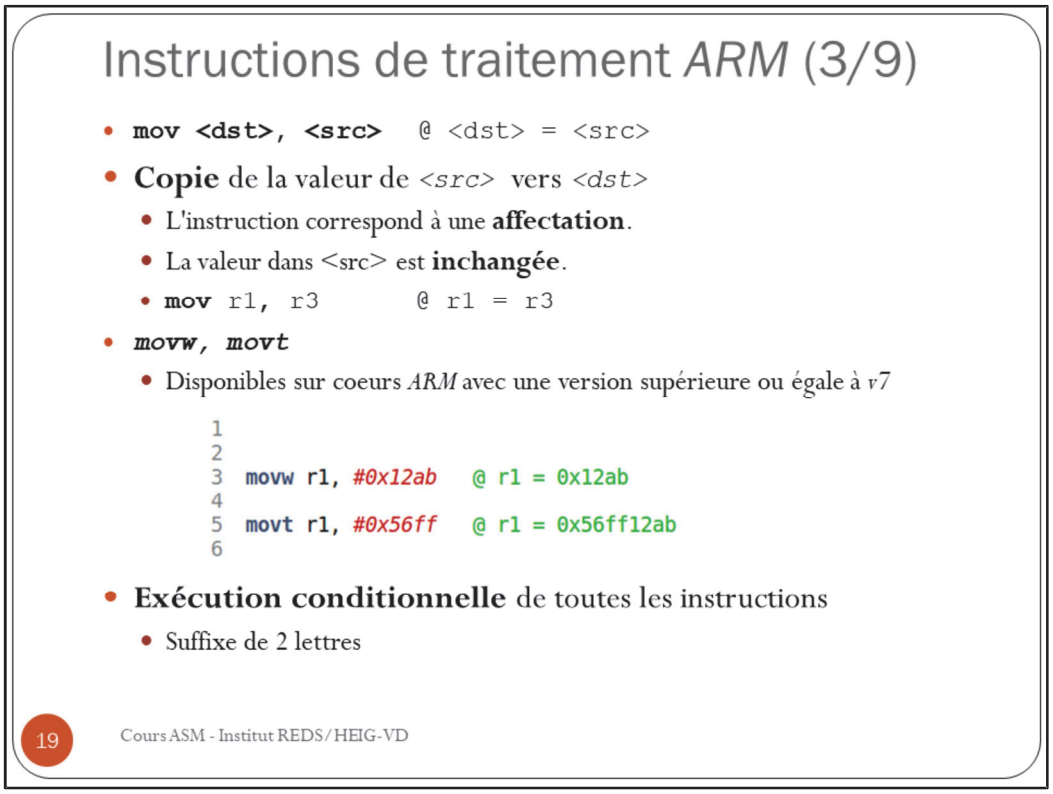

Nous avons vu qu'une instruction ARM est codée sur 32 bits, comprenant tout ce qui caractérise l'instruction (*opcode*, opérandes, valeur immédiate, etc.). Cette approche ne permet donc pas d'utiliser une valeur immédiate sur 32 bits, mais nécessite un codage particulier de celle-ci.
Seul 12 bits sont réservés pour la valeur immédiate. Avec 12 bits, il est possible d'exprimer une valeur dans un intervalle de 0 à $2^{12} - 1$ (4095); ce qui serait un peu limitatif. C'est pourquoi les 12 bits sont utilisés d'une autre manière: on utilise les 8 bits de poids faible pour coder une valeur entre 0 et 255, puis les 4 bits de poids fort pour coder un exposant. Ainsi une valeur immédiate (*imm*) est calculée selon la formule suivante: **`imm = val(8 bits) * 2^2*n`**, `n` étant l'exposant.

De cette manière, une valeur immédiate peut prendre les valeurs entre 0 et 255, puis tous les nombres qui peuvent s'exprimer comme étant le décalage de 2 bits d'une valeur entre 0 et 255.

- Si *n* vaut 0, on obtient toutes les valeurs entre 0 et 255.
- Si *n* vaut 1, on obtient toutes les valeurs entre 0 et 255 multipliées par 4
- Si *n* vaut 2, on obtient toutes les valeurs entre 0 et 255 multipliées par 16
- Si *n* vaut 3, on obtient toutes les valeurs entre 0 et 255 multipliées par 64

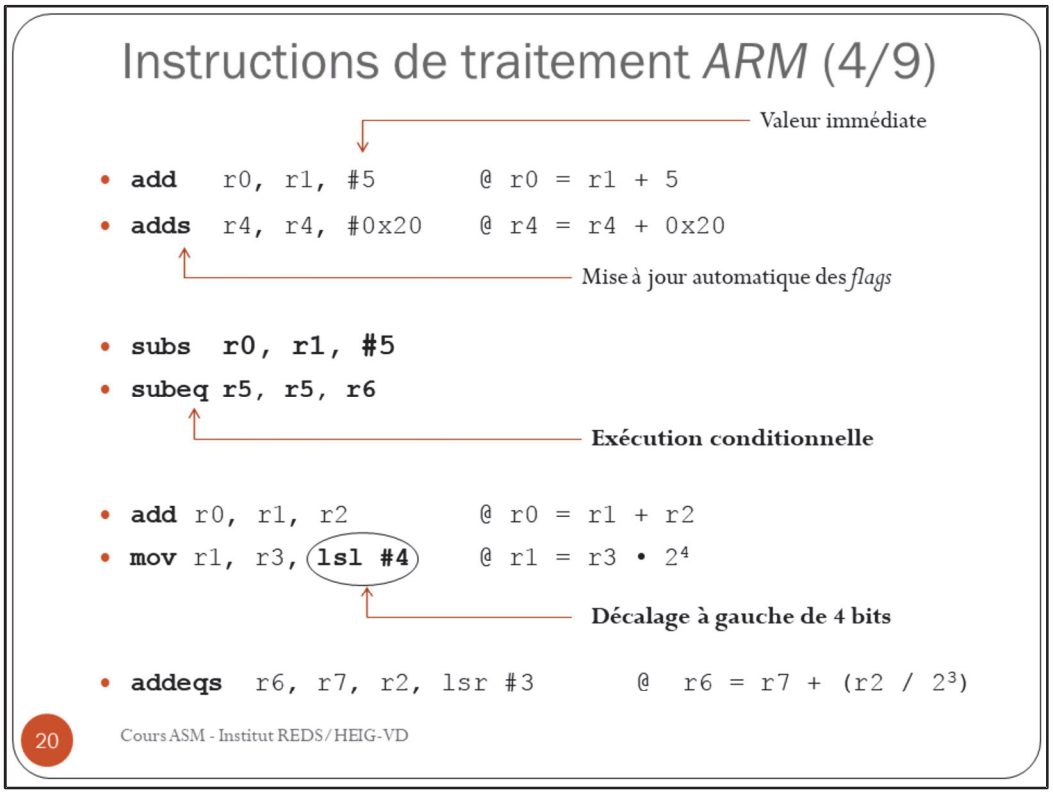
…et ainsi de suite. On voit donc apparaître les multiples de 4, 16, 64, etc. (jusqu'à la valeur maximale donnée par le facteur 255).
Cela permet ainsi de donner une meilleure *amplitude* de la valeur immédiate.

Lorsqu'il n'est pas possible de coder une valeur immédiate sous cette forme, il faut utiliser une autre approche, détaillée ci-après et dans le chapitre suivant.




Une différence essentielle avec la norme *AT&T* de l'assembleur *x86* réside dans le sens de fonctionnement des opérandes : alors que l'opérande source apparaît en second (à droite de la virgule) sur *x86*, elle apparaît toujours en premier (à gauche de la virgule) en assembleur *ARM*. A l'instar du *x86*, l'instruction *mov* permet de copier une valeur dans un registre. L'opérande source peut être soit un registre, soit une valeur immédiate. Nous avons découvert que dans ce dernier cas, la valeur immédiate est encodée d'une certaine manière. Depuis les version *v7* des architectures *ARM*, deux nouvelles instructions ont fait leur apparition : il s'agit de *movw* et *movt* qui permettent de stocker une valeur quelconque sur 32 bits. Ces deux instructions prennent un registre commun comme première opérande, puis une valeur immédiate sur 16 bits comme seconde opérande. Elles doivent être exécutées dans cet ordre *(movw, movt*) et ne pas être "*trop distantes*" l'une de l'autre (en principe, les deux instructions se suivent). La première instruction permet de stocker les 16 bits de poids faible de la valeur, alors que la seconde les 16 bits de poids fort; cette dernière opère un décalage de 16 bits à gauche d'une valeur immédiate de 16 bits.
Il existe une autre manière générale de manipuler des valeurs 32 bits que nous étudierons dans le chapitre portant sur les instructions de transfert.

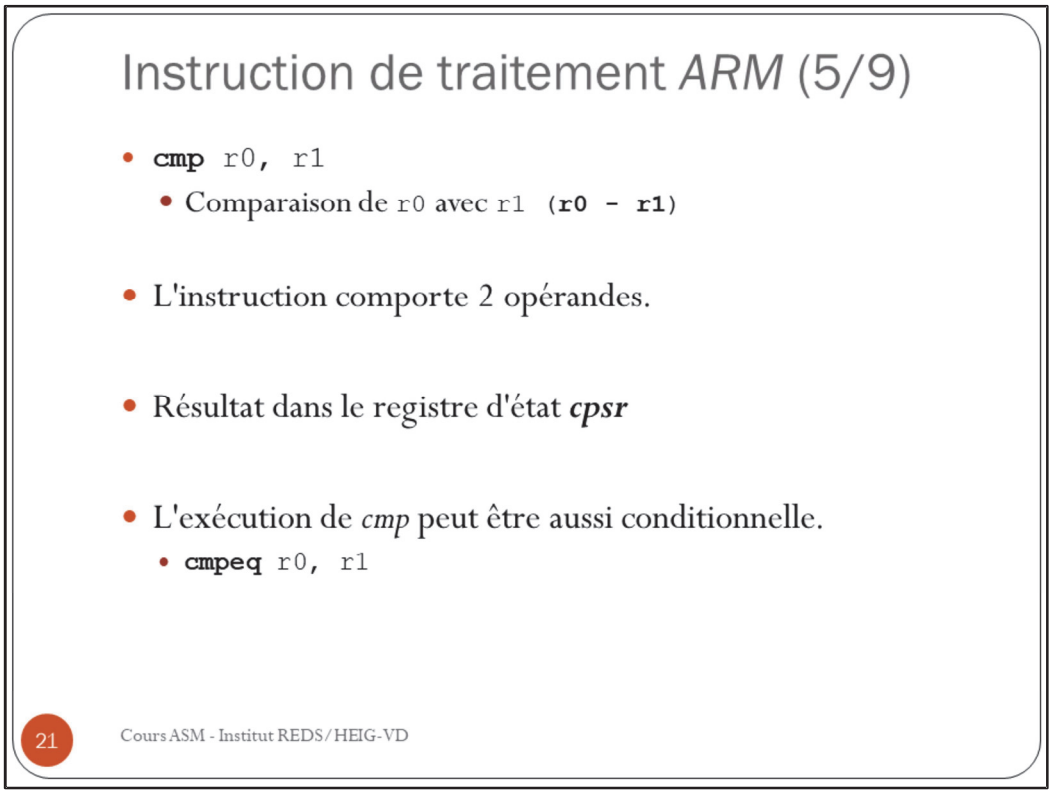
Une autre différence majeure porte sur **l'exécution conditionnelle** des instructions *ARM*. En effet, un code de condition sur deux lettres peut être utilisé pour décider si l'instruction doit être exécuté ou non. Bien entendu, ce code de condition se basera sur l'état des *flags* du registre d'état (registre *cpsr*).



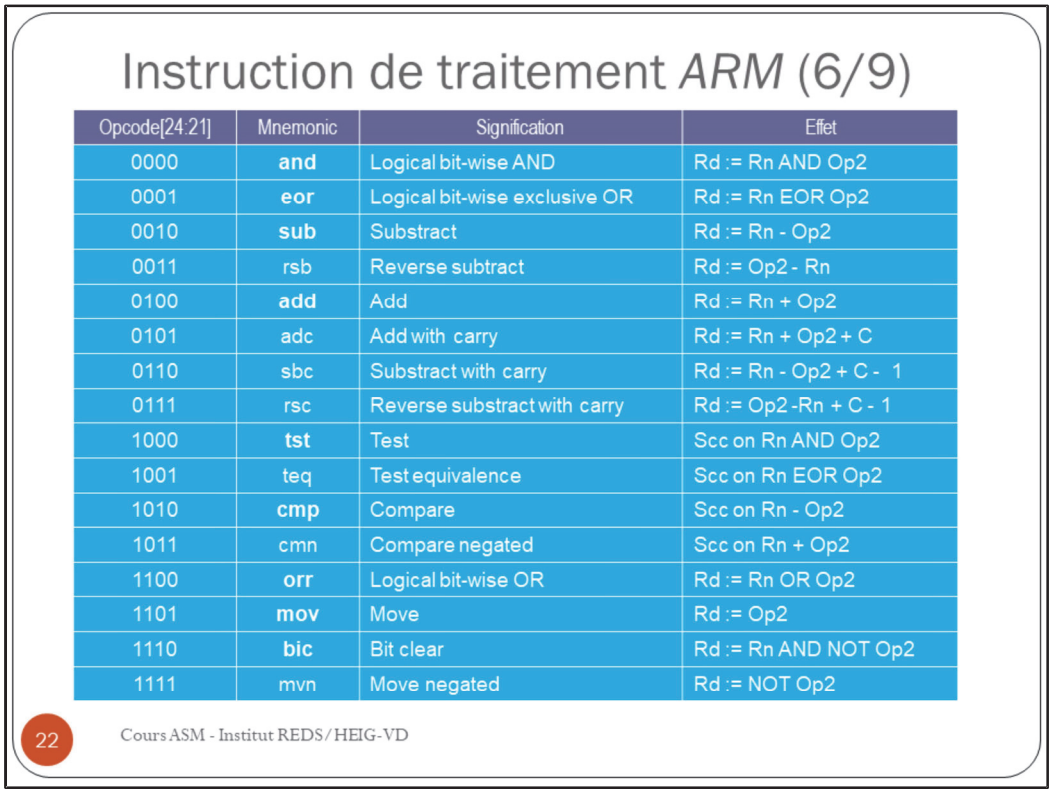

Dans les exemples ci-dessus, on voit qu'une instruction ARM peut se décliner différemment (différents mnémoniques) selon que l'on veut une mise à jour automatique des *flags*, ou une exécution conditionnelle de l'instruction.

Dans le premier cas, l'instruction *adds* aura pour effet de mettre à jours les *flags* du *cpsr* en fonction du résultat de l'addition (un résultat nul mettra le bit **Z** à 1 (*Z set*).

L'exécution conditionnelle de l'instruction *sub* dépendra des *flags* du *cpsr*. Ainsi, dans cet exemple, en fonction du résultat de la première soustraction, la seconde soustraction sera effectuée **seulement si le bit Z est à 1** (donc résultat nul de la première instruction).



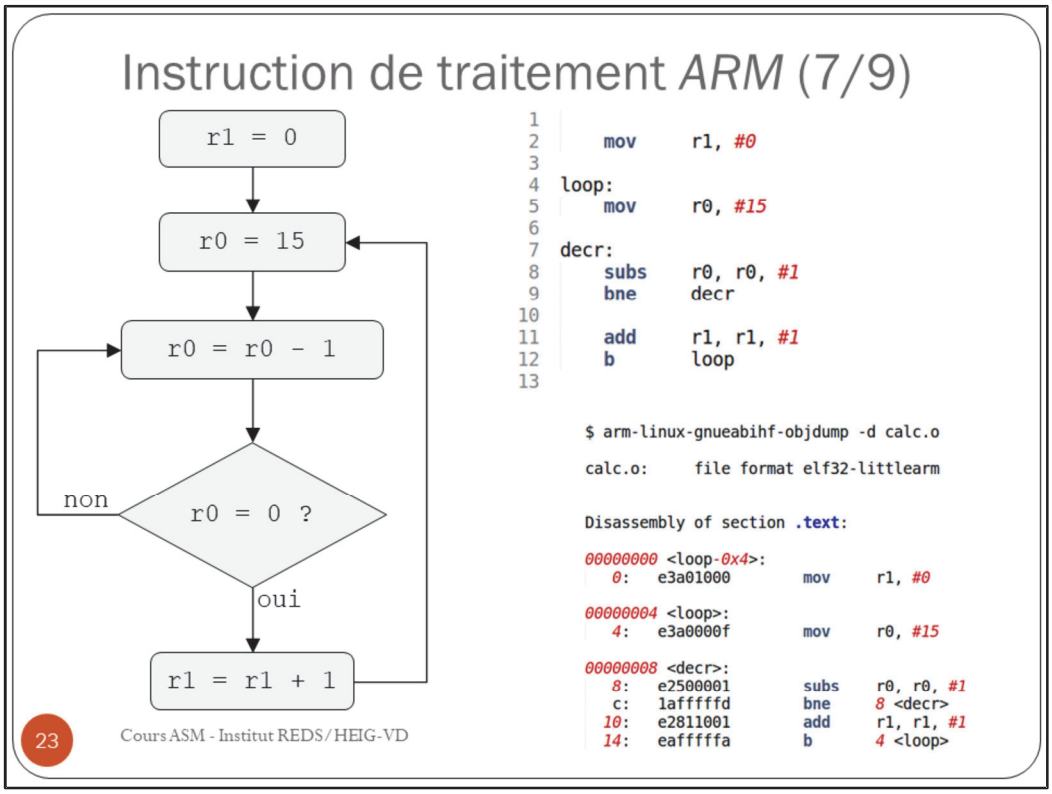

Comme sur *x86*, l'instruction de comparaison effectue une soustraction entre les deux opérandes et le résultat de la comparaison est caractérisé par les *flags* du registre d'état *cpsr.*




Sur ARM, il existe également une instruction de multiplication: *mul Rd, Rm, Rs*

Cette instruction est particulière et son implémentation matérielle peut varier d'une version du processeur à l'autre. Elle est plus complexe et nécessite plusieurs cycles.

Les registres des opérandes doivent être obligatoirement différents les uns des autres (il n'est pas possible d'effectuer: *mul r2, r2, r3* par exemple. De plus, la dernière opérande **ne peut pas** être une valeur immédiate.




L'exemple ci-dessus montre l'utilisation du suffixe **s** pour utiliser une instruction de saut de manière conditionnelle. On remarque l'absence de l'instruction ***cmp***.

Dès que le registre contient la valeur 0, le bit **Z** du registre d'état **CPSR** est automatiquement mis à 1 par l'instruction de soustraction, qui possède le suffixe ***s***.

L'instruction de branchement suivante, avec la condition "***ne" (not equal),*** ne sera plus exécutée.

# Instruction de traitement *ARM* (8/9)

- Soit les registres suivants associés aux variables x, y et z respectivement: *r4*, *r5* et *r6*. Aidez-vous des opérateurs de décalage arithmétique.

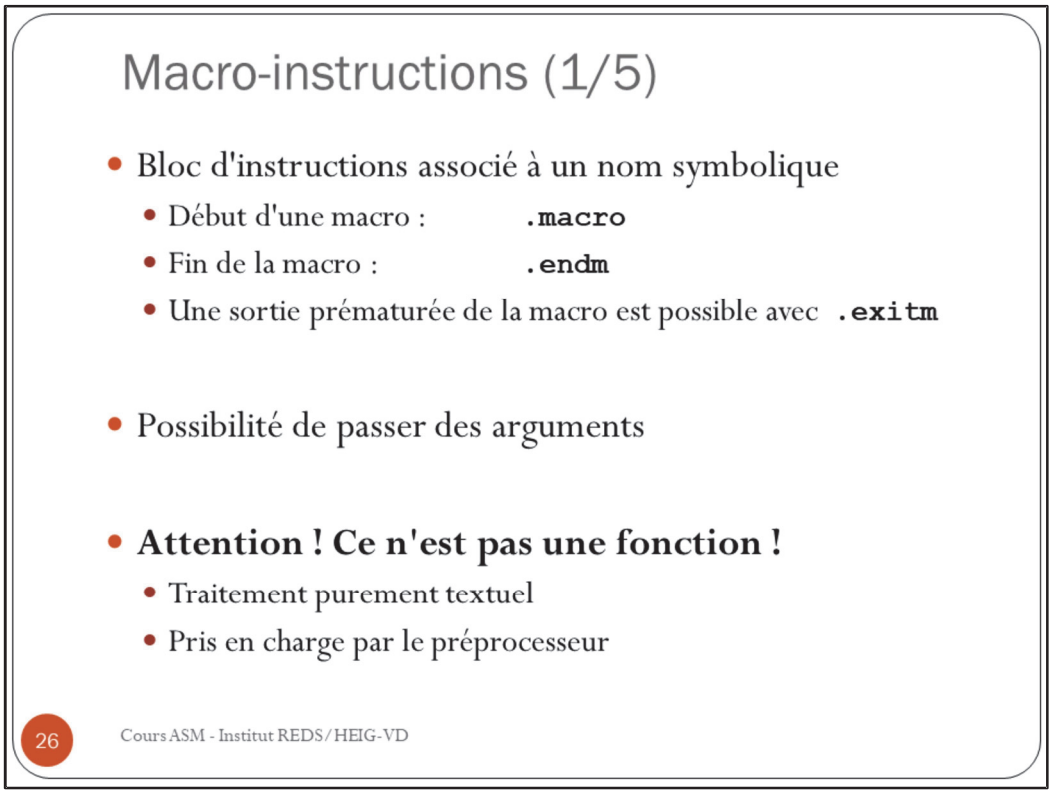
⇒ Codez en assembleur les opérations ci-dessous.

```
@ z = x + y + 4

@ z = 3x – 5

@ z = (x + y) / 4

@ z = 4x + 3
```

# Instruction de traitement *ARM* (9/9)

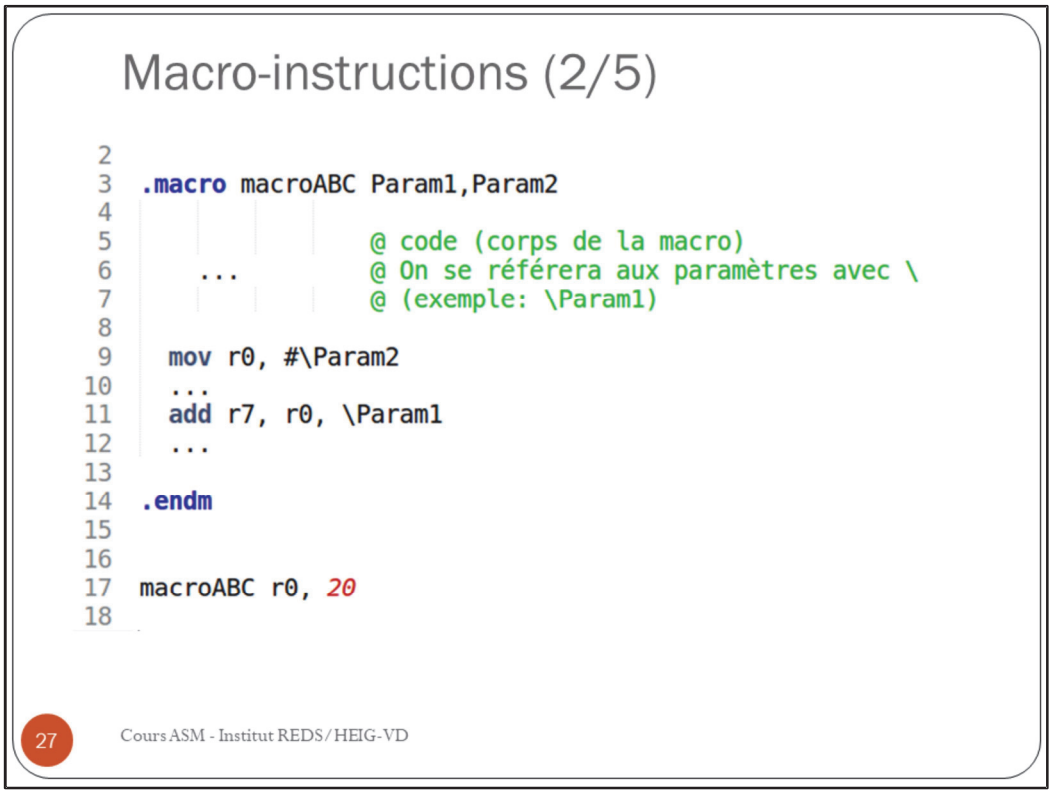
- Commentez le programme ci-dessous et dessinez l'organigramme correspondant.

```
1
2  cmp r5, #5
3
4  cmpne r4, #-5
5
6  addeq r0, r1, r1, lsl #2
7
8  cmp r0, #0
9
10 rsbmi r0, r0, #0
11
```




Une macro-instruction est la description d'une inscription *logique* (qui n'existe pas du tout au niveau du processeur) à partir d'instructions assembleur.

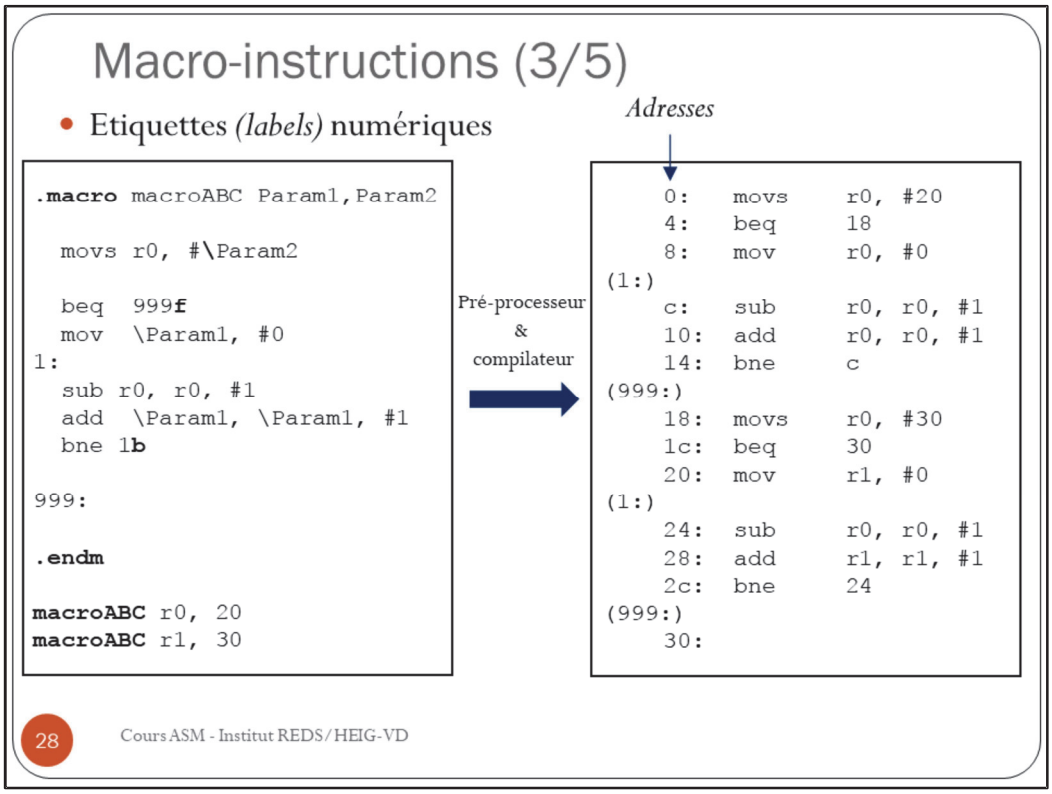
C'est le préprocesseur qui remplacera **chaque occurrence** de l'instruction logique (définie par un nom symbolique) avec le contenu correspondant. Il s'agit donc bien d'une manipulation du code source effectuée par le préprocesseur.




Des arguments peuvent être passés à la macro-instruction. Dans ce cas, chaque occurrence faisant référence aux arguments sera remplacé par l'argument effectif correspondant.

Les arguments sont préfixés avec le symbole "****" dans le corps de la macro.

Lorsqu'on utilise une macro-instruction, et que celle-ci est utilisé à différents endroits dans le code source, il est toujours très important de "visualiser" le texte en sortie avant compilation. Si des *labels* sont utilisés dans le corps de la macro, ces *labels* apparaîtront à plusieurs reprises. Cela est permis sous certaines conditions (***labels* numériques**) et il faut être prudent lorsque l'on s'y réfère.




Le code de gauche est traduit par le préprocesseur et le compilateur vers le code de droite. Dans le code source, on remarque deux appels successifs à la macro *macroABC*, ce qui a comme conséquence l'apparition de deux fois le *label* **1:** et deux fois le *label* **999:** dans le code produit.

Il faut donc indiquer au compilateur à quel *label* l'instruction se réfère, soit le premier qui apparaît en reculant (*backward*), soit celui qui apparaît en avant *(forward)*. L'utilisation des lettres **b** et **f** accolées directement au *label* en suffixe permet de définir la direction dans laquelle rechercher le *label*.

# Macro-instructions (4/5)

- Ecrire en assembleur *x86* une macro-instruction qui stocke dans un registre la multiplication de deux entiers 32 bits (*x***y*), puis qui incrémente *x* et décrémente *y*, *x* et *y* étant des registres.
- Utilisez une fois cette macro une fois dans un programme, avec
  `x = 0xa, y = 2`

# Macro-instructions (5/5)

- Ecrire une macro-instruction *compare* en assembleur *ARM* qui accepte trois paramètres (*src, dest, cond*).
  La macro compare les valeurs de *src* et *dest* en utilisant un code de condition passé dans *cond*. Si la condition est satisfaite, on additionne *dest* à *src* en rajoutant 20 Mo.
- Utilisez cette macro-instruction dans un exemple.

# 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.