Syntaxe d'un fichier

Les commentaires commencent par `;` et s'étendent jusqu'à la fin de la ligne.

Sections disponibles :

`.text` signale le début d'une section d'instructions assembleur.

`.data` signale le début d'une section de déclarations de données.

Chaque instruction ou déclaration peut être localisée par un *label* qui sera converti en une adresse par l'assembleur. Un label peut être composé des caractères suivants : `a-z`, `A-Z`, `0-9` ou `_`.

On localise une instruction/déclaration à l'aide de la syntaxe :

```
label:
    instruction ou déclaration
```

On note que l'adressage de la mémoire se fait par mots de 32 bits (et pas moins !). Pour faire appel à des fonctions ou des sections définies dans un autre fichier on utilise la syntaxe suivante à n'importe quel endroit du fichier : `.include file`. Dans ce cas, le code du fichier référencé est inséré à cette position.

Les dossiers utilisés pour la recherche des fichiers à insérer est configurable via la ligne de commande.

Quel flag ? À la GCC avec un `-I` ?

Données

Chaines de caractères

Les chaines de caractères sont composées des caractères suivants :

- Les caractères ASCII affichable (codes 32 à 126 inclus).
- Le caractère nul noté `\0` (code 0).
- Le caractère d'appel noté `\a` (code 7).
- Le caractère de tabulation noté `\t` (code 9).
- Le caractère de saut de ligne noté `\n` (code 10).

Ajouter les caractères permettant de se déplacer dans le terminal ? Quid de la gestion de «l'écran» par le simulateur ? Le `\a` est là pour faire *ding* :)

Les caractères `"` et les `\` sont échappés : `\"` pour le guillemet et `\\` pour la barre oblique inversée. Chaque caractère est codé sur 8 bits, le bit de poids fort étant toujours fixé à 0.

Puisque les calculs ne peuvent s'opérer que sur des mots de 32 bits, on encode les caractères par bloc de 4, complété si besoin par plusieurs caractères nul, de la manière suivante :

$$\texttt{ab\textbackslash nc} \rightsquigarrow \texttt{01100001 01100010 00001010 01100011} \rightsquigarrow \overline{1633815139_{10}}$$

Vraiment, Vraiment pas pratique pour la manipulation des chaines... Soit faire une stdlib poussée pour manipuler les chaines (concat, substr, getchr, ...) Soit faire de l'adressage mémoire par octet : BadBadBad mal au cerveau.

Entiers

Les entiers peuvent être donnés dans plusieurs bases :

— En base décimale, base par défaut.

— En base binaire, lorsque préfixés par `0b`.

— En base hexadécimale, lorsque préfixés par `0x`.

Déclaration des données

On utilise les notations suivantes pour déclarer des données :

`.ascii "..."`	déclare une chaine de caractère. Cette chaine n'est pas forcément terminée par `\0`.
`.string "..."`	déclare une chaine de caractère forcément terminée par `\0`.
`.uint ...`	déclare un entier non signé sur 32 bits (de 0 à $2^{32} - 1$).
`.int ...`	déclare un entier signé sur 32 bits (de -2^{31} à $2^{31} - 1$).

Instructions

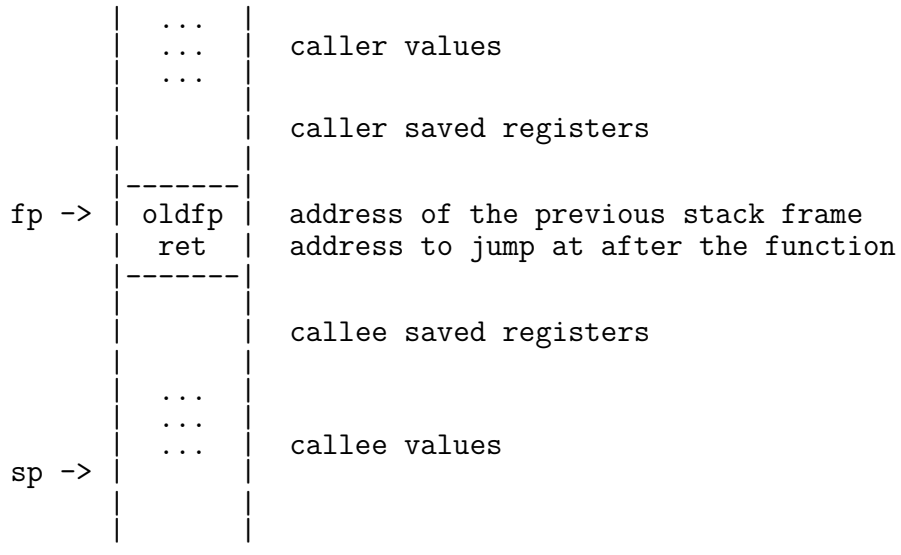
Registres

L'assembleur possède 31 registres de travail : de `r0` à `r28`, `rout`, `sp`, `fp`. Les registres suivants ont un sens particulier :

`r0`	n'est pas modifiable et a comme valeur 0.
`r1`	n'est pas modifiable et a comme valeur 1.
`rout`	est supposé être utilisé afin de stocker la valeur de retour des fonctions.
`sp`	a comme valeur la prochaine adresse libre du tas.
`fp`	a comme valeur l'adresse du tableau d'activation de la fonction en cours.

Convention d'appel

La pile est organisée de cette manière lors d'un appel de fonction :



Les arguments d'une fonction sont données dans les registres `r20` à `r28`. Les registres de `r15` à `r28` sont dits *caller-saved* : une fonction est susceptible d'écraser la valeur de ces registres sans les restaurer. Les registres de `r0` à `r14` sont dits *callee-saved* : lors de l'appel à une fonction, cette dernière ne doit pas modifier ces registres.

Labels

Les *labels* ne peuvent être déclarés qu'une seule fois sauf les *labels* ayant un entier en base décimale comme identifiant. Dans ce cas, on fait référence au *label* N précédent par N`b` et au label suivant par N`f`. La référence à un *label* N de manière directe n'est pas autorisée.

Format des Sauts

Lors de saut les formats d'adresse suivants sont autorisés :

— Saut à une adresse absolue. L'adresse est donnée sans signe dans n'importe quelle base, préfixée de `$`. Exemple : `jmp $12` saute à l'adresse 12 du programme.

— Saut à une adresse relative. L'adresse est donnée avec un signe (`+` ou `-`) dans n'importe quelle base, préfixée de `$`. Exemple : `jmp $+12` saute 12 instruction après, `jmp $-12` saute 12 instruction avant.

— Saut à un *label*. L'adresse absolue est calculée par l'assembleur. Exemple : `jmp lbl` saute au label *lbl* du programme.

Vraiment pas pratique de gérer à la fois les labels "numériques" et les sauts à des offsets... Pourquoi autoriser les offsets ? C'est bien les labels.

Instructions

Instructions Logiques

Instruction	Destination	Arguments		Description
`and`	$r1$	$r2$	$r3$	et logique
`orr`	$r1$	$r2$	$r3$	ou logique
`nor`	$r1$	$r2$	$r3$	non-ou logique
`xor`	$r1$	$r2$	$r3$	ou exclusif logique
`not`	$r1$	$r2$		non logique

Instructions Arithmétiques

Instruction	Destination	Arguments		Description
`add`	$r1$	$r2$	$r3$	addition
`sub`	$r1$	$r2$	$r3$	soustraction
`mul`	$r1$	$r2$	$r3$	multiplication
`div`	$r1$	$r2$	$r3$	division
`neg`	$r1$	$r2$		négation
`inc`	$r1$	$r2$		incrémente
`dec`	$r1$	$r2$		décrémente

Décalages

Instruction	Destination	Argument	Description
`asr`	$r1$	$r2$	Décalage à droite arithmétique
`lsr`	$r1$	$r2$	Décalage à droite logique
`lrl`	$r1$	$r2$	Décalage à gauche

Opération sur le Tas

Instruction	Destination	Argument	Description
`push`		$r1$	Empile $r1$ sur le tas
`pop`	$r1$		Dépile le sommet du tas dans $r1$

Opération sur la mémoire

Instruction	Destination	Argument	Description
`mov`	$r1$	$r2$	Copie le registre $r2$ dans $r1$
`load`	$r1$	m	Copie la valeur de la mémoire à l'adresse m dans r
`li`	$r1$	i	Copie la valeur i dans $r1$
`loadi` `loadia`	$r1$	$\$l$	Copie l'adresse associée à l dans $r1$