RASM

roudoudou Assembler
v0.38

Documentation RASM v0.40 (c) Édouard BERGÉ 2017

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. »

Documentation RASM v0.40 (c) Édouard BERGÉ 2017

Documentation RASM v0.40 (c) Édouard BERGÉ 2017

1. 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 publié 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.

2. Installation

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

3. Compilation

compilation Linux:

```
cc rasm_v038.c -lm -lrt
mv a.out rasm
strip rasm
```

compilation Windows:

```
cl.exe rasm_v038.c -DOS_WIN -Ox
```

4. 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 BUILDCPR.

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.

5. Utilisation de la ligne de commande

-i <fichier à assembler> seule option indispensable au bon fonctionnement de Rasm, elle précise le nom du fichier à assembler.

-s exporter les symboles au format Rasm
-sp exporter les symboles au format Pasmo
-sw exporter les symboles au format Winape

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

- L'émulateur Arnold est compatible avec Rasm et Pasmo.
- 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

-v mode verbeux, affiche des informations et statistiques sur l'assemblage

-m mode de compatibilité avec maxam

-o <préfixe de sortie> définir le nom de base pour les fichiers de sortie par défaut (fichier binaire, cartouche, symboles). Par défaut la valeur du préfixe de sortie est “rasmoutput”.

6. Format du code source

6.1 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.

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.

6.2 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

6.3 Valeurs litérales

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 #.
- En valeur ascii si un caractère unique est entre quote.
- En valeur “interne” à une variable ou label si la litérale 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 chaine de calcul pour les besoins en nombres entiers.

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

6.4 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.

6.5 Fichiers

6.6 Spécificités sur les labels

À l'intérieur d'une boucle (REPEAT/WHILE) 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.

7. Expressions

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

- * multiplication
- / division
- + addition
- - soustraction
- & opérateur booléen ET
- | opérateur booléen OU
- ^ opérateur booléen OU exclusif
- § modulo
- 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
- 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

8. Variables

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

Syntaxe: `mavariable=5` ou `LET mavariable=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
```

9. Directives

9.1 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

9.2 AMSDOS

Ajoute un entête Amsdos au fichier binaire produit automatiquement par Rasm. Note: Cet entête n'est pas ajouté lors d'un SAVE.

9.3 BUILDCPR

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

9.4 BANK

Sélectionner un emplacement ROM. L'usage de cette instruction active automatiquement l'écriture de la cartouche en fin d'assemblage. Les valeurs possibles vont de 0 à 31.

9.5 IF, IFNOT, ELSE, ENDIF

Permet d'écrire du code conditionnel.

Exemple:

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

9.6 LZ48 / LZ49

Ouvrir un segment de code compressé en LZ48 ou LZ49. 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.

9.7 LZCLOSE

Fermer un segment compressé.

9.8 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.

9.9 INCBIN 'fichier à lire'[,offset[,size[,offset étendu]]]

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.

9.10 INCL48 / INCL49 'fichier à lire'

Lire un fichier binaire, le compresser en LZ48/LZ49 et l'injecter directement dans le code.

9.11 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.

9.12 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.

9.13 PROTECT <adresse début>,<adresse fin>

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

9.14 STOP

Arrêter l'assemblage.

9.15 PRINT

Écrire du texte lors de l'assemblage.

9.16 NOLIST / LET / RUN <valeur> / BRK

Fonctions ignorées, pour compatibilité avec Winape.

9.17 WHILE / WEND

9.18 REPEAT / UNTIL / REND

9.19 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 créé à chaque appel un nouvel espace mémoire. On peut ainsi assembler plusieurs code au même emplacement mémoire, mais dans un espace différent.

Exemple:

```
;par defaut 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.

9.20 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.

9.21 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.

9.22 MACRO / MEND

Voir chapitre sur les macros

10. 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
else
if {dep}>128
	push bc
	ld bc,{dep}
	add ix,bc
	ld (ix+0),{registre}
	pop bc
else
	ld (ix+{dep}),{registre}
endif
endif
mend
```

11. 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)
```

12. Limitations

- Il n'est pas possible d'utiliser la mnémonique d'une instruction Z80 comme un label.