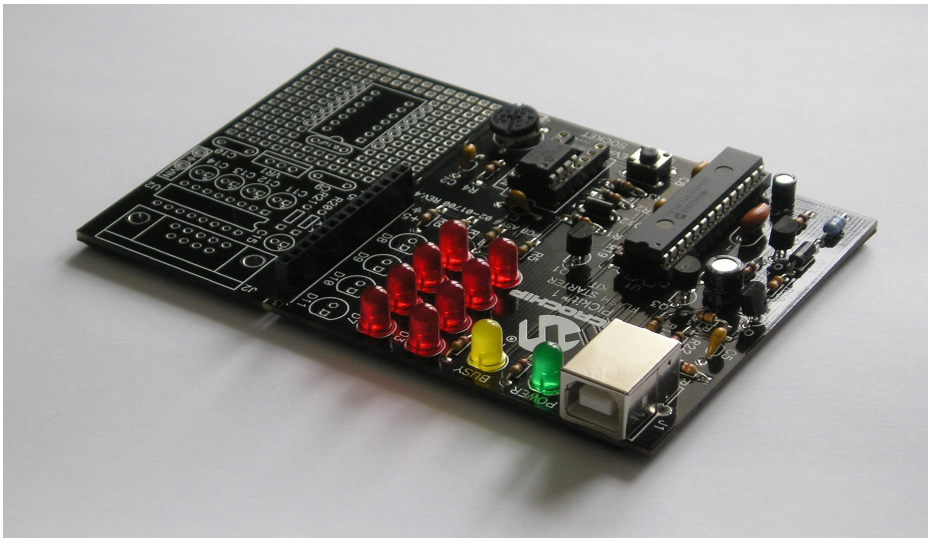
Maxime DICKERSON
Sophie FRANÇOISE
2A SICOM

7 octobre 2013

Projet d'informatique : simulateur MIPS

Livrable n° 1



Professeurs : Nicolas CASTAGNÉ et François PORTET

Introduction

L'objectif de ce projet est de réaliser en langage C un simulateur de microprocesseur MIPS. Dans ce premier livrable, nous nous sommes concentrés sur l'implantation des registres et de la mémoire, et la réalisation de quelques commandes liées à l'utilisation basique du simulateur (ouvrir un fichier, afficher et écrire dans la mémoire et les registres, quitter le programme). Ces commandes (sauf trois qui seront spécifiées) ne sont pas encore fonctionnelles, mais doivent être capable de vérifier que les paramètres entrés par l'utilisateur sont valides.

I – Les modules .c/.h

Nous avons utilisé l'interpréteur de commandes fourni sur la page du projet, en y apportant des modifications au fur et à mesure de l'implantation de nos fonctions. Pour ce faire, nous avons créé différents modules de fichiers .c/.h.

1) memoire.c/memoire.h, fichier reg.h

Ce module sert à définir les structures nécessaires à l'utilisation des registres et de la mémoire. En effet, un microcontrôleur MIPS comporte 36 registres de 32 bits (32 registres d'utilisation générale et les registres PC SR HI et LO). Les registres sont définis sous la forme d'une structure :

```
typedef struct {
        char mnemo[5]; // nom du registre
        int numero;
        int valeur; //32 bits est la taille d'un int
        } REGISTRE;
```

L'ensemble des 36 registres est rassemblé dans un tableau, et la fonction `void init_reg(REGISTRE tab[])` permet de remplir le champ mnémonique de chaque registre ($at, $s8, etc).

Nota : La structure REGISTRE est définie dans le fichier reg.h et non memoire.h à cause de problèmes de compilation que nous n'avons su résoudre que de cette manière. Reg.h n'a que cette utilité et nous envisageons de nous en passer à l'avenir.

Un fichier ELF comporte trois sections auxquelles nous nous intéressons : .text .data et .bss. On définit donc une structure nommée SECTION :

```
typedef struct {
        unsigned int adresse_debut;
        unsigned int taille;            //en octets, lue dans le programme au moment où on le charge
        char nom[taille_nom];
        int* donnees;
        } SECTION;
```

Le fichier d'entrée au format ELF contenant les instructions en assembleur pourra donc être chargé en mémoire *via* la structure suivante :

```
struct ENTREE {
        SECTION text;
        SECTION data;
        SECTION bss;
        };
```

2) parseExecute.c/parseExecute.h

Ce module contient toutes les fonctions utiles à la lecture et à l'exécution des commandes entrées dans l'invite de commandes. Nous y avons donc déplacé la fonction parse_and_execute_cmd_testcmd qui se trouvait dans simMips.c, pour plus de clarté dans le texte du programme principal. Il contient pour l'instant les fonctions concernant les commandes décrites dans le paragraphe 3.6.1 du sujet :

– la fonction fournie dans le simMips.c

```
int parse_and_execute_cmd_testcmd(char * paramsStr);
int execute_cmd_testcmd(int hexValue);
```

– sortie du programme

```
int parse_and_execute_cmd_exit(char * paramsStr);
int execute_cmd_exit();
```

– ouverture d'un fichier ELF

```
int parse_and_execute_cmd_lp(char* paramsStr);
int execute_cmd_lp(char* token);
```

– affichage des registres

```
int parse_and_execute_cmd_dr(char* paramsStr);
int execute_cmd_dr_un(int num_registre);          //cas où un seul registre est demandé
int execute_cmd_dr_tous();                        //affichage de tous les registres
```

– écriture dans les registres

```
int parse_and_execute_cmd_lr(char* paramsStr);
int execute_cmd_lr(int num_reg, int value);
```

– affichage de la mémoire

```
int parse_and_execute_cmd_dm(char* paramsStr);
```

– écriture dans la mémoire

```
int parse_and_execute_cmd_lm(char* paramsStr);
```

– affichage du code assembleur;

```
int parse_and_execute_cmd_da(char* paramsStr);
```

Nous n'avons pas modifié la fonction de sortie du programme. Les fonctions d'écriture et d'affichage des registres sont opérationnelles. Les fonctions d'écriture et d'affichage de la mémoire peuvent vérifier la correction de la syntaxe des paramètres, mais ne peuvent pas encore vérifier que les adresses entrées soient valides.

3) fonctions.c/fonctions.h et automateF.c/automateF.h

Ces deux modules contiennent des fonctions auxiliaires appelées par les fonctions principales du type parse_and_execute_cmd_<commande>. Pour l'instant il n'y a que les deux fonctions suivantes :

```
int isregister(char* param);          //teste si param fait référence à un registre

int automate(char* nombre );          //teste le type de nombre : entier, binaire, etc
```

II – Vérification du programme

1) Les tests

Chaque fonction doit s'accompagner d'un jeu de tests permettant de vérifier chaque aspect de son fonctionnement (ou dysfonctionnement). Cependant, nous avons rencontré des difficultés liées à la compréhension des tests, puis à leur mise en œuvre : pendant longtemps nos test n'écrivaient rien dans le fichier <test>.out, ce qui faisait systématiquement échouer le test même si la fonction testée était efficace. C'est pour cela que nous notre livrable ne comporte que trois tests, et qu'ils ne « passent » pas.

Cependant, si l'on entre « à la main » les commandes contenues dans les fichiers <test>.simcmd, le programme renvoie bien les résultats attendus (en cas d'absence d'un paramètre, de mauvais paramètre, etc). Seule l'automatisation des tests reste donc à voir. Nous corrigerons ce problème avant le deuxième livrable.

2) Gestion des erreurs et génération des traces

Lorsqu'une commande rencontre un paramètre incorrect ou manquant, nous utilisons les macros définies dans le fichier notify.h pour afficher des messages WARNING_MSG. De plus, pour suivre l'évolution de certaines fonctions complexes, nous utilisons des messages DEBUG_MSG afin de vérifier le passage par les différentes étapes, les tests et les entrées dans les boucles.

Conclusion

Pour ce premier livrable, nous avons pu implémenter toutes les fonctions demandées : elles peuvent vérifier les paramètres entrés dans l'invite de commande, et dans le cas de dr et lr, le programme est capable de les exécuter. Malheureusement, nous n'avons pas pu réaliser tous les tests nécessaires à la vérification (bien qu'au cours de l'écriture des fonctions nous ayons déjà testé manuellement les retours des fonctions). Ce retard ne semble pas trop pénalisant.