

# RASM

roudoudou Assembler  
v0.64

Ce logiciel et sa documentation utilisent la licence MIT "expat"

« Copyright © BERGÉ Édouard (roudoudou)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation/source files of RASM, to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The Software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the Software. »

## Sommaire

Introduction.....	5
Installation.....	5
Compilation.....	6
Comportement de Rasm.....	7
Utilisation de la ligne de commande.....	8
Options de fichier.....	9
Options de symboles.....	9
Options de compatibilité.....	11
Options développeur.....	12
Format du code source.....	13
Généralités.....	13
Commentaires.....	13
Valeurs littérales.....	14
Caractères autorisés.....	14
Fichiers.....	15
Spécificités sur les labels.....	15
Expressions.....	16
Opérateurs de calcul.....	16
Opérateurs de comparaison.....	16
Spécificités sur les expressions.....	17
Variables statiques ou alias.....	18
Variables dynamiques.....	18
Directives.....	19
ASSERT <condition>.....	20
EQU.....	21
ALIGN <valeur>.....	21
AMSDOS.....	21
BUILDCPR.....	21
BUILDSNA.....	21
BANK [<numéro de page ROM, numéro de page RAM>].....	22
BANKSET <numéro de bloc de 64K>.....	22
IF, IFNOT, ELSE, ELSEIF, ENDIF.....	23
LZ48 / LZ49 / LZ4 / LZX7 / LZEXO.....	24
LZCLOSE.....	24
READ / INCLUDE 'fichier à lire'.....	25
INCBIN 'fichier à lire',[offset[,size[,offset étendu[,OFF]]]].....	25
INCL48 / INCL49 / INCLZ4 / INCZX7 / INCEXO 'fichier à lire'.....	26
LIMIT <adresse limite>.....	26
ORG <adresse logique>[,<adresse d'écriture du code>].....	27
PROTECT <adresse début>,<adresse fin>.....	28
STR '<chaîne>','<chaîne2>',.....	28
STOP.....	28
PRINT '<string>',variables,expression,.....	29
RUN <valeur>.....	29
LIST / NOLIST / LET / BRK.....	29

WHILE / WEND.....	30
REPEAT <n> / REND   REPEAT / UNTIL <condition> .....	30
WRITE DIRECT <rom basse>,<rom haute>,<RAM>.....	31
SAVE 'fichier binaire à écrire',<adresse>,<taille>.....	32
CHARSET 'chaîne',<valeur>   <code>,<valeur>   <début>,<fin>,<valeur> .....	32
SWITCH, CASE, BREAK, DEFAULT, ENDSWITCH.....	33
Macros.....	34
Instructions.....	35
Limitations.....	35
Exemples de programmation avec RASM.....	36
Récupérer le numéro de bank d'un label pour connecter une ROM.....	36

## Introduction

Il y a 18 ans, j'avais programmé un assembleur/désassembleur appelé Zasm/Dizasm. En réalité c'était surtout une démonstration qu'il était possible de faire un assembleur mono-passe. J'ai un peu utilisé le désassembleur sur des projets personnels. Les sources ont été publiées mais la diffusion a été si confidentielle que j'ai eu du mal à les récupérer à nouveau sur mon disque dur.

Depuis tout ce temps, un autre assembleur du nom de Zasm est sorti, créé par une équipe qui développe pour le ZX spectrum, un autre ordinateur équipé du Z80.

Vu que tout le code de cet assembleur est nouveau (à part quelques concepts), c'est l'occasion de le baptiser avec un nouveau nom: Rasm.

## Installation

Rasm est un exécutable indépendant, il s'utilise tel que, sans installation aucune.

## Compilation

### compilation Linux:

```
cc rasm_v063.c -O2 -lm -lrt -march=native  
mv a.out rasm  
strip rasm
```

### compilation Windows:

```
cl.exe rasm_v063.c -O2
```

## Comportement de Rasm

Rasm essaie d'être simple d'utilisation. En ce sens il va produire des noms de fichier par défaut, déterminer la quantité de mémoire à enregistrer ou même créer un fichier cartouche si les banques ROM ont été sélectionnées lors de l'assemblage.

Rasm permet l'utilisation de plusieurs ORG au sein d'un même espace mémoire. Par contre il n'autorise pas de ré-écrire sur les mêmes adresses mémoire. À chaque nouveau ORG, Rasm contrôle qu'aucune zone de code écrite ne se chevauche.

Si vous avez besoin de générer plusieurs morceaux de code à la même adresse, vous avez deux possibilités. Soit vous utilisez le paramètre <output> de la fonction ORG pour écrire ce code ailleurs, soit vous pouvez créer à tout moment un nouvel espace mémoire avec la fonction WRITE DIRECT -1,-1,#C0

La sauvegarde forcée d'espaces mémoire avec la fonction SAVE désactive l'écriture automatique de fichier ou cartouche. Il reste possible de forcer l'écriture cartouche avec la directive BUILD CPR.

Dans la mesure du possible, Rasm essaie d'afficher des messages d'erreurs afin de vous orienter vers la solution syntaxique convenable.

Rasm va d'abord pré-traiter le fichier à assembler pour enlever les espaces superflus, les commentaires, vérifier les quotes, que les caractères utilisés soient conformes et enfin transformer certaines instructions en d'autres pour convenance interne. Par exemple les opérateurs maxam XOR,AND,OR,MOD seront convertis en un seul caractère approchant la syntaxe du C.

Lorsqu'une directive de lecture ne fait pas référence à un chemin absolu, le répertoire racine au chemin relatif est celui du fichier en cours.

## Utilisation de la ligne de commande

La syntaxe de base est: `rasm.exe <fichier à assembler> [options]`

Exemple: `rasm.exe monfichier.asm -s`



## ***Options de fichier***

-o <radix de fichier> définir un nom commun pour chaque type de fichier en sortie, quel que soit son type (.bin, .cpr, .sym). La valeur par défaut est “rasmoutput”.

`rasm.exe source.asm -o output -s` produira les noms suivants selon le type: `output.bin`, `output.cpr`, `output.sym`

-ob <nom de fichier binaire> définir le nom complet pour le fichier de sortie binaire.

-oc <nom de fichier cartouche> définir le nom complet pour le fichier de sortie cartouche.

-oi <nom de fichier snapshot> définir le nom complet pour le fichier de sortie snapshot.

-os <nom de fichier symbole> définir le nom complet pour le fichier de sortie des symboles.

## ***Options de symboles***

-s exporter les symboles au format Rasm

-sp exporter les symboles au format Pasm

-sw exporter les symboles au format Winape

-ss exporter les symboles dans un snapshot

-sl option additionnelle aux trois précédentes qui permet d'exporter aussi les symboles locaux aux macros ou boucles de répétition.

-sv option additionnelle aux trois précédentes qui permet d'exporter aussi les variables.

-sq option additionnelle aux trois précédentes qui permet d'exporter aussi les alias EQU.

-sa option équivalente à -sl -sv -sq

– L'émulateur Arnold est compatible avec Rasm et Pasm.

– L'émulateur Winape n'est pas capable de prendre en charge des labels trop longs!

-l <fichier label> importer un fichier de labels pour l'assemblage au format Rasm, Pasm ou Winape



## ***Options de compatibilité***

-m    mode de compatibilité avec maxam

L'usage de cette option change le comportement des comparaisons et des calculs internes. Tous les calculs sur les variables seront réalisés sur des entiers 16 bits non signés. La comparaison se fera avec l'opérateur = simple.

## ***Options développeur***

- v mode verbeux, affiche des informations et statistiques sur l'assemblage
- d produit des informations lors du pré-processing
- a produit des informations lors de l'assemblage
- n affiche les licences tierces

## **Format du code source**

### ***Généralités***

L'assembleur n'est pas du COBOL, il est inutile d'indenter vos sources avec Rasm, autrement que pour faire joli. Il n'est pas nécessaire de séparer un label d'une instruction ou d'un autre label par le caractère deux points, bien qu'il soit possible de le faire. La conséquence directe de cette écriture libre implique une différence avec les assembleurs de conception obsolète. Il n'est pas possible (et heureusement) de créer un label qui ait le même nom qu'une directive ou instruction Z80.

Le défaut de cette liberté est que l'assembleur ne peut pas savoir si vous voulez déclarer un label quand vous vous trompez dans l'écriture du nom d'une macro sans paramètre. Pour palier à ce défaut, vous pouvez ajouter un paramètre fictif “(void)” qui déclenchera une erreur si le nom de macro n'est pas connu.

Les fichiers peuvent être lus au format windows ou unix, une conversion interne et transparente sera réalisée en interne.

Rasm n'est pas sensible à la casse, toutes les lettres sont converties en majuscules en interne. Ne soyez pas surpris de ne voir que des majuscules dans les messages d'erreur.

### ***Commentaires***

La saisie de commentaire sous Rasm est classique et précédée du caractère point-virgule. Tous les caractères suivants sont ignorés jusqu'au prochain retour chariot.

Il n'y a pas de commentaire multi-lignes

## ***Valeurs littérales***

Rasm interprète les valeurs numériques suivantes:

- En décimal si la valeur commence par un chiffre.
- En binaire si la valeur commence par un %.
- En hexadécimal si la valeur commence par un # ou 0x.
- En valeur ascii si un caractère unique est entre quote.
- En valeur “interne” à une variable ou label si la littérale commence par une lettre ou un '@' pour les labels locaux.

Rasm fait tous ses calculs internes en nombre flottant double précision. Un arrondi correct est réalisé en fin de chaîne de calcul pour les besoins en nombres entiers.

Attention, le caractère & est réservé pour l'opérateur AND.

## ***Caractères autorisés***

Entre quotes, tous les caractères sont autorisés, à vos risques et périls concernant la conversion ASCII vers l'Amstrad. En dehors des quotes, vous pourrez utiliser toutes les lettres, tous les chiffres, le point, l'arobas, les parenthèses, le dollar, les opérateurs plus, moins, multiplié, divisé, le pipe, circonflex, le pourcent, le dièse, le paragraphe, les chevrons et les deux types de quotes.

## ***Fichiers***

RASM utilise en interne le modèle de fichiers Unix qu'il converti automatiquement au besoin. En utilisant uniquement des chemins relatifs (et des noms de fichiers que Windows peut comprendre), on peut tout à fait avoir un code qui compile à la fois sous Unix et Windows sans faire de modification.

La règle de gestion des chemins relatifs est simple. Tout chemin relatif a pour origine le répertoire du fichier dans lequel il a été lu.

## ***Spécificités sur les labels***

À l'intérieur d'une boucle (REPEAT/WHILE/UNTIL) ou dans une macro, il est possible d'utiliser des labels locaux de la même façon qu'avec l'assembleur intégré de Winape en préfixant le label par le caractère '@'.

À chaque itération de boucle et ce, pour chaque imbrication de boucle, un suffixe est ajouté au label local contenant la valeur hexadécimale du compteur interne de répétition. Il est ainsi possible d'appeler un label local à une répétition en dehors de la boucle, mais cet usage n'est pas conseillé.

Il est possible d'utiliser la déclaration désuète d'un label en le préfixant d'un point. L'appel à ce label se fera sans le point du début.

Il est possible d'utiliser la valeur d'un label dans une commande (ORG par exemple) si et seulement si le label précède la directive.

Le préfixe {BANK} devant un label (exemple: {BANK}monlabel ) permet de récupérer la valeur de la BANK dans laquelle est déclaré le label, plutôt que l'adresse du label.

Le préfixe {PAGE} devant un label (exemple: {PAGE}monlabel ) permet de récupérer la valeur type Gate array de la BANK dans laquelle est déclaré le label, plutôt que l'adresse du label. Par exemple pour un label situé dans la BANK 5 -> #C4

## Expressions

Rasm utilise un moteur d'expression et de comparaison simple. Il supporte les opérateurs et fonctions suivant(e)s:

### ***Opérateurs de calcul***

- \* multiplication
- / division
- + addition
- - soustraction
- & opérateur booléen ET
- | opérateur booléen OU
- ^ opérateur booléen OU exclusif
- § modulo (utilisez MOD si votre clavier ne dispose pas de ce caractère)
- AND, OR, XOR, MOD en mode maxam
- sin() calcul de sinus
- cos() calcul de cosinus
- asin() calcul d'arc-sinus
- acos() calcul d'arc-cosinus
- atan() calcul d'arc-tangente
- int() conversion en nombre entier
- floor() conversion au nombre entier directement inférieur
- abs() valeur absolue
- ln() logarithme népérien
- log10() logarithme base 10
- exp() exponentielle
- sqrt() racine carrée

### ***Opérateurs de comparaison***

- == égalité (ou un seul = en mode maxam)
- != différent de
- <= inférieur ou égal
- >= supérieur ou égal
- < inférieur
- > supérieur



### ***Spécificités sur les expressions***

Les opérateurs de comparaison ne doivent pas être imbriqués dans des parenthèses:

syntaxe correcte: `IF (4+mavar)*5==monresultat`

syntaxe incorrecte: `IF ((4+mavar)*5==monresultat)`

## Variables statiques ou alias

Il est possible de créer des alias avec la directive EQU. Ces alias ne peuvent pas être modifiés une fois qu'ils sont définis.

## Variables dynamiques

Rasm autorise un nombre illimité de variables pour des calculs internes.

Syntaxe: `mavariabile=5` ou `LET mavariabile=5`

Ces variables peuvent être utilisées comme compteur de boucle ou comme offset lors d'une boucle de répétition.

Exemples:

```
dep=0
repeat 16
ld (ix+dep),a
dep=dep+8
rend
```

```
ang=0
repeat 256
defb 127*sin(ang)
ang=ang+360/256
rend
```

## Directives

Une directive n'est pas une fonction, c'est un mot clef qui doit être séparé de ses paramètres par au moins un espace.

Syntaxe correcte: `ASSERT (4*mavar)`

Syntaxe incorrecte: `ASSERT(4*mavar)`

***ASSERT <condition>***

Vérifier une condition et arrêter l'assemblage si la condition est fausse.

## ***EQU***

Créer un alias

Exemples:

`mavariabLe EQU 5`

`monautre EQU mavariabLe*2`

## ***ALIGN <valeur>***

Aligner le code produit sur une valeur multiple du paramètre valeur.

Exemples:

`ALIGN 2` pour aligner sur une adresse paire

`ALIGN 256` pour aligner sur le poids fort d'adresse

## ***AMSDOS***

Ajoute un entête Amsdos au fichier binaire produit automatiquement par Rasm.

Note: Cet entête n'est pas ajouté lors d'un SAVE.

## ***BUILD CPR***

Pour forcer l'écriture de la cartouche lorsqu'on a utilisé une instruction SAVE.

## ***BUILD SNA***

Pour forcer l'écriture d'un snapshot (v3).

### ***BANK [<numéro de page ROM, numéro de page RAM>]***

Sélectionner un emplacement ROM (cartouche) ou RAM (snapshot). L'usage de cette instruction active par défaut l'écriture de la cartouche en fin d'assemblage.

Les valeurs possibles vont de 0 à 31. En mode snapshot on peut aussi utiliser cette directive, cette fois avec des valeurs de 0 à 35 (64K de base + 512K d'extension mémoire).

Sans paramètre, la directive ouvre un nouvel espace mémoire de travail.

### ***BANKSET <numéro de bloc de 64K>***

Sélectionner un emplacement mémoire pour les snapshots, en groupant les pages 4 à 4. Le format snapshot v3 supportant au maximum une extension de 512K, il y a 9 sets mémoire indexés de 0 à 8.

Il est possible d'utiliser BANK et BANKSET en même temps mais il est impératif de ne pas sélectionner la même page à la fois avec BANK et BANKSET. Un contrôle déclenchera une erreur si vous tentez de le faire.

L'appel de cette fonction active automatiquement la génération de snapshot.

## ***IF, IFNOT, ELSE, ELSEIF, ENDIF***

Permet d'écrire du code conditionnel.

Exemple:

```
CODE_PRODUCTION=1
[...]
IF CODE_PRODUCTION
OR #80
ELSE
PRINT 'Version de test'
ENDIF
```

## ***LZ48 / LZ49 / LZ4 / LZX7 / LZEXO***

Ouvrir un segment de code compressé en LZ48,LZ49, LZ4, ZX7 ou Exomizer. Le code produit sera compressé après assemblage et le code suivant le segment compressé sera relogé.

Il n'est pas possible d'appeler un label situé après un segment compressé depuis le code compressé pour des raisons évidentes dûes aux aléas de la compression. Une erreur s'affichera expliquant pourquoi.

Limitations:

- Pour le moment, le code avant compression et les éventuelles données situées après ne peuvent pas dépasser l'espace d'adressage de 64Ko.
- Il n'est pas possible d'imbriquer les segments compressés.

## ***LZCLOSE***

Fermer un segment compressé.



## ***READ / INCLUDE 'fichier à lire'***

Lire un fichier texte et l'intégrer au code source à l'emplacement de l'instruction de lecture. Le chemin relatif de lecture a pour racine l'emplacement du fichier dans lequel est l'instruction de lecture. Un chemin absolu s'affranchit de ce répertoire racine.

Il n'y a pas de limite de récursivité en lecture. Attention à ce que vous faites.

## ***INCBIN 'fichier à lire',[offset],[size],[offset étendu],[OFF]]]]***

Lire un fichier binaire. Les données lues seront directement injectées. Les paramètres optionnels sont compatibles avec la fonction INCBIN de Winape. L'offset n'est pas limité à 64Ko comme Winape. L'offset étendu est là pour compatibilité.

Il est possible de donner un offset négatif, relatif à la fin du fichier.

Il est possible de donner une taille de fichier négative. La taille lue sera égale à la taille totale du fichier ajoutée à cette valeur négative. Pour tout lire sauf les 10 derniers octets, on précisera une taille de -10 dans la commande.

Une taille de zéro chargera tout le fichier.

Paramètre OFF: Si on souhaite charger un fichier pour initialiser de la “mémoire” et qu'on souhaite assembler du code par dessus, on peut désactiver pour la lecture de ce fichier le contrôle d'écrasement.

Exemple:

```
org #4000
incbin'makeraw.bin',0,0,0,OFF ; lecture en #4000
org #4001
defb #BB ; écrasement du deuxième octet sans erreur
```

***INCL48 / INCL49 / INCLZ4 / INCZX7 / INCEXO 'fichier à lire'***

Lire un fichier binaire, le compresser en LZ48, LZ49, LZ4, Exomizer ou ZX7 et l'injecter directement dans le code.

***LIMIT <adresse limite>***

Imposer une limite plus basse à l'écriture de l'assemblage. Par défaut, la limite est de 65536 mais on peut avoir besoin de ne pas dépasser une certaine valeur. Pour protéger une zone définie, il vaut mieux utiliser la fonction PROTECT.

***ORG <adresse logique>[,<adresse d'écriture du code>]***

Assembler le code à une adresse spécifique. On peut optionnellement choisir d'assembler le code à une adresse, mais l'écrire à un autre endroit avec le deuxième paramètre.

***PROTECT <adresse début>,<adresse fin>***

Empêcher l'écriture du code dans la zone début/fin de l'espace mémoire courant.

***STR '<chaine>','<chaine2>',...***

Similaire à DEFB '<chaine>', le dernier caractère de chaque chaine a son bit 7 forcé à 1 (OR #80 sur le dernier octet de chaque chaine).

***STOP***

Arrêter l'assemblage.

## ***PRINT '<string>',variables,expression,...***

Écrire du texte, variables ou expressions lors de l'assemblage. Il est possible de formater les variables en préfixant la variable par des tags:

**{hex}** afficher la variable en hexadécimal. Si la variable vaut moins de #FF alors l'affichage sera forcé sur deux chiffres. Si la variable vaut moins de #FFFF alors l'affichage sera forcé sur quatre chiffres. Au dessus il n'y aura pas d'extra-zéros.

**{hex2}**, **{hex4}**, **{hex8}** pour forcer l'affichage quel que soit la valeur, sur 2, 4 ou 8 chiffres.

**{bin}** afficher la variable en binaire. Si la variable vaut moins de #FF alors l'affichage sera forcé sur 8 bits. Si la variable vaut moins de #FFFF alors l'affichage sera forcé sur 16 bits. Un pré-traitement enlève les 16 bits supérieurs de la valeur 32 bits au cas où tous les bits sont à 1 (nombre négatif).

**{bin8}**, **{bin16}**, **{bin32}** pour forcer l'affichage quel que soit la valeur, sur 8, 16 ou 32 bits.

**{int}** afficher en décimal, nombre entier.

Sans préfixe la variable ou expression est affichée en nombre flottant.

## ***RUN <valeur>***

Cette directive n'est prise en compte que si on génère un snapshot. Alors l'adresse de démarrage du code sera injectée dans le snapshot.

## ***LIST / NOLIST / LET / BRK***

Fonctions ignorées, pour compatibilité avec Winape.

## **WHILE / WEND**

Répète un bloc d'instructions tant que la condition est vérifiée.

Exemple:

```
cpt=10
while cpt>0
ldi
cpt=cpt-1
wend
```

## **REPEAT <n> / REND | REPEAT / UNTIL <condition>**

Répète un bloc d'instructions. On peut soit fixer un nombre de répétitions, soit utiliser le mode conditionnel avec la directive de fin de bloc UNTIL.

Exemples:

```
repeat 16
ldi
rend
```

```
cpt=10
repeat
ldi
cpt=cpt-1
until cpt>0
```

## ***WRITE DIRECT <rom basse>,<rom haute>,<RAM>***

En spécifiant une rom basse (entre 0 et 7) ou une rom haute (entre 0 et 31), cette instruction remplace la directive BANK et a le même effet.

En spécifiant uniquement l'adresse RAM (n'importe quelle valeur) et en désactivant les numéros de rom avec la valeur -1, on crée à chaque appel un nouvel espace mémoire. On peut ainsi assembler plusieurs code au même emplacement mémoire, mais dans un espace différent.

Exemple:

```
;par default un espace memoire est cree, ORG en zero
defs 65536,0
WRITE DIRECT -1,-1,#C0
defs 65536,0
; cree un nouvel espace memoire peu importe
; la valeur de la BANK, ici #C0 deux fois
WRITE DIRECT -1,-1,#C0
; ecriture au meme endroit ne genere pas d'erreur
defs 65536,0
```

Rasm aura créé trois espaces mémoire. L'espace par défaut et un espace supplémentaire à chaque WRITE DIRECT. Il n'y a pas de limite au nombre d'espace mémoire dynamique autre que la mémoire totale de votre système.

Il n'est pas possible de revenir sur un espace mémoire dynamique une fois qu'on l'a quitté.

### ***SAVE 'fichier binaire à écrire',<adresse>,<taille>***

Enregistre le fichier binaire correspondant à la mémoire adresse jusqu'à adresse+taille. Bien que l'instruction SAVE puisse être déclarée à n'importe quel moment, les enregistrements de fichier sont toujours réalisés en fin d'assemblage si et seulement si il n'y a pas eu d'erreur.

En conséquence de quoi il n'est pas possible d'enregistrer des états intermédiaires d'assemblage.

### ***CHARSET 'chaine',<valeur> | <code>,<valeur> | <début>,<fin>,<valeur>***

La directive permet de redéfinir des valeurs aux caractères assemblés entre quotes selon quatre possibilités:

- 'chaine',<valeur> Le premier caractère de la chaine aura pour nouvelle valeur la <valeur>. Le caractère suivant, la <valeur>+1 et ainsi de suite pour tous les caractères de la chaine.
- <code>,<valeur> Attribuer au caractère de numéro <code> la valeur <valeur>.
- <début>,<fin>,<valeur> Attribuer aux caractères de <début> à <fin> une valeur incrémentale en partant de <valeur>.
- “aucun paramètre” réassigne à tous les caractères leur valeur par défaut.

Cette fonction est compatible Winape.



## ***SWITCH, CASE, BREAK, DEFAULT, ENDSWITCH***

La syntaxe est une mimic du case en C, avec la particularité de pouvoir écrire plusieurs CASE qui ont la même valeur, ce qui donne plus de souplesse au code conditionnel.

Dans cet exemple, l'appel de la macro avec 5 assemblera toutes les chaines de defb avec 'oui' ou 'encore oui':

```
macro ouioui mavar
switch {mavar}
    nop ; pas assemblé car hors case
    case 3
        defb 'non'
    case 5
        defb 'oui'
    case 7
        defb 'encore oui'
        break
    case 8
        defb 'non'
    case 5
        defb 'encore oui'
        break
    default
        defb 'non'
endswitch
mend
```

## Macros

Rasm supporte les macros avec chevrons (compatible avec Winape). Il est possible de faire de l'assemblage conditionnel avec les macros car à chaque appel de macro, le code d'origine est inséré, les paramètres substitués et enfin le code est interprété de façon classique.

Exemple pour une écriture longue distance générique:

```
macro LDIXREG registre,dep
if {dep}<-128
    push bc
    ld bc,{dep}
    add ix,bc
    ld (ix+0),{registre}
    pop bc
elseif {dep}>128
    push bc
    ld bc,{dep}
    add ix,bc
    ld (ix+0),{registre}
    pop bc
else
    ld (ix+{dep}},{registre}
endif
mend
```

Note: Le marqueur de fin de macro peut être indifféremment MEND ou ENDM

## Instructions

Toutes les instructions documentées et non documentées sont supportées.

L'adressage 8 bits des registres d'indexe IX et IY se fait indifféremment avec lx ou xl, hx ou xh, etc.

Les instructions complexes s'écrivent de la façon suivante:

```
res 0,(ix+0),a  
bit 0,(ix+0),a  
sll 0,(ix+0),a  
rl  0,(ix+0),a  
rr  0,(ix+0),a
```

Syntaxe des instructions de port non documentées:

```
out (<n>),a ; <n> est une valeur 8 bits  
in a,(<n>)  
in 0,(c) ou in f,(c)
```

## Limitations

- Il n'est pas possible d'utiliser la mnémonique d'une instruction Z80 comme un label.
- Il n'est pas possible d'utiliser le même nom pour un label, une variable ou un alias.
- Rasm n'est pas sensible à la casse.
- Les blocs de code à compresser dynamiquement ne peuvent excéder 64K.

## Exemples de programmation avec RASM

### ***Récupérer le numéro de bank d'un label pour connecter une ROM***

L'usage du préfixe {BANK} est pratique quand on prévoit d'appeler des routines susceptibles d'être relogées dans n'importe quelle banque.

Bank 0

```
ld a,{bank}maroutine ; sera assemblé LD A,1
call connect_bank    ; il faudra faire la votre!
jp maroutine
```

bank 1

```
defb 'coucou'
maroutine
jr $
```