SONKO Mohamet P2023
CHANAL Valentin 22/12/2020

Émulateur MIPS - Bilan

Sommaire :

Présentation :

Notre objectif est de faire un émulateur d'un processeur MIPS.

L'émulateur devra :

- Lire un programme (consistant en une suite d'instruction) sous format texte dans un fichier (.txt)
- Écrire la traduction de chaque instruction sous format hexadécimal dans un autre fichier texte (.txt)
- Exécuter le programme en simulant l'intégralité du processeur, c'est-à-dire :
    - Simuler l'ensemble des registres
    - Simuler la mémoire de donnée
    - Veillez à ce que les données dans le processeur ne dépasse 4 Go (capacité d'un processeur MIPS)
    - Effectuer les bonnes interactions en fonction de l'instruction exécuté

L'émulateur disposera d'un mode pas à pas et d'un mode interactif qui vont respectivement :

- Effectuer ce que fait déjà l'émulateur mais en lisant et exécutant les instructions une à une.
- Ouvrir un champ de texte accessible à l'utilisateur où il pourra écrire les instructions qu'ils veulent exécuter dans l'émulateur directement.

Structure et fonctionnement de l'émulateur :

Afin de répondre à l'ensemble des contraintes pour ce processeur, il est nécessaire de séparer le code de l'émulateur en plusieurs modules qui interagiront entre eux. Après concertation, nous avons séparé notre code en cinq modules :

- Traduction des instructions en hexadécimal
- Gestion des registres
- Gestion de la mémoire de donnée
- Lecture / Écriture / Exécution du programme
- Fonctions utiles supplémentaires / nécessaires

Nous présentons chacun de ces modules, ainsi que leur rôle dans les points suivants.

Lecture du programme, Écriture du code hexa, exécution du programme, *file* :

Ce module est chargé de lire le fichier en entrée, trier les informations, organiser les instructions du programme et écrire la forme hexadécimal de chaque instruction dans un fichier en sortie. L'entièreté du module est basée sur une structure appelée *"instruction"*.
Elle contient les champs suivants :

- champ *address* (entier non signé)
- champ *line* (chaîne de caractères)
- champ *line_hexa* (entier non_signé)
- champs *next* et *prev* (pointeur vers un élément de type *structure instruction*
- champ *exec* (pointeur vers une fonction à argument de type entier)

Une instruction sera associée à un élément de type *structure instruction*.

Le champ ***address***, comme son nom l'indique, contient l'adresse de l'instruction dans la mémoire du processeur.

Les champs ***line*** et ***line_hexa*** contiennent l'instruction en détail respectivement dans sa forme assembleur en chaîne de caractère et dans sa forme hexadécimal en entier non signé.

Le champ ***exec*** indique la fonction à exécuter. Il pointera vers la fonction correspondant à l'instruction (Pour une instruction "*ADD*", le pointeur pointera vers la fonction "*exec_ADD*" par exemple).

Un champ *Label* sera éventuellement ajouté. Ce sera une chaîne de caractère traduisant le label associé à l'instruction.

Comme l'indiquent les champs ***next*** et ***prev***, l'objectif principal de ce module est d'effectuer une liste chaînée contenant la totalité du programme. Cette liste chaînée, appelée ***"prog"*** sera la base de l'ensemble de l'émulateur et servira pour lire le programme assembleur, l'écrire sous forme hexadécimal, et l'exécuter.

Il fallait trouver un moyen de lire le programme et de différencier toutes les instructions les unes des autres, sachant que la taille du programme n'est pas connue en avance. L'idée de faire une liste chaînée "*prog*" est rapidement venue et les différents champs se sont ajoutés au fur et à mesure. Au lieu de faire une petite liste contenant seulement les chaînes de caractère et l'utiliser seulement pour les traductions en hexadécimal (ce qui était prévu à la base), nous avions décidé d'utiliser cette même liste

pour les autres fonctionnalités de l'émulateur. Il fallait donc un ensemble de fonctions qui initialise et remplit les différents champs de la liste chaînée *"prog"* qui seront utilisés par les fonctions des autres modules.

Traduction des instructions en hexadécimal, *translation* :

Ce module regroupe les fonctions reliées à la traduction des lignes d'opérations MIPS en hexadécimale. Nous avons alors créé 3 types de fonctions de traduction générale pour les opérations directes (*ADD*, *AND*, …), les opérations immédiates (*ADDI*, *BEQ*, …) et les opérations jump (*T*, *JAL*). Ces fonctions sont celles créant le mot de 32 bits résultant de la traduction. Ces trois fonctions sont ensuite appelées par les fonctions de traduction particulières pour chaque opération. Nous avons fait cela car la structure en hexadécimal de toutes les opérations sont d'une de ces formes là et cela rend facile l'implémentation de nouvelles opérations. Enfin, nous lisons la ligne de façon à récupérer l'opération et les différents opérandes. Nous utilisons alors la bonne fonction de traduction en fonction de l'opération lue.

La gestion des labels sera effectuée dans ce module. Une structure "*label*" a été créée et contient les champs suivants :

- champ *label_name* (chaîne de caractères)
- champ *address* (entier non signé)
- champ *next* (pointeur vers un élément de type *structure instruction*

Nous avons opté pour une liste chaînée contenant les différents labels du programme ainsi que l'adresse qui lui est associée. Lors de la traduction d'une instruction de type *Jump* (en l'occurrence *J* et *JAL*), nous vérifions s'il s'agit d'un paramètre entier ou d'un label. Le cas échéant, alors une recherche du label sera effectuée dans la liste chaînée de labels, afin de trouver l'adresse qui lui est associée.

Gestion des registres + exécution des instructions, *registers* :

Ce module permet de gérer la valeur des registres et d'exécuter les instructions.

Pour la simulation de registres, nous avons décidé de les stocker de façon globale. Nous avons donc créé un tableau contenant tous les registres. Aussi, nous avons créé 3 pointeurs globaux vers les registre *pc*, *hi* et *lo* car nous voulions que leurs utilisations soit plus claires. Nous avons choisi d'utiliser des variables globales car cela permet à toutes les fonctions de les utiliser sans les passer en argument. Ce qui aurait pu être un peu lourd. Toutes ces variables globales sont initialisées grâce à la fonction *init_register*. Nous avons ensuite 3 fonctions permettant de lire et d'écrire dans le tableau de registre ainsi qu'une fonction pour afficher l'état des registres.

Enfin, toutes les autres fonctions de ce modules sont les fonctions d'exécution des instructions. Nous avons décidé d'en faire une par opération MIPS.

Gestion de la mémoire de donnée, *memory* :

Ce module a pour rôle de simuler la mémoire de donnée du processeur MIPS. Il fournira des fonctions de lecture de données et d'écriture en mémoire et sera principalement utilisé par le module ***registers.*** Une seconde liste chaînée y est mise en place, utilisant une structure nommée ***cell***. Elle contient les champs suivants :

- champ *address* (entier non-signé)
- champ *data* (valeur sur un octet)
- champ *next* (pointeur vers un élément de type *cell*

La liste chaînée se nommera ***data_memory*** (les champs ont des noms assez explicites).

Au lieu de simuler d'un coup les 4 Gio de mémoire directement, nous avons choisi une allocation dynamique au fur et à mesure que les valeurs sont stockées dans la mémoire de donnée. Si une demande d'écriture en mémoire se passe, l'émulateur vérifiera d'abord si la case mémoire existe déjà. Sinon, elle créera un nouvel élément de la liste d'adresse demandé et l'ajoutera à la liste (trié par ordre croissant en fonction de l'adresse). Une variable sera présente pour vérifier qu'on ne dépasse les 4 Gio de mémoire.

De plus, les fonctions *SW* et *LW* lisent et stockent des valeurs de 4 octets en plusieurs cases d'un octet chacun. L'émulateur devra donc, lors du stockage, séparer le mot en 4 et stocker ces valeurs là dans les cases d'adresses qui se suivent, et lors de la lecture, prendre l'octet à la case mémoire demandé, ainsi que les 3 octets suivants, les rassembler, et les envoyer dans le registre correspondant.

Ajout de fonctions utiles supplémentaires, *function* :

Ce module a pour but de fournir quelques fonctions supplémentaires, utiles pour l'émulateur mais n'appartenant pas forcément à un module. On y trouvera des fonctions de conversions, de recherche de caractères, ou encore une fonction qui permet de créer des masques.

Fonctionnement :

Deux possibilités nous sont offertes en lançant l'émulateur : le mode exécution de programme à partir d'un fichier et le mode interactif. Nous décrirons chacun de ces modes.

Exécution via un fichier :

L'émulateur est exécuté de la façon suivante :
& ./emul-mips *fichier.txt* **(-pas)**

Le fichier devra être placé dans le dossier *tests/*. L'émulateur ira chercher directement dans le dossier en question. La mention *-pas* est optionnelle et permet d'activer le mode pas à pas.

Les noms des fichiers source et destination sont identifiés grâce à une fonction *test_and_hexified* et la liste chaînée de programme est initialisée. Ensuite, la fonction de lecture du fichier est exécutée.

Cette fonction de lecture du fichier source est une des fonctions les plus importantes du programme puisqu'elle permettra d'initialiser tout ce qui est nécessaire pour l'exécution de ce dernier, en l'occurrence :

- Les adresses des instructions
- L'ordre des instructions
- La forme hexadécimale de la fonction qui sera écrite en sortie et qui sera utilisé lors de l'exécution de l'instruction
- Le pointeur de fonction qui mène vers les fonctions d'exécution correspondantes.
- Les étiquettes (labels)

L'émulateur entier se repose sur cette liste chaînée (sauf en mode interactif). Il est donc important qu'elle soit bien fonctionnelle et complète.

Cette fonction fait appel à d'autres sous-fonctions pour remplir les champs différents de la structure *instruction :* une de traduction (traduit vers la forme hexadécimale), une qui va associée l'instruction à la fonction d'exécution qui lui correspond, et enfin une qui mettra à jour la liste chaînée de labels. Si le mode pas à pas est activé, alors à chaque lecture d'instruction, un retour de cette dernière, ainsi que l'adresse de l'instruction et sa traduction hexadécimale sont affichés à l'écran.

Concernant la fonction de traduction hexadécimale, il est important de noter que les labels sont identifiés lors de la lecture d'une instruction de type saut et sont traduits avec l'adresse correspondante grâce à la liste chaînée de labels.

Le tableau des registres ainsi que la liste chaînée représentant la mémoire seront initialisés.

L'émulateur va ensuite exécuter la fonction d'écriture et écrire dans le fichier, en parcourant la liste chaînée du programme préalablement remplie.
Un récapitulatif du programme sera ensuite affiché. L'exécution commence dès qu'un appui sur le bouton [Entrée] est détecté ; la fonction *execution()* sera exécutée.

Cette fonction *execution()* parcourera la liste chaînée de programme et exécutera les instructions une à une grâce au champ *exec* de la structure. Lors de chaque exécution, le *program counter* est systématiquement mis à jour. Pour passer à la prochaine instruction, un comparateur entre l'adresse de l'instruction de la liste et le *program counter* a lieu et un parcours sur la liste chaînée est effectué jusqu'à trouver l'adresse correspondante. (Méthode de parcours du programme pertinente pour les instructions de sauts.)
Si le mode pas à pas est activé, alors diverses options seront proposées à l'utilisateur :

- [c] pour continuer le programme
- [n] pour afficher la prochaine instruction
- [r] pour afficher les registres
- [m] pour afficher l'état de la mémoire

Concentrons nous sur les registres et la mémoire de données.

Nous avons modélisé les registres en un tableau de valeurs à 35 cases : 32 cases pour les 32 registres différents, et 3 supplémentaires pour le compteur de programme et les deux registres *HI* et *LO*. Les instructions appelleront les registres avec l'index correspondant pour modifier le registre en question.

La mémoire est, comme dit précédemment, modélisée par une liste chaînée. Accessible uniquement avec les instructions *Store Word* et *Load Word*. Un module entier est consacré à cette liste chaînée.
Un compteur de données est mis en place dans le module *memory* qui a pour objectif de vérifier qu'on ne stocke pas plus de 4 Gio en mémoire.

Une fois que le programme est terminé, l'état final des registres et de la mémoire sera affiché, ainsi que la quantité de données utilisée. On libéra les mémoire de données, de labels et du programme et l'émulateur se fermera.

Mode Interactif :

L'émulateur en mode interactif est exécuté de la façon suivante :
& ./emul-mips

Ce mode particulier permet à l'utilisateur d'écrire directement les instructions et de les exécuter directement en direct.
Pour ceci, la mémoire de donnée et les registres sont bien initialisés et on appellera une autre fonction présente dans le module *file*, *interactive().*

Un élément de type *instruction* y est créé et aura les champs *address, next* et *prev* non initialisée (car inutile). À chaque fois qu'une instruction y sera entrée, elle sera automatiquement traduite avec la fonction de traduction, et aura son champ *exec* assigné à la fonction d'exécution correspondante. Elle exécutera la fonction, et donnera l'option à l'utilisateur d'afficher la mémoire et/ou les registres, ou de passer à une prochaine instruction.
Pour quitter l'émulateur, il suffira de mettre EXIT en instruction. L'émulateur sortira de la fonction *interactive()*, libérera la mémoire de donnée, et quittera l'émulateur.

Répartition des tâches :

Durant ce projet, nous avons débattu du fonctionnement de nos différents modules et fonctions tous les deux afin d'être sûr que lors de l'implémentation, l'agencement se passe pour le mieux.

Ensuite lors de l'implémentation, nous nous sommes répartis le travail. Par exemple, Mohamet s'est occupé de la gestion de la mémoire et de la lecture et écriture des fichiers. Valentin s'est, quant à lui, occupé de la partie traduction et exécution des instructions.

Cependant, même si nous nous sommes répartis clairement les implémentations, nous avons quand même fait certaines parties ensemble. Par exemple, l'implémentation des fonctions d'exécution en utilisant des pointeurs de fonctions étant une des parties que nous ne maîtrisions le moins, nous l'avons fait à deux.

Conclusion :

Pour conclure, nous avons atteint l'objectif de créer un émulateur MIPS. Il contient bien toutes les fonctionnalités attendues. Comme nous avions rarement fait de projet aussi conséquent, nous pensons avoir plutôt bien réussi à nous organiser comme il fallait et mener ce projet de programmation à terme.

Nous avons pensé pour le compléter à créer une page de manuel, grâce à un *-help* ou à l'utilisation de *man*. Ce serait un complément intéressant mais nous n'avions pas pu le réaliser, faute de temps.