

# **Programmation Assembleur NASM**

## **Résumé**

## Contents

<b>1 Les bases de programmation en NASM X86</b>	<b>3</b>
1.1 Ce dont vous avez besoin	3
1.1.1 Télécharger le compilateur	3
1.1.2 Compiler sous Linux	3
1.1.3 Compiler sous Windows	3
1.2 Structure d'une programme NASM	4
1.2.1 Une ligne de code NASM	4
1.2.2 Partitionnement du programme	4
1.2.3 Définir un point de départ et sortir du programme (Linux)	4
1.3 Définir des données	5
1.3.1 Les tailles des données	5
1.3.2 Tout est une adresse (ou presque)	5
1.3.3 Les variables initialisés	5
1.3.4 Les variables non-initialisées	6
1.3.5 Les constantes	6
1.4 Les registres	7
1.4.1 Les registres généraux	7
1.4.2 Les registres de contrôle	8
1.4.3 Les registres de segment	8
1.5 Précisions sur les instructions	9
1.5.1 Toujours utiliser au moins UN registre	9
1.5.2 La division entière ( <code>div, idiv</code> )	9
1.5.3 La multiplication entière ( <code>mul, imul</code> )	9
1.5.4 Le fonctionnement de la pile	10
1.6 Les instructions de contrôle de flot	11
1.6.1 Les labels et les sauts	11
1.6.2 Le bloc de boucle <code>label ... loop label</code>	12
1.6.3 Le bloc de procédure <code>label ... ret</code>	12
1.6.4 Le bloc d'exécution conditionnel <code>cmp jcc labelSi</code>	12
1.7 Les fonctions et procédures en NASM	13
1.7.1 L'effet d'une fonction sur la pile	13
1.7.2 Le passage de paramètre sans sauvegarde de pile	13
1.7.3 Le passage de paramètre avec sauvegarde de pile	14

# 1 Les bases de programmation en NASM X86

NASM est un compilateur et un langage assembleur X86 libre et modulaire supportant une grande variété de formats d'objets. Les parties suivantes résument brièvement les bases de la programmation en langage NASM et sont extraites du manuel de NASM (<http://http://www.nasm.us/doc/nasmdoc0.html>) que je vous incite à lire pour plus de détails.

## 1.1 Ce dont vous avez besoin

Pour créer un programme assembleur NASM, il vous faut :

- Le compilateur NASM
- Un éditeur de liens

Le compilateur crée à partir de votre fichier `monProgramme.asm` un fichier objet (`monProgramme.o`) similairement aux compilateurs C et C++. L'éditeur de liens peut ensuite créer un exécutable pour le système voulu (en général le système sur lequel il est exécuté) en interfaçant votre programme avec le système.

### 1.1.1 Télécharger le compilateur

De nombreuses distributions Linux proposent NASM directement dans leur gestionnaire de paquets. Pour les autres systèmes, vous pouvez télécharger le compilateur sur le site <http://www.nasm.us/>, dans la section "download". Après installation ou copie des fichiers, vous pouvez compiler via un terminal (`cmd.exe` pour Windows). Si `nasm` n'est pas trouvé par défaut, vérifiez que votre variable d'environnement `PATH` contient le chemin absolu du dossier contenant l'exécutable `nasm`.

Pour compiler un fichier source `.asm`, NASM s'utilise de la façon suivante :

```
nasm -f <format> monProgramme.asm [-g]
```

L'option `-f` permet de spécifier le format de fichier compilé et `-g` permet de générer les symboles de debug. Le fichier généré peut ensuite être utilisé dans un éditeur de liens.

### 1.1.2 Compiler sous Linux

Pour créer un programme NASM pour les systèmes Linux récents, vous pouvez compiler votre programme au format ELF ("Executable and Linkable Format") et utiliser l'éditeur de liens `ld` :

```
nasm -f elf monProgramme.asm -g  
ld [-m elf_i386] monProgramme.o -o monExecutable
```

Pour exécuter les exemples présent dans ce document (écrits en assembleur X86), il est nécessaire de passer l'option `-m elf_i386` à `ld` si vous travaillez sur un système 64 bits et que vous compilez en format ELF (qui par défaut est en 32 bits). Vous pouvez compiler votre programme au format `elf64`, mais attention : la création d'un programme 64bits (X86\_64) est différente de celle d'un programme 32bits (X86).

Si vous désirez utiliser des fonctions issues du langage C tel que `_printf`, vous devez utiliser `gcc` comme éditeur de liens et respecter certaines conventions d'appels pour le passage de paramètres :

```
gcc monProgramme.o -o monExecutable
```

### 1.1.3 Compiler sous Windows

Pour créer un programme sous Windows, si vous possédez Visual Studio, vous pouvez utiliser le linker fournis par microsoft `link.exe`. Par exemple, pour créer le programme `HelloWorld` à partir du fichier compiler `HelloWorld.obj`, vous pouvez utiliser la ligne de commande suivante sous Windows :

```
nasm -f win32 HelloWorld.asm  
link HelloWorld.obj /OUT:HelloWorld.exe /SUBSYSTEM:CONSOLE /ENTRY:_start
```

Si vous utilisez `minGW` ou `cygwin`, vous pouvez utiliser l'exécutable `ld` au lieu de `link`.

## 1.2 Structure d'un programme NASM

### 1.2.1 Une ligne de code NASM

```
label: instruction  opérandes  ; commentaires
```

Le symbole `;` correspond aux `//` en Langage C, pour signaler le début d'une ligne de commentaires. La présence d'un label (autrement appelé étiquette), est facultative devant une instruction, et sera principalement considéré dans ce document :

- comme début d'un bloc de code (`nomBloc:`)
- comme nom de variable (`nomVariable[:]` `<type>` `<donnée>`).

### 1.2.2 Partitionnement du programme

Un programme assembleur est en général séparé en plusieurs parties (appelées segments). Trois segments sont considérés comme standard en assembleur X86 : `.data`, `.bss` et `.text`.

- Le segment `.data` contient les définitions des variables initialisées à une ou plusieurs valeurs spécifiées (instructions `db`, `dw`, `dd...`).
- Le segment `.bss` contient les définitions des variables non-initialisées, c'est à dire uniquement allouées en mémoire. (instructions `resb`, `resw`, `resd...`).
- Le segment `.text` contient le code exécuté par votre programme.

En plus de ces parties, le programmeur à accès via des instructions `push` et `pop` à une pile mémoire lié au programme (Stack en anglais), souvent utilisée pour le passage de paramètres à des fonctions ou sauvegarder des données de registres avant leur utilisations (section 1.5.4).

### 1.2.3 Définir un point de départ et sortir du programme (Linux)

Le point d'entrée de votre programme ne dépend actuellement pas de NASM, mais de l'éditeur de liens (ou "Linker") utilisé pour transformer le code compilé en exécutable. Pour beaucoup de linker, le point d'entrée par défaut est une fonction `_start` défini comme un symbole `global` : ce symbole sera exporté par votre programme pour être utilisable par le système ou par d'autres fichiers compilés lors de l'édition.

Si une fonction (ou une variable) identifiée par un label est également déclaré comme global dans le fichier où elle est définie, cette fonction (ou variable) peut être appelée dans un autre fichier, à la condition que son label soit également déclaré en `extern` dans le fichier voulant l'utiliser.

La programme suivant présente la structure élémentaire d'un programme NASM, dans laquelle sont incluses les instructions pour quitter un programme proprement :

```
global _start ; déclaration de _start en global
                ; => export du point d'entrée pour créer le programme
segment .data
    ; création de variables initialisées
segment .bss
    ; création de variables non-initialisées
segment .text
    ; création de vos procédures/fonctions entre le segment et le point d'entrée
_start:
    ; instructions de votre programme
    ; Les 3 lignes suivantes doivent être à la fin de votre _start
    mov eax,1    ; signalement de fin de programme
    mov ebx,0    ; code de sortie du programme
    int 0x80     ; interruption Linux : votre programme rend la main au système
```

## 1.3 Définir des données

### 1.3.1 Les tailles des données

NASM considère les types de donnée suivants :

- le Byte ( 1 octet = 8 bits)
- le Word ( 2 octet = 16 bits)
- le Double word ( 2 word = 4 octets = 32 bits)
- le Quad word ( 4 word = 8 octets = 64 bits)
- et le exTended word ( type spécial sur 80 octets pour les réels)

Les lettres majuscules de cette liste sont utilisées pour définir le type de la variable déclarée. Cette lettre est associée à un préfixe pour préciser au système si la variable doit être initialisée (d pour "define") ou juste allouée en mémoire (res pour "reserve").

### 1.3.2 Tout est une adresse (ou presque)

Une variable NASM se comporte comme un pointeur en Langage C : la variable est une adresse mémoire qui pointe vers un tableau de un ou plusieurs éléments. Chaque donnée est accessible par déréréférencement du pointeur ou d'une adresse obtenue par décalage depuis celui-ci.

Plus généralement, pour accéder à la valeur d'un emplacement mémoire pointé par une adresse, on utilise l'opérateur de déréréférencement [ ]. Par exemple, pour accéder au contenu de la variable/adresse var, on utilisera la notation [var] en NASM, équivalente à la notation (\*var) en langage C et C++.

**Toute élément déclaré par le programmeur en assembleur est en règle générale une adresse, sauf exceptions des constantes.**

### 1.3.3 Les variables initialisés

Lors de la déclaration d'une variable initialisée, l'opérande de droite est la valeur d'initialisation, c'est à dire le contenu de la variable :

```
var1 db 123 ; "Define Byte" [var1] = 123, sizeof([var1]) = 8 bits
var2 dw 456 ; "Define Word" [var2] = 456, sizeof([var2]) = 16 bits
var3 dd 789 ; "Define Double" [var3] = 789, sizeof([var3]) = 32 bits
var4 dq 123 ; "Define Quad" [var4] = 123, sizeof([var4]) = 64 bits
var5 dt 1.23; "Define exTended word" [var4] = 1.23, sizeof([var5]) = 80 bits
```

Une variable est une adresse qui référence un tableau de un élément, ou plus si plusieurs valeurs (séparées d'une virgule) ou une chaîne de caractères sont utilisées comme opérandes. L'instruction times n peut être utilisée entre le nom de la variable et l'instruction pour répéter la définition d'une valeur particulière dans la variable :

```
var1 db 123,12,3 ; [var1] = 123, [var1+1] = 12, [var1+2] = 3
var2 db 'Hi',0xa ; [var2] = 'H', [var2+1] = 'i', [var2+2] = '\n' = 0xa
var3 dd 255,6554 ; [var3] = 255, [var3+1] = 6554
var4 times 4 db '*' ; var4 = "****"
```

Les 3 premières lignes peuvent se traduire en langage C par :

```
char var1[3] = {123,12,3};
char var2[3] = "Hi\n";
int var3[2] = {255,6554};
```

### 1.3.4 Les variables non-initialisées

Les instructions de déclarations de variables non-initialisées allouent un tableau dont le nombre d'élément est définis par l'opérande de droite :

```
var1 resb 1024 ; "Reserve 1024 Bytes"  
var2 resw 1 ; "Reserve 1 Word"  
var3 resd 12 ; "Reserve 12 Double"  
var3 resq 24 ; "Reserve 24 Quad"  
var4 rest 2; "Reserve 2 exTended word"
```

Dans cet exemple, var1 est l'adresse du premier élément d'un tableau de 1024 octets.

**ATTENTION, la mémoire est alloué mais non-initialisé : l'emplacement alloué peut contenir des valeurs complètement aléatoires.**

### 1.3.5 Les constantes

Pour définir une constante, on utilise l'instruction `equ` (pour "equal") précédée d'un label et suivi de la valeur de la constante :

```
myConstante equ 12345
```

**ATTENTION, une constante n'est pas une variable et, similairement aux registres, n'a pas d'adresse à proprement parler : une constante est directement interprétée comme une valeur.**

## 1.4 Les registres

Le processeur ne peut pas directement réaliser une instruction entre deux variables : Les sorties de ses unités de calcul ne sont pas directement connecté à la mémoire centrale dans laquelle les variables sont normalement stockés. Il doit alors utiliser ses propres emplacements mémoires avant de demander le transfère du résultat d'une instruction dans votre mémoire centrale. Ces emplacements mémoire, appelés "registres", contiennent des informations sur l'état courant du processeur et sont utilisés par celui-ci pour réaliser des instructions et sauvegarder le résultat. Pour accéder au contenu des registres, aucun déréférencement n'est nécessaire : le nom d'un registre sert d'alias permettant de récupérer ou modifier la valeur qu'il contient (similairement au fonctionnement des constantes). Par exemple, pour transférer le contenu du registre EAX dans une variable `var`, on peut simplement faire l'instruction :

```
mov [var], eax
```

En assembleur, différents types de registres processeur sont utilisables. : les registres généraux, les registres de contrôle et les registres de segment.

### 1.4.1 Les registres généraux

Pour la plupart des registres généraux, il est possible d'accéder aux registres en mode 32bits (préfixe E) ou 16bits. Si vous utilisez un système 64 bits et que vous compilez en `elf64`, ces registres ont également un mode 64bits (préfixe R). Pour les 4 registres de stockage principaux (AX, BX, CX et DX) il est en plus possible d'accéder directement aux parties hautes et basses des 16 bits de poids faibles (les 8 bits de poids fort et les 8 bits de poids faible).

La liste suivante présente les alias 64, 32 et 16 bits des registres généraux ainsi que leur utilité par défaut pour le processeur. les préfixe R et E et les alias [octet fort:octet faible] des 2 octets de poids faibles sont signalés seulement si ces modes d'accès sont disponibles pour le registre en question :

- (R,E)AX - [AH:AL] : accumulateur, stockage du retour d'une fonction ou d'un appel système.
- (R,E)BX - [BH:BL] : décalage dans le segment des données pointé par le registre DS.
- (R,E)CX - [CH:CL] : compteur de boucle (instruction `loop`).
- (R,E)DX - [DH:DL] : registre de données, utilisé lors des opérations d'entrées/sorties et pour certains calculs longs (instructions `div` et `mul`).
- (R,E)SI : Source Index : pointeur "source" pour les opérations sur des chaînes de caractères.
- (R,E)DI : Destination Index : pointeur "destination" pour les opérations sur des chaînes de caractères.
- (R,E)BP : Base Pointer : Pointeur du début de la pile mémoire du programme.
- (R,E)SP : Stack Pointer : Pointeur de la position actuelle de la pile.

En plus de leur utilité listée ci-dessus, les 4 registres `**X` sont également les registres de stockage à utiliser pour les opérations arithmétiques. Les registres de type "Pointer" ne contiennent pas une adresse "absolue" en mémoire, mais une adresse relative à un `segment` du programme. Il est cependant possible de recalculer une telle adresse en utilisant les registres dit de "segments", qui conservent l'adresse de départ de chaque segment du programme.

### 1.4.2 Les registres de contrôle

Pour contrôler l'ordonnancement des instructions lors de l'exécution, deux registres sont utilisés par le processeur :

- (E)IP : Instruction Pointer : Registre stockant l'adresse de la prochaine instruction à exécuter (relatif au segment de code).
- (E)FLAGS : registre contenant des valeurs binaires (flag) utilisées dans certaines instructions.

Ce dernier registre est découpé en plusieurs bits de contrôle situés aux positions suivantes dans le registre (du poids faible au poids fort) :

- bit 0 : CF (Carry flag) : si une opération arithmétique ou de décalage génère une retenue, ce bit correspond à la valeur de la retenue
- bit 2 : PF (Parity flag) : nombre de bits à 1 dans le résultats d'une opération arithmétique (1 si nombre de bit impair).
- bit 4 : AF (Auxiliary Carry Flag) : contient la retenue du bit 3 au bit 4 spécifiquement après une opération arithmétique sur un octet.
- bit 6 : ZF (Zero Flag) : après une opération arithmétique ou de comparaison, ce bit est à 1 si le résultat est égale à 0.
- bit 7 : SF (Sign Flag) : Donne le signe d'une opération arithmétique (0 pour positif, 1 pour négatif).
- bit 8 : TF (Trap Flag) : permet le passage du processeur en mode pas-à-pas (mode debug).
- bit 9 : IF (Interrupt Flag) : si ce bit est à 0, tout signal d'interruption envoyer par une entrée (clavier ou autre) est ignoré.
- bit 10 : DF (Direction Flag) : direction du curseur pour les opération sur les chaînes (si 0, de gauche à droite, sinon de droite à gauche).
- bit 11 : OF (Overflow Flag) : indique si une erreur de dépassement de capacité à eu lieu après une opération arithmétique signée.

### 1.4.3 Les registres de segment

Certains registres présentés dans les sections précédentes servent à conserver des décalages mémoire (adresses relatives) par rapport à des adresses conservées dans d'autres registres. Ces derniers sont les registres de segment, dont voici la liste et leur utilité :

- CS : Code Segment : ce registre contient l'adresse de départ du segment de code (plus ou moins la partie .text du programme), contenant toutes les instructions à exécuter.
- DS : Data Segment : ce registre contient l'adresse de départ du segment contenant les données et les constantes (parties .bss et .data).
- SS : Stack Segment : ce registre contient l'adresse de départ de la pile du programme (qui contient des données et également les adresses de retour des fonctions).
- ES, FS et GS sont des registres de segment supplémentaires (ES peut par exemple être utiliser avec EDI (ES:EDI) pour manipuler des adresses situées plus loin dans l'espace mémoire).



## 1.5 Précisions sur les instructions

Cette partie a pour but d'apporter des précisions supplémentaires sur certaines instructions présentées dans la quickcard x86 pour NASM proposés par l'Université de Nantes.

### 1.5.1 Toujours utiliser au moins UN registre

La majorité des instructions de votre processeur stocke le résultat dans la première opérande (i.e. l'opération `add EAX,2` stocke le résultat de l'opération dans EAX) ou directement dans un registre précis. Pour des opérations arithmétiques, le processeur ne peut pas directement lire et écrire un même emplacement mémoire : il peut déplacer des données depuis celui-ci ou vers celui-ci (l'instruction `mov`) respectivement vers ou depuis un registre, mais il ne peut pas directement exécuter un calcul sur l'emplacement. Par extension, lorsque plusieurs opérandes sont requises dans une opération arithmétiques, la première opérande doit être un registre (de préférence un registre de stockage `**X`) :

```
add [var1], [var2] ; erreur
mov eax, [var1]
add eax, [var2]
mov [var1], eax ; [var1] = [var1] + [var2]
```

### 1.5.2 La division entière (`div`, `idiv`)

Les instructions de divisions entières signées (`idiv`) et non-signées (`div`) divisent le contenu de EAX par une opérande passé en paramètre de l'instruction. La taille de l'opérande (le diviseur) va cependant définir la taille et le nombre de registre utilisés pour l'opération et la sauvegarde du résultat :

- Si le diviseur est déclaré sur 8 bits, l'opération prendra le registre AX comme dividende. Le résultat de la division sera conservé dans AL, et son reste dans AH
- Si le diviseur est déclaré sur 16 bits, le double registre DX:AX (DX pour les bits de poids forts et AX pour ceux de poids faibles) sera utilisé comme dividende. Le résultat de la division sera conservé dans AX, et le reste dans DX.
- Si le diviseur est déclaré sur 32 bits, le double registre EDX:EAX sera utilisé comme dividende. Le résultat de la division sera conservé dans EAX, et le reste dans EDX.

La division utilisant EDX pour des tailles de données supérieur a 8 bits, l'oubli d'une remise à zéro de EDX avant l'opération peut mener à un résultat faux. Il est possible de préciser directement la taille de la division grâce à un spécificateur de type (`dbyte`, `dword`, `ddouble`) :

```
a: dd 12      ; (dans .data) a sur 32 bits
mov eax, 15   ; initialisation du dividende sur 32 bits
div byte [a]  ; spécificateur de taille byte :
               ; divise ax par le contenu de a transformé (cast) en octet
```

### 1.5.3 La multiplication entière (`mul`, `imul`)

Si une seule opérande est passée en paramètre de l'instruction `mul` ou `imul`, cette instruction multiplie cette opérande par :

- AL si l'opérande est de type byte (ou si `byte` est spécifié devant l'opérande)
- AX si l'opérande est de type word (ou si `word` est spécifié devant l'opérande)
- EAX si l'opérande est de type double (ou si `dword` est spécifié devant l'opérande)

Le résultat est alors respectivement stocké dans AX, le double registre DX:AX ou le double registre EDX:EAX.

#### 1.5.4 Le fonctionnement de la pile

Comme dit dans la section 1.2.2, chaque programme assembleur a accès à une structure de pile mémoire, utilisée par certaines instructions pour stocker des données. Mais vous pouvez également ajouter et enlever des données de cette pile respectivement grâce aux instructions `push` and `pop`. La pile peut être utilisée, par exemple, pour sauvegarder la valeur d'un registre avant d'utiliser ce dernier :

```
        ; Etat pile début : ESP = EBP (le haut et le bas ont la même adresse)
push eax ; sauvegarde de la valeur de eax en haut de la pile
        ; Pile : [ESP] = valeur de eax (4 octets), ESP+4 = EBP
push ebx ; sauvegarde de la valeur de ebx en haut de la pile
        ; Pile : [ESP] = valeur de ebx, [ESP+4] = valeur de eax
;
; instructions utilisant eax et ebx
;
pop ebx  ; dépile [ESP] dans ebx et décale ESP,
        ; Etat pile : [ESP] = valeur de eax (4 octets)
pop eax  ; dépile [ESP] dans eax et décale ESP
        ; Etat pile : ESP = EBP (le haut et le bas ont la même adresse)
```

Lorsque vous remplissez la pile, vous pouvez directement accéder au contenu de celle-ci en utilisant les registres ESP et EBP. Ces registres contiennent respectivement les adresses du haut de la pile et du bas de la pile. Dans l'exemple précédent, après les 2 instructions de `push`, il est possible de récupérer respectivement les valeurs de `eax` et `ebx` stockées dans la pile grâce à ESP :

```
        ; Etat pile début : ESP = EBP
push eax      ; sauvegarde de la valeur de eax en haut de la pile
              ; Pile : [ESP] = valeur de eax (4 octets), ESP+4 = EBP
push ebx      ; sauvegarde de la valeur de ebx en haut de la pile
              ; Pile : [ESP] = valeur de ebx, [ESP+4] = valeur de eax
mov ebx,[esp+4] ; copie dans EBX de la valeur de EAX
mov eax,[esp]   ; copie dans EAX de l'ancienne valeur de EBX
mov ecx,[esp+4] ; copie dans ECX de l'ancienne valeur de EAX
```

Notez que le décalage par rapport à ESP (et également EBP) descend dans la pile.

## 1.6 Les instructions de contrôle de flot

### 1.6.1 Les labels et les sauts

Pour contrôler le flot d'instructions, un label peut être utilisé pour identifier le début d'un bloc d'instruction : la plupart des instructions de contrôle ont besoin d'un point de repère (un label ou une adresse) pour se déplacer dans les instructions à l'aide de saut. Il existe en assembleur 2 type de sauts : le saut inconditionnel `jmp <adresse ou label>` et les sauts conditionnelles. Voici un exemple de saut inconditionnel.

```
unLabel:
    ; instructions
    jmp unLabel ; retourne à la position identifiée par "unLabel"
```

Les sauts conditionnelles dépendent du résultat de l'instruction `cmp val1, val2` : cette instruction réalise l'opération `val1 - val2` et modifie le registre EFLAGS en conséquence (notamment les bits ZF, SF et OF). Ces instructions de sauts sont nombreuses mais sont toujours de la forme `j<cc> label`, où `<cc>` est la condition du test et `label` est la position à laquelle le flot d'instructions doit reprendre.

Ce saut conditionnel peut être "non-signé" :

- `je` : jump if val1 is equal to val2 ( $|val1| = |val2|$ )
- `ja` : jump if val1 is above val2 ( $|val1| > |val2|$ )
- `jae` : jump if val1 is above or equal to val2 ( $|val1| \geq |val2|$ )
- `jb` : jump if val1 is below val2 ( $|val1| < |val2|$ )
- `jbe` : jump if val1 is below or equal to val2 ( $|val1| \leq |val2|$ )

Ou "signé":

- `jg` : jump if val1 is greater than val2 ( $val1 > val2$ )
- `jge` : jump if val1 is greater than or equal val2 ( $val1 \geq val2$ )
- `jl` : jump if val1 is lower than val2 ( $val1 < val2$ )
- `jle` : jump if val1 is lower than or equal val2 ( $val1 \leq val2$ )

Il existe également des versions prévues pour tester spécifiquement certains bits de EFLAGS :

- `jc` : jump if CF = 1 (si une retenue a été générée)
- `jo` : jump if OF = 1 (s'il y a eu overflow)
- `jz` : jump if ZF = 1 (si le dernier calcul a renvoyé 0)
- `js` : jump if SF = 1 (si le résultat du dernier calcul est négatif)

Pour chacune de ces instructions, il existe un version inverse `j<cc>` (par exemple `jne` pour "jump if not equal").

```
unLabel:
    mov cl, [byte1]; contenu de byte1 (1 octet) dans CL
    cmp cl, [byte2]; comparaison de [byte2] et [byte1]
    jnz unLabel    ; si cl - [byte2] ne renvoi pas 0, on saute à "unLabel"
```

### 1.6.2 Le bloc de boucle `label ... loop label`

Pour créer une boucle similaire à une boucle `for`, il est possible d'utiliser l'instruction `loop` associée à un label définissant le début des instructions à répéter. Lorsque le programme atteint cette instruction, il décrémente la valeur du registre `ECX`, et si cette valeur est différente de 0, le programme retourne au label donnée en paramètre de l'instruction :

```
mov ecx, 5 ; pour faire 5 itérations
ma_boucle:
    ; mes instructions
    loop ma_boucle ; ecx = ecx - 1, si ecx != 0, alors jmp ma_boucle
```

### 1.6.3 Le bloc de procédure `label ... ret`

En NASM, une procédure ou une fonction est simplement un ensemble d'instructions commençant par un label et se terminant par l'instruction `ret` :

```
ma_procedure:
    ; instructions sans push ni pop
    ret ; équivalent à pop eip, ce qui à pour effet dans ce cas à jmp apresLeCall
; Dans le _start
call ma_procedure ; équivalente à push eip et jmp ma_procedure
apresLeCall : ; suite du programme
```

Pour appeler cette fonction/procédure, il suffit d'appeler l'instruction `call ma_procedure`.

**ATTENTION : si vous modifiez la pile, vous risquez de perdre le pointeur stocké par le `call`.**  
Plus de détails sur les fonctions sont fournis dans la section 1.7.

### 1.6.4 Le bloc d'exécution conditionnel `cmp jcc labelSi`

En utilisant le système de label et de saut conditionnel, il est possible de reproduire le comportement d'un `if{ } else { }` en utilisant des sauts. Voici un exemple illustrant la structure d'un tel bloc en utilisant `EAX` comme registre de calcul :

```
mov eax, [var1]
cmp eax, [var2] ; calcul de la comparaison [var1] - [var2]
j<cc> si_condition_vrai ; si [var1] <cc> [var2] est vrai, aller a si_condition_vrai
    ; instructions sinon
jmp fin_si ; fin du sinon, aller à la fin du bloc conditionnel
si_condition_vrai:
    ; instruction si [var1] <cc> [var2] est vrai
fin_si:
    ; fin du bloc
```

La liste des instructions de sauts conditionnels (`j<cc>`) et des tests qui leur correspondent sont données dans la section 1.6.1.

## 1.7 Les fonctions et procédures en NASM

### 1.7.1 L'effet d'une fonction sur la pile

L'instruction `call label` utilise la pile du programme pour sauvegarder le pointeur vers la prochaine instruction (stocké dans EIP) et réalise un saut inconditionnelle vers le label définissant le début du bloc de la procédure. L'instruction `ret` est équivalente à un `pop eip` : le dernier élément de la pile est transféré dans le pointeur vers la prochaine instruction à exécuter, ce qui aura un effet similaire à un saut vers l'adresse anciennement contenu dans EIP si aucune modification de la pile n'a eu lieu :

```
ma_fonction:
    ; Etat de la pile : [ESP] = adresse de prochaine_instruction
    ; instructions
    ; Si pas de modification de la pile, [ESP] = adresse de prochaine_instruction
    ret ; = pop EIP => changement du pointeur de la prochaine instruction

_start:
    ; instructions
    ; Etat pile : ESP = EBP
    call ma_fonction ; push EIP ([ESP] = prochaine_instruction) et jmp ma_fonction
    ; au retour de la fonction : ESP = EBP
    prochaine_instruction
```

Si la pile n'a pas été modifié dans la fonction, le pointeur positionné en haut de la pile par le `call` correspond à l'instruction suivant ce `call` dans le code. Pour que cette instruction fonctionne, en cas d'utilisation de la pile dans la fonction, un nettoyage de la pile est nécessaire (un `pop` pour chaque `push`).

### 1.7.2 Le passage de paramètre sans sauvegarde de pile

Le nombre de registre de stockage étant limité, il est courant d'utiliser la pile en assembleur pour passer des paramètres à une fonction. Voici un exemple :

```
ma_fonction:
    ; Etat supposé de la pile (du haut vers le bas de la pile) :
    ; ancien EIP (32 bits = 4 octets), une valeur 32 bits, une adresse 32 bits
    mov ecx, [esp+8] ; accès à l'adresse stocké dans la pile
    mov edx, [esp+4] ; accès à la valeur stocké dans la pile
    mov eax, 4
    mov ebx, 1
    int 0x80
    ret ; pop dans EIP
    ; autre version (pour que la fonction nettoie la pile)
    ; ret 8 => pop EIP et dépile N (ici 8) octets (2*32 bits)

_start:
    ; Au début : ESP = EBP
    push msg ; variable contenant une chaine de caractères
    ; [ESP] = msg
    push len ; (constante égale à la taille de msg)
    ; [ESP] = len, [ESP+4] = msg
    call ma_fonction ; [ESP] = EIP, [ESP+4] = len, [ESP+8] = msg
    ; après le ret : [ESP] = len, [ESP+4] = msg
    ; après le ret N : ESP = EBP
    ; nettoyage de la pile si non fait dans la fonction avec ret N
    pop eax ; dépile dans eax pour nettoyage
    pop eax
```

Le type de retour `ret` utiliser est en réalité dépendant du type de convention : certaines conventions réclament au module appelant la fonction de nettoyer lui même la pile (`pop` des arguments après le `call`, alors que d'autres spécifient que la fonction appelée est responsable du nettoyage des arguments situés dans la pile (`ret N` avec N égale à la taille en octets des paramètres).

Cette approche est pratique pour des fonctions très simples et rapide, mais la gestion de la pile devient plus complexe si une autre fonction est appelé dans la fonction.

### 1.7.3 Le passage de paramètre avec sauvegarde de pile

Une manière plus souple d'utiliser la pile consiste à créer une pile locale dans la fonction, et d'utiliser le pointeur de bas de pile EBP pour récupérer les valeurs de l'ancienne pile. En effet, ce pointeur n'est pas modifié par les instructions push et pop, ce qui permet de revenir facilement à l'ancien état de la pile comme dans l'exemple suivant :

```
ma_fonction:
    ; Etat supposé de la pile (du haut vers le bas de la pile) :
    ; ancien EIP (32 bits = 4 octets), une valeur 32 bits, une adresse 32 bits
    ; nouveau cadre de pile
    push ebp      ; Sauvegarde de l'ancien bas de pile
    mov ebp, esp ; on remonte le bas de pile à la position du haut de la pile
    ; EBP = ESP => nouveau cadre de pile
    ; Pour accéder au ancien paramètre, on peut se se décaler par rapport à EBP
    ; [EBP] = ancien EBP, [EBP+4] = EIP, [EBP+8] = len, [EBP+12] = msg
    mov ecx, [ebp+12] ; accès à l'adresse stocké dans la pile
    mov edx, [ebp+8] ; accès à la valeur stocké dans la pile
    mov eax, 4
    mov ebx, 1
    int 0x80
    leave ; = mov esp, ebp et pop ebp => la pile reprend son ancien état
    ret   ; pop dans EIP
    ; autre version (pour que la fonction nettoie la pile)
    ; ret 8 => pop EIP et dépile N (ici 8) octets (2*32 bits)
_start:
    ; Au début : ESP = EBP
    push msg      ; variable contenant une chaine de caractères
    ; [ESP] = msg
    push len      ; (constante égale à la taille de msg)
    ; [ESP] = len, [ESP+4] = msg
    call ma_fonction ; [ESP] = EIP, [ESP+4] = len, [ESP+8] = msg
    ; après le ret : [ESP] = len, [ESP+4] = msg
    ; après le ret N : ESP = EBP
    ; nettoyage de la pile si non fait dans la fonction avec ret N
    pop eax      ; dépile dans eax pour nettoyage
    pop eax
```