HE2B ESI

MICL : TD03 : Instructions de saut `jf` / `jnf` et alternatives

BEJ – DBO – DHA – HAL – NVS – SRE – YVO *

Année académique 2020 – 2021

Ce troisième TD commence par la présentation de l'instruction `cmp`. La notion de *label* est ensuite abordée. L'instruction permettant de faire un saut sans condition, `jmp`, suit. Finalement, les instructions de saut conditionnel dépendant de la valeur de *flags*, `jf` et `jnf`, sont montrées, ainsi que l'implémentation des alternatives *si* et *si-sinon*.

1 Comparaison `cmp`

L'instruction `cmp`[1] a deux opérandes de même taille. Ils peuvent être des registres ou des variables de 8, 16, 32 ou 64 bits. Ils ne peuvent cependant pas être tous les deux des variables. L'opérande de droite peut être un immédiat.

Cette instruction compare l'opérande de gauche à celui de droite. Elle positionne les *flags* du registre `rflags` comme le ferait une soustraction de ceux-ci. Notez bien qu'elle ne modifie aucun de ses opérandes.

La Table 1 donne un résumé de l'instruction `cmp`.

En voici un exemple d'utilisation :

```
1    mov rax, 4
2
3    cmp rax, 4      ; ZF : 1 (4 - 4 == 0), SF : 0 (4 - 4 >= 0)
4    cmp rax, 5      ; ZF : 0 (4 - 5 != 0), SF : 1 (4 - 5 < 0)
5    cmp rax, 2      ; ZF : 0 (4 - 2 != 0), SF : 0 (4 - 2 >= 0)
```

*Et aussi, lors des années passées : ABS – BEJ – DWI – EGR – ELV – FPL – JDS – MBA – MCD – MHI – MWA.

1. https://www.felixcloutier.com/x86/cmp (consulté le 7 février 2020).

Instruction	Effet	Contraintes	*Flags* affectés
`cmp X, Y`	`temp` ← `X - Y`	Registres ou variables de 8, 16, 32 ou 64 bits, pas deux variables à la fois, `Y` peut être un immédiat (8, 16 ou 32 bits)	`OF`, `CF` selon arithmétique [a] `X - Y` `ZF` ← 1 si `temp = 0` (`X = Y`), 0 sinon `SF` ← bit de signe de `temp` : 1 si `X < Y`, 0 sinon

TABLE 1 – Instruction `cmp`.

a. `OF` est levé s'il y a un débordement considérant que `X` et `Y` représentent des entiers codés en complément à deux, tandis que `CF` est levé s'il y a une retenue considérant que `X` et `Y` représentent des entiers codés en représentation par position simple. Pour un rappel et des informations complémentaires sur ces représentations, consultez la TABLE 2(c) et les liens qui y figurent.

```
6     cmp rax, -3      ; ZF : 0 (4 - -3 != 0), SF : 0 (4 - -3 >= 0)
7     cmp rax, -10     ; ZF : 0 (4 - -10 != 0), SF : 0 (4 - -10 >= 0)
8
9     mov rax, -3
10
11    cmp rax, 4       ; ZF : 0 (-3 - 4 != 0), SF : 1 (-3 - 4 < 0)
12    cmp rax, 5       ; ZF : 0 (-3 - 5 != 0), SF : 1 (-3 - 5 < 0)
13    cmp rax, 2       ; ZF : 0 (-3 - 2 != 0), SF : 1 (-3 - 2 < 0)
14    cmp rax, -3      ; ZF : 1 (-3 - -3 == 0), SF : 0 (-3 - -3 >= 0)
15    cmp rax, -10     ; ZF : 0 (-3 - -10 != 0), SF : 0 (-3 - -10 >= 0)
16
17    mov ebx, 42
18    ; cmp rax, ebx    ; compile pas : tailles différentes
```

Remarque Comme pour ce qui concerne les instructions `and`, `or` et `xor` étudiée lors du TD02, la taille de l'immédiat est limitée à 32 bits. Dans le cas où l'opérande de gauche fait 64 bits, l'immédiat sur 32 bits est étendu à 64 bits par extension de signe. L'extrait de code suivant l'illustre :

```
1     mov rax, 0x80    ; rax : 0x00_00_00_00_00_00_00_80
2     cmp rax, 0x80    ; ZF : 1, SF : 0 car extension de signe donne
3     ; cmp 0x00_00_00_00_00_00_00_80, 0x00_00_00_00_00_00_00_80
4
5     mov rax, 0x80808080 ; rax : 0x00_00_00_00_80_80_80_80
6     cmp rax, 0x80808080 ; ZF : 0, SF : 0 car extension de signe donne
7     ; cmp 0x00_00_00_00_80_80_80_80, 0xFF_FF_FF_FF_80_80_80_80
8
9     ; à comparer à
10    mov rax, 0x80808080 ; rax : 0x00_00_00_00_80_80_80_80
```

```
11      mov ebx, 0x80808080 ; rbx : 0x00_00_00_00_80_80_80_80
12                          ; mov rbx, 0x80808080 donne idem car pas
13                          ;  d'extension de signe avec mov !
14      cmp rax, rbx        ; ZF : 1, SF : 0 car
15      ; cmp 0x00_00_00_00_80_80_80_80, 0x00_00_00_00_80_80_80_80
```

2 Label

Un *label*[2] (« étiquette » en français) est un repère que le programmeur met dans le code afin de donner un nom à une ligne du programme. Lors du premier TD, nous avons déjà employé un label (`main`) pour signaler la ligne de début du programme.

La définition d'un *label* se termine par le caractère « : », qui ne fait pas partie du nom du *label*. Celui-ci peut contenir des chiffres, des lettres ou le caractère « _ », mais doit commencer par une lettre, un blanc souligné (*underscore*) « _ » ou un point « . [3] ».

3 Branchement

Les instructions d'un programme sont exécutées par le microprocesseur les unes à la suite des autres, dans l'ordre où elles se trouvent en mémoire centrale. Cet ordre est fixé par celui des instructions du code source en langage d'assemblage. On parle d'*exécution séquentielle*. On appelle *saut* (ou branchement [4]) le fait de passer à une instruction autre que celle qui suit en mémoire celle en cours d'exécution.

De tels cas se rencontrent notamment dans les alternatives [5] (*si*, *si-sinon*, *selon que*, etc.) et les boucles [6] (*tant que*, *pour*, *faire jusqu'à ce que*, etc.).

Nous abordons deux types d'instructions de rupture de séquence dans ce TD : les sauts inconditionnels [7] (`jmp`) et les sauts conditionnels [8] `jf` et `jnf`.

3.1 Saut inconditionnel `jmp`

L'instruction `jmp` [9] (aller au label) permet d'effectuer un saut vers un certain label. Ce saut est appelé *inconditionnel* car le saut a lieu dans tous les cas, donc sans aucune

2. https://en.wikipedia.org/wiki/Label_(computer_science) (consulté le 7 février 2020).
3. https://www.nasm.us/doc/nasmdoc3.html#section-3.9 (consulté le 7 février 2020).
4. https://fr.wikipedia.org/wiki/Branchement (consulté le 7 février 2020).
5. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Alternatives (consulté le 7 février 2020).
6. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Boucles (consulté le 7 février 2020).
7. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Sauts_inconditionnels (consulté le 7 février 2020).
8. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Sauts_conditionnels (consulté le 7 février 2020).
9. https://www.felixcloutier.com/x86/jmp (consulté le 7 février 2020).

condition. Cette instruction ne nécessite qu'un seul opérande : le label de la position vers laquelle on veut sauter [10]. Elle ne modifie aucun *flag*.

Voyons cela sur l'extrait de programme suivant :

```
1     mov rax, 123     ; rax <-- 123
2     mov rbx, 150     ; rbx <-- 150
3
4     jmp fin          ; saut inconditionnel vers le label fin
5
6     mov rax, 0       ; ces instructions
7     mov rbx, 0       ; ne sont pas exécutées
8
9   fin:               ; label fin
10    mov rax, 60
11    mov rdi, 0
12    syscall
```

Les instructions situées entre l'instruction `jmp fin` et le label `fin` (lignes 6 et 7) ne sont jamais exécutées [11].

Nous voyons dans la section 4.2 que l'instruction `jmp` est indispensable pour programmer l'alternative *si-sinon* en assembleur.

3.2 Sauts conditionnels `jf` et `jnf`

Les instructions de saut conditionnel [12] permettent d'effectuer un saut si une certaine condition est vraie.

Pour les sauts conditionnels `jf`, cette condition consiste à vérifier si un *flag* donné, `f`, est à 1. La Table 2(a) montre certaines des instructions `jf` [13].

Pour les sauts conditionnels `jnf`, cette condition consiste à vérifier si un *flag* donné, `f`, est à 0. La Table 2(b) montre certaines des instructions `jnf` [14].

3.3 Applications

Les instructions `jf` et `jnf` permettent d'implémenter diverses conditions de saut, à l'aide d'instructions bien choisies. La Table 2(c) en montre quelques exemples, dont l'imparité, l'infériorité et l'inégalité.

10. En mémoire, l'instruction `jmp` est traduite par l'ajout à `rip` d'un déplacement (*offset*) d'un certain nombre d'octets (déplacement relatif, c'est-à-dire signé, codé en complément à deux) qui séparent l'instruction suivante de l'instruction où l'on veut aller.
11. À moins d'y arriver par une instruction située avant l'extrait `;-)`.
12. https://www.felixcloutier.com/x86/jcc (consulté le 7 février 2020).
13. Nous nous limitons ici au *carry flag* (CF), au *zero flag* (ZF) et au *sign flag* (SF), même si des sauts `jf` existent aussi pour d'autres *flags*.
14. Nous nous limitons ici à CF, ZF et SF, mais des sauts `jnf` existent également pour d'autres *flags*.

Instruction	Nom	Effet
`jc label`	*jump carry*	Si `CF` = 1, on effectue le saut vers le label, sinon on passe à l'instruction suivante
`jz label`	*jump zero*	Si `ZF` = 1, on effectue le saut vers le label, sinon on passe à l'instruction suivante
`js label`	*jump sign*	Si `SF` = 1, on effectue le saut vers le label, sinon on passe à l'instruction suivante

(a) Instructions `jc`, `jz` et `js`.

Instruction	Nom	Effet
`jnc label`	*jump not carry*	Si `CF` = 0, on effectue le saut vers le label, sinon on passe à l'instruction suivante
`jnz label`	*jump not zero*	Si `ZF` = 0, on effectue le saut vers le label, sinon on passe à l'instruction suivante
`jns label`	*jump not sign*	Si `SF` = 0, on effectue le saut vers le label, sinon on passe à l'instruction suivante

(b) Instructions `jnc`, `jnz` et `jns`.

Instructions	Effet	Note
`bt rax, 0` `jc label`	Saut si `rax` est impair	`rax` : représentation par position [a] ou complément à 2 [b]
`cmp rax, rbx` `js label`	Saut si `rax` < `rbx`	`rax` et `rbx` : cpt. à 2
`cmp rax, rbx` `jnz label`	Saut si `rax` ≠ `rbx`	`rax` et `rbx` : rep. / position ou cpt. à 2

(c) Exemples d'applications des sauts `jf` et `jnf`.

a. https://fr.wikipedia.org/wiki/Syst%C3%A8me_binaire#D%C3%A9finition (consulté le 7 février 2020).

b. https://en.wikipedia.org/wiki/Two's_complement#Explanation (consulté le 7 février 2020).

TABLE 2 – Instructions `jc`, `jz`, `js`, `jnc`, `jnz` et `jns` et exemples d'applications.

si condition alors	mettre à jour les *flags*
	sauter vers `_fin_si` si condition *fausse*
instructions diverses	instructions diverses
fin si	
`_fin_si:`	
suite du code	suite du code

(a) Schéma de programmation d'un *si* en assembleur.

si `rax` est pair alors	`bt rax, 0`
	`jc _fin_si`
`rbx` ← 5	`mov rbx, 5`
fin si	
`_fin_si:`	
`rcx` ← 12	`mov rcx, 12`

(b) Test de parité.

si `rax` $<$ −10 alors	`cmp rax, -10`
	`jns _fin_si`
`rbx` ← 5	`mov rbx, 5`
fin si	
`_fin_si:`	
`rcx` ← 12	`mov rcx, 12`

(c) Test d'infériorité.

si `rax` ≠ `rdx` alors	`cmp rax, rdx`
	`jz _fin_si`
`rbx` ← 5	`mov rbx, 5`
fin si	
`_fin_si:`	
`rcx` ← 12	`mov rcx, 12`

(d) Test d'inégalité.

TABLE 3 – Exécution conditionnelle sans alternative et exemples.

4 Alternative

4.1 Alternative si

La TABLE 3(a) illustre comment programmer l'alternative *si*[15] en assembleur.

Les TABLES 3(b), 3(c), 3(d) illustrent l'implémentation d'un test de parité, d'infériorité et d'inégalité, respectivement.

15. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Test_si (consulté le 7 février 2020).

si condition alors	mettre à jour les *flags* sauter vers `_sinon` si condition *fausse*
instructions diverses	instructions diverses
	sauter inconditionnellement vers `_fin_si`
sinon	`_sinon:`
autres instructions	autres instructions
finsi	`_fin_si:`
suite du code	suite du code

(a) Schéma de programmation d'un *si-sinon* en assembleur.

si `rax` est pair alors	`bt rax, 0` `jc _sinon`
`rbx` ← 5	`mov rbx, 5`
	`jmp _fin_si`
sinon	`_sinon:`
`rbx` ← 6	`mov rbx, 6`
finsi	`_fin_si:`
`rcx` ← 12	`mov rcx, 12`

(b) Test de parité avec *sinon*.

si `rax` < −10 alors	`cmp rax, -10` `jns _sinon`
`rbx` ← 5	`mov rbx, 5`
	`jmp _fin_si`
sinon	`_sinon:`
`rbx` ← 6	`mov rbx, 6`
finsi	`_fin_si:`
`rcx` ← 12	`mov rcx, 12`

(c) Test d'infériorité avec *sinon*.

si `rax` ≠ `rdx` alors	`cmp rax, rdx` `jz _sinon`
`rbx` ← 5	`mov rbx, 5`
	`jmp _fin_si`
sinon	`_sinon:`
`rbx` ← 6	`mov rbx, 6`
finsi	`_fin_si:`
`rcx` ← 12	`mov rcx, 12`

(d) Test d'inégalité avec *sinon*.

TABLE 4 – Exécution conditionnelle avec alternative et exemples.

4.2 Alternative si-sinon

La Table 4(a) illustre comment programmer l'alternative *si-sinon*[16] en assembleur.
Les Tables 4(b), 4(c), 4(d) illustrent des *si-sinon* implémentant un test de parité, d'infériorité et d'inégalité, respectivement.

5 Exercices

Pour réaliser les exercices qui suivent, vous ne pouvez utiliser que les instructions étudiées au long des TD précédents ainsi que celui-ci. Lorsque le contenu d'un registre est un nombre, considérez que le codage utilisé est la représentation en complément à deux.

Ex. 1 Écrivez un code source complet qui :

1. initialise `rax` à la valeur de votre choix ;
2. met `rbx` à 1 si le contenu de `rax` est non nul.

Par exemple, si le contenu de `rax` vaut −478, le registre `rbx` n'est pas mis à 1.

Ex. 2 Écrivez un code source complet qui :

1. initialise `rax` à la valeur de votre choix ;
2. met `r8` :
   — à 1 si le contenu de `rax` est impair ;
   — à 0 si le contenu de `rax` est pair.

Par exemple, si le contenu de `rax` vaut −478, le registre `r8` est mis à 0.

Ex. 3 Écrivez un code source complet qui :

1. initialise `r14` et `r15` aux valeurs de votre choix ;
2. réalise les traitements suivants :
   — assigne la valeur 0 aux registres `r14` et `r15` si leurs contenus sont égaux ;
   — échange les contenus des registres `r14` et `r15` si ils sont différents.

Par exemple, si initialement les contenus de `r14` et `r15` vallent −478 et 147, respectivement, alors le contenu final de `r14` égale 147 et celui de `r15` est −478.

16. https://fr.wikipedia.org/wiki/Instruction_conditionnelle_(programmation)#Si%E2%80%93alors(%E2%80%93sinon) (consulté le 7 février 2020).

Notez qu'en utilisant `xor`, il est possible de permuter les contenus[17] d'une paire de registres de tailles identiques, sans recourir à un registre tiers. Il est *possible* de le faire. Cela ne vous est pas imposé.

Ex. 4 Écrivez un code source complet qui :

1. initialise `rax` et `rbx` aux valeurs de votre choix ;
2. copie :
   — dans `r8` le maximum des valeurs contenues dans `rax` et `rbx` ;
   — dans `r9` le minimum des valeurs de `rax` et `rbx`.

Par exemple, si `rax` contient la valeur -78 et `rbx` la valeur -15, alors le contenu final de `r8` (*maximum*) est -15 et celui de `r9` (*minimum*) vaut -78.

Ex. 5 Écrivez un code source complet qui :

1. initialise `rdi` à la valeur de votre choix ;
2. met `rsi` :
   — à 0 si le contenu de `rdi` est impair ;
   — à 1 si le contenu de `rdi` est multiple de 2, sans être multiple d'une plus grande puissance de 2 ;
   — à 2 si le contenu de `rdi` est multiple de 4, sans être multiple d'une plus grande puissance de 2 ;
   — à 3 si le contenu de `rdi` est multiple de 8 ou d'une plus grande puissance de 2.

Par exemple :
— si `rdi` vaut 5, `rsi` est mis à 0 ;
— si `rdi` vaut 2, 6, 10, 14, 18 ou 22, `rsi` est mis à 1 ;
— si `rdi` vaut 4, 12 ou 20, `rsi` est mis à 2 ;
— si `rdi` vaut 0, 8, 16 ou 24, `rsi` est mis à 3.

Notions à retenir

Notion de *label* (étiquette), instruction de comparaison arithmétique `cmp`, saut inconditionnel `jmp`, sauts conditionnels `jf`, alternatives *si* et *si-sinon*.

17. https://en.wikipedia.org/wiki/XOR_swap_algorithm (consulté le 7 février 2020).

Références

[1] *Intel© 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, octobre 2017. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[2] Igor Zhirkov. *Low-Level Programming*. Apress, 2017. https://www.apress.com/gp/book/9781484224021.