Thibaut Barnel - Pierre Coimbra

Structure du code

Nous avons segmenté notre v1 en deux modules, un module hex qui permet de transformer une instruction assembleur en un code hexadécimal et un module tools qui contient des fonctions que l'on utilisera dans tous les modules de notre programme.

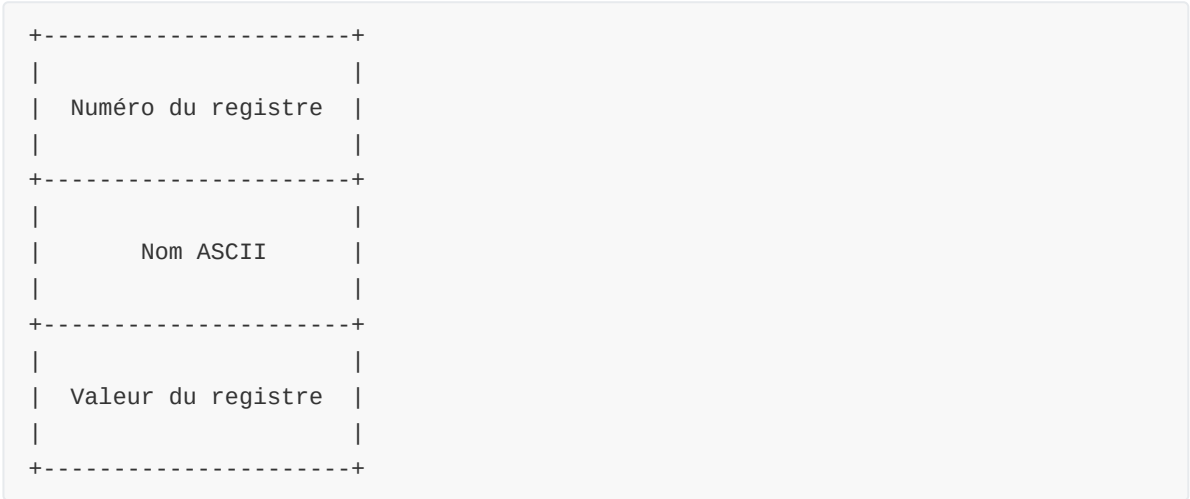
Dans la suite, nous allons ajouter d'autres modules :

- Un module registre et un module mémoire détaillé plus bas
- Un module calcul que nous réaliserons plus tard et qui s'occupera d'exécuter les instructions (opérations, utilisations des registres et de la mémoire...)
- Un module console qui permet de gérer l'affichage et qui utilisera tous les autres modules pour faire fonctionner notre émulateur.

Module registres

L'objectif de ce module est de fournir une implémentation des registres d'un processeur MIPS en C. Ce module permettra de lire la valeur d'un registre, de la changer et de traduire un registre mnémonique.

Pour la représentation en mémoire de ces registres, nous avons procédé de la manière suivante, une structure représentant un registre organisé de la manière suivante :



Le numéro du registre, le nom mnémonique et la valeur du registre seront initialisés par une fonction à l'aide d'un fichier. Par défaut, la valeur du registre sera initialisée à une valeur arbitraire pour éviter des valeurs aléatoires.

Afin de pouvoir utiliser la même structure pour les 32 registres GPR. Si le numéro du registre est supérieur à 31 on à affaire à un registre spécialisés. Cela permet de simplifier la recherche de ces registres.

- Le numéro du registre sera un nombre entier entre 0 et 34, on le stockera sous la forme d'un int.
- Les registres spéciaux PC,HI,LO auront comme numéro respectif 32,33,34
- Le nom mnémonique sera un tableau de 5 char car le plus long est zero.
- La valeur du registre sera un unsigned long int (non signé grâce au complément à deux).
  Il y a donc 35 registres pour notre implémentation. Nous allons donc utiliser un tableau de

35 cases avec chaque case pointant vers une structure décrite ci-dessus associé à un registre.

Les 35 registres seront stockés dans un tableau de 35 cases ou chaque case contiendra la structure associée au registre portant le numéro de la case.

Ainsi pour trouver la structure associé à un registre, il nous suffira d'aller voir dans la case du tableau associé au registre, on pourra à partir de là consulter ou modifier la valeur du registre.

Module mémoire

Une case mémoire est composée de 32 bits.

La mémoire est stockée entre les adresses 0x0000 et 0xFFFF.

Les instructions pouvant accéder à la mémoire dans notre implémentation sont LW et SW. Elles permettent de faire le lien entre un registre et la mémoire en transférant un mot de 32 bits soit 4 octets.

Ces 4 octets devront être stockés à la suite dans la mémoire en respectant la norme Big Endian (octet de poids fort à l'adresse la plus basse). De plus, l'adresse du premier octet d'un mot devra être divisible par 4.

En résumé la quantité de mémoire utilisée par un programme n'est pas prédictible.

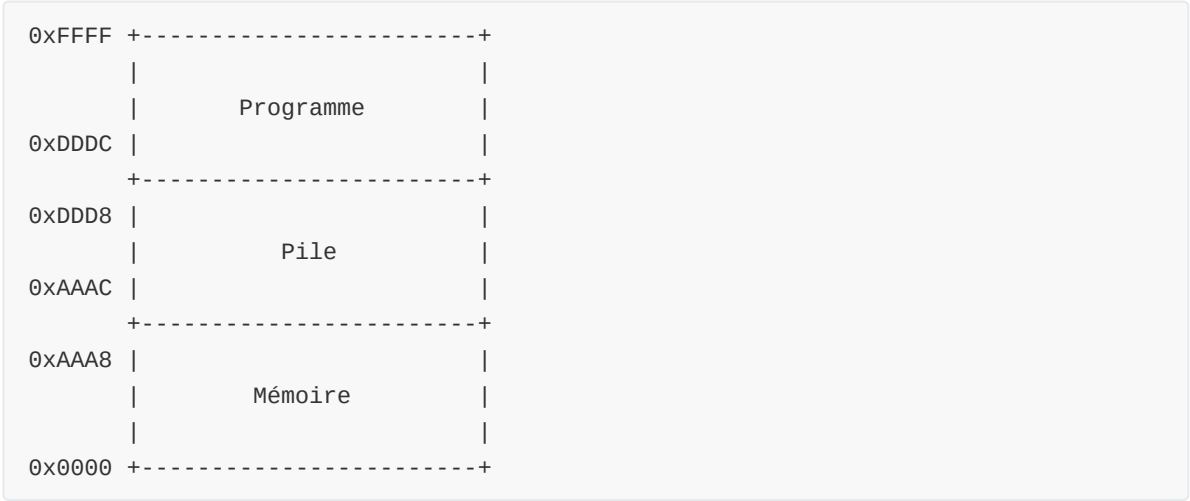
La première question soulevée est la suivante :

- Faut-il avoir une mémoire gérée par mots (bloc de 4 octets) ou une mémoire gérée par octet ?

Vus que pour notre implémentation, nous n'utilisons pas les directives et que les deux seules opérations accédant à la mémoire sont LW et SW.

Nous avons fait le choix d'une implémentation modulaire sous forme de liste chaînée supportant à la fois l'adressage à l'octet et l'adressage au mot. Dans notre cas, nous stockerons que des mots, l'adresse dans la mémoire à laquelle ce mot sera accessible devra donc être l'adresse du premier octet de ce mot (et donc un multiple de 4).

Nous allons utiliser la map mémoire suivante.



La mémoire est accessible avec les instructions *LW* et *SW* se trouvera donc entre les adresses 0x0000 et 0xAAA8 et sera remplit des adresses bases aux adresses hautes.

Pour la pile, elle se remplit par le bas donc elle se trouvera entre 0xDDD8 et 0xAAAC. Cela nous laisse de la place si on veut par la suite ajouter des parties à notre map mémoire.

Le programme sera lui stocké entre les adresses 0xDDDC et 0xFFFF. Ce qui est suffisant pour stocker notre programme dans notre cas.

Vu que la mémoire ne sera pas trop utilisée nous allons utiliser une liste chaînée dynamiquement alloué pour utiliser le moins de mémoire possible sur l'hôte.

On aura donc la map mémoire qui contiendra :

- Une partie pour la mémoire
- Une partie pour le programme
- Une partie pour la pile descendante

L'utilisation de la pile est gérée par le registre *sp* qui pointe vers l'adresse du haut de la pile. On peut donc utiliser la même implémentation pour la pile et pour la mémoire seule l'utilisation varie.

De même pour le programme, l'instruction en cours sera pointée par le registre *PC*. À noté qu'avant d'incrémenter le PC il faudra vérifier que l'on est bien une case après.

Pour l'implémentation, nous avons retenu celle d'une unique liste chaînée.

Chaque maillon de la liste sera composé comme décrit ci-dessous :

- L'adresse de la case mémoire du premier octet du mot (un multiple de 4)
- La valeur du mot sous forme de tableau
- Un pointeur vers l'adresse suivante, si ce pointeur vaut NULL, c'est la fin de la mémoire

Pour des questions de simplicité, on veillera à ce que les adresses reste dans l'ordre. De plus, si on insère une valeur dans une case mémoire déjà existante, on remplacera la valeur.

En résumé, un maillon contiendra un mot et sera désigné par l'adresse de la case mémoire du premier octet du mot.

NB: si l'on souhaite un jour stocker par octet il suffira de rajouter des fonctions d'insertion pour un octet, l'implémentation ne changera pas.