Marie-Ange Stefanos
Chloé Poulic

Rapport de projet n°1

20/09/2020

Introduction **1**

Gestion des commandes de l'interpréteur **1**

La commande exit 1
La commande load 2
La commande assert 2
La commande disp (display) 3
Les autres commandes : debug, resume, set 3

Modélisation de l'état de l'interpréteur **3**

Gestion des erreurs et génération des traces **3**

I. Introduction

Ce livrable 1 permet de se familiariser avec l'interpréteur de commande. L'objectif est de programmer les commandes relatives à la gestion de l'environnement de l'émulateur et relatives au test du programme en cours. Ce développement de code s'accompagne par une série de jeux de tests mais aussi de mains afin de tester les fonctions créées.

Au cours de cette semaine de recherches et d'écriture, il nous a paru utile de créer des nouveaux types, dont celui des registres. Ainsi ceux-ci vous seront présentés dans ce rapport afin de clarifier leur structure.

De plus sera présenté la modélisation de l'état de l'interpréteur, ainsi que les méthodes adoptées pour gérer les erreurs et générer des traces.

II. Gestion des commandes de l'interpréteur

A. La commande exit

Nous avons traité dans un premier temps la commande exit, afin de la réaliser nous n'avons rien développé d'autre que le dossier donné initialement.

Afin de réaliser le jeu de test, nous avons considéré deux cas : le cas où l'utilisateur rentre correctement la commande, et le cas où l'utilisateur passe des arguments à exit. Nous avons alors fait le choix de retourner une commande d'erreur lorsque l'utilisateur se trouve dans le deuxième cas cité

précédemment. Nous aurions pu aussi faire le choix de laisser l'écriture d'arguments possible, cependant nous avons arbitrairement choisi qu'un argument serait synonyme d'erreur.

B. La commande load

La commande load était déjà fournie mais pas totalement terminée. L'objectif ici était de la rendre plus robuste. En effet, donnée ainsi, la commande load ne permettait pas de bloquer des adresses dont l'hexadécimal été erroné. C'est-à-dire, que 0x3000ZZZZ est accepté pour la commande, alors que 0x3000ZZZZ n'est pas un nombre hexadécimal valide. Il faut donc que votre code soit capable de détecter un nombre hexadécimal valide/invalide.

Pour ce faire, nous avons modifié la fonction is_hexa(char*token) donnée dans interpreter.c. Afin de modifié l'algorithme, nous avons utilisé la fonction strtol qui permet de convertir une chaîne de caractère en un long. Nous avons donc vérifié que l'expression donnée dans token était bien un pointeur vers une valeur hexadécimale existante.

C. La commande assert

La commande assert prend en paramètre valeur, adresse et registre. Or les registres ne sont pas encore définis dans notre projet. Nous nous sommes donc penché sur ce problème. La création de ce type nous a mené à la réalisation de fonctions qui vont nous permettre de manipuler les registres mais aussi de faire le lien entre le nom des registres et leur mnémonique.

a. Création du type reg_tab

Nous avons décidé de travailler sur des tableaux afin de réaliser nos registres. Chaque registre est défini comme étant une structure composée de son nom, introduite à travers un tableau de dimension 4, et de son contenu, content qui sera un word, donc un entier non signé. Nous avons défini, ici, reg.

Enfin nous avons créé une autre structure, qui est un tableau de taille 35 composé d'éléments reg. de cette manière, nous avons débuté la création des registres. Ce type vient compléter la structure machine.

b. Création du type mnemo_tab

Un problème s'est posé lorsque nous avons pris en compte le fait qu'un même registre avait deux noms, par exemple, $zero et $0. Ainsi, nous avons procédé de la même façon que pour la création de registre. mnemo_tab est un tableau de taille 36 composé de char de longueur 6.

c. Fonctions développées autour de ces types

Nous avons donc implémenté les fonctions de création de registre et de mnémonique. Pour les registres, nous avons créé un tableau de 35 éléments qui contiennent les noms des différentes possibilités parmi par exemple $0,$31, HI, LO, PC , et des valeurs aléatoires. Pour les mnémoniques, nous avons créé un tableau des noms de registres $zero, $fp, HI, LO donc les numéros d'indices correspondent à l'entier placé derrière le $ pour son registre associé. C'est-à-dire que le mnémonique $sp est l'élément au rang 29 du tableau mnemo_tab.

Nous avons aussi créé les fonctions dont les objectifs sont les suivants :

- vérifier que le registre demandé existe effectivement
- trouver l'indice du registre entré (string) dans le tableau de registres
- affecte une valeur dans le registre entré

- affiche le contenu d'un registre
- affiche le contenu du tableau de registre
- convertir le registre entré en mnémonique dans son autre valeur

Cependant nous avons un phénomène que nous ne comprenons pas dans la fonction is_mnemo. En effet nous voulions comparer la chaîne de caractère qui suit le $ dans le nom du registre (pour les registres du type $4) avec "0" et "31" en utilisant le code ASCII. Cependant l'instruction implémentée de fonctionnant pas comme nous le souhaitions nous avons fait une comparaison avec l'entier qui suivant le $ car nous maîtrisons mieux ceci. Cependant, si nous avons le temps, nous réessayerons notre première idée.

d. Commande assert

Nous avons créé une disjonction de cas suivant la commande donnée par l'utilisateur. La fonction identifie s'il s'agit de l'étude d'un byte ou d'un word, stocke la valeur dont on souhaite trouver la présence soit à l'adresse identifiée, soit dans le registre transmis. Les fonctions que nous avons précédemment créées permettent le bon fonctionnement de cette nouvelle commande.

D. La commande disp (display)

Par manque de temps nous n'avons pas pu pousser notre code jusqu'au bout pour réaliser cette fonction. Le squelette présenté, ici, est l'idée que nous avons pour faire fonctionner la commande disp.

E. Les autres commandes : debug, resume, set

Pour la commande debug, nous n'avons pas créé de types ou de fonctions particulières. En effet, il fallait lire un script, repérer "debug" et à ce moment changer le mode de l'interpréteur : SCRIPT devient INTERACTIVE pour que l'utilisateur inscrive des commandes dans l'interpréteur.

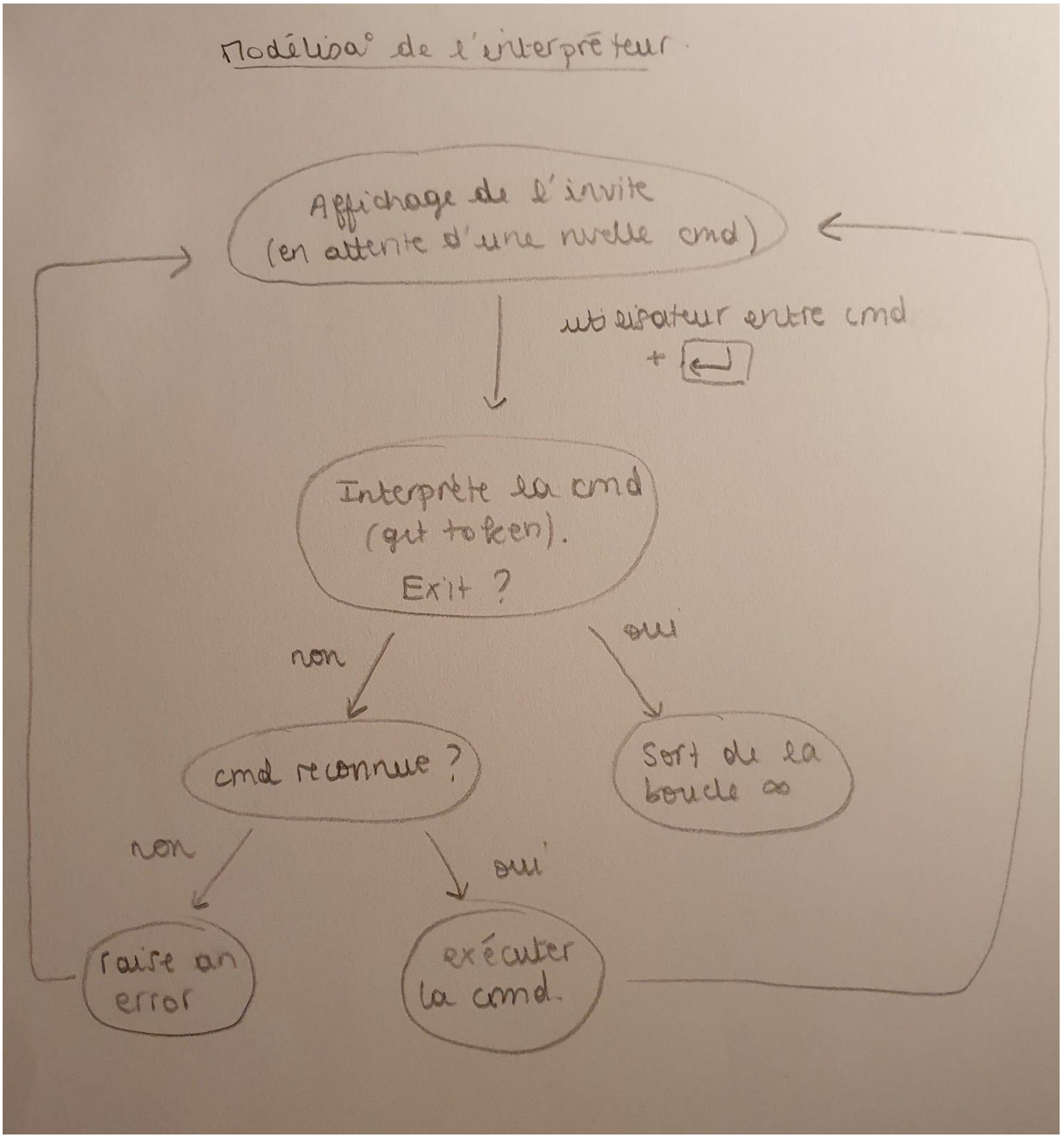
De la même manière, la fonction resume faisant l'opération inverse de la précédente, nous n'avons rien créé de plus que la fonction de commande pour qu'elle fonctionne.

Pour la fonction set, il y a un cas qui fonctionne et ceci ne devrait pas être le cas. En effet si $zoom est donné en argument, la commande set fonctionne normalement alors qu'un message d'erreur devrait s'afficher. Ceci est dû à un mauvais fonctionnement de la fonction is_mnemo. Nous allons donc la modifier dès que possible pour que cela fonctionne.

De plus, quand nous lançons la commande ./simpleUnitTest.sh -e ./emulmips -b testing/cmd/set/*.emu, les tests mènent tous à un succès. Ceci est dû au fait que le test n'est pas mené jusqu'au bout. Nous ne comprenons pas pourquoi, ceci doit venir de la fonction de la commande elle-même.

Modélisation de l'état de l'interpréteur

Pour modéliser l'état de l'interpréteur nous avons réalisé la machine d'état suivante :



Gestion des erreurs et génération des traces

Afin de générer des erreurs au sein du code, nous avons utilisé les commandes suivantes

- MAKE_ERR_LOC
- ERROR

Et pour générer les traces, nous avons utilisé la commande INFO.

Gestion de la mémoire et des registres

La machine est constituée d'une mémoire segmentée en différentes sections stockées continûment dans la mémoire virtuelle et de registres.

D'une part, nous avons créé les registres sous forme de structure composée d'une chaîne de caractère (le nom conventionnel du registre) et de son contenu. Pour plus de simplicité, nous avons également créé la structure reg_tab correspondant à un tableau de registres, nous permettant d'y accéder facilement grâce à leur indice. Enfin, les registres ayant deux noms, le premier que l'on considère comme conventionnel ($nombre) et le second, le nom mnémonique. Afin de passer de l'un à l'autre, nous avons créé une fonction qui associe chaque mnémonique à l'indice du tableau correspondant.

Les fonctions principales associées à cette structure de registre sont les suivantes : initialisation du tableau de registres, recherche de l'indice correspondant à un registre à partir de son nom, remplissage d'un registre par une valeur donnée et récupération d'une donnée dans un registre.

D'autre part, la mémoire est chargée et initialisée lors du chargement (commande load). Pour pouvoir accéder à une case mémoire grâce à son adresse virtuelle, nous avons créé une fonction permettant la conversion de l'adresse virtuelle en adresse réelle, c'est-à-dire qui trouve le numéro du segment dans laquelle elle se trouve ainsi que son indice dans le tableau de ce segment.

Cela nous a donc entre autres permis d'écrire les fonctions set_mem et get_mem (deux versions pour chacune : une pour les byte, l'autre pour les word, sachant qu'il fallait penser à passer en little endian pour les mots). Comme pour les registres mais cette fois à partir d'une adresse virtuelle, l'une écrit dans la mémoire la valeur voulue, l'autre récupère la valeur qui y est stockée.

Toutes ces fonctions sont particulièrement utiles pour les commandes set, assert et disp qui ont besoin d'avoir accès à des valeurs stockées dans la mémoire ou dans un registre, que ce soit pour y stocker une donnée, au contraire afficher la valeur stockée, ou encore la comparer à une autre.