Séance 1

Tout d’abord nous nous sommes préalablement approprié le sujet ainsi que les annexes fournis pour gagner un maximum de temps.
On crée plusieurs fichier de tests, que l'on test avec le compilateur fourni pour être sur de ce que l'on voudra lorsque l'on testera notre programme.
Pour tester efficacement, on crée les tests après différentes utilisations des instructions.
On test également la syntaxe de lecture comme les lignes vides, les commentaires en début et fin de fichier, ou à la suite d’une instruction.

Séance 2

On a commencé à créer une fonction qui : décode un fichier texte se trouvant dans le dossier "tests" en hexadécimal et place ce code hexadécimal dans un nouveau fichier texte dans le dossier "hexified".
Pour l’instant, l’implémentation des noms mnémoniques des registres n’a pas été effectuée car non obligatoire. Le professeur nous ayant indiqué qu'on en avait pas besoin.
Nous avons observé plusieurs problèmes.
Premièrement, la lecture du fichier renvoyait un « segmentation fault » car en fonction de la position des commentaires, en début/fin de fichier ou après une instruction, une double ligne vide dans le fichier …
Ensuite, nous avons utilisé les deux fonctions fscanf et fread pour lire le fichier. En particulier, nous utilisons fscanf pour connaître l’instruction et fread pour parcourir le fichier.

Séance 3

Nous avons terminé d'implémenter le décodage des instructions commencé en séance 2, nous avons fait toutes celles présentes dans l'annexe 2, sauf l'instruction System Call car nous ne l'utiliserons pas.
Après l'avoir testé plusieurs fois avec les tests réalisés lors de la séance 1, le programme fonctionne.

Séance 4

On a créé deux modules : un module qui s'occupe de toutes les opérations sur les registres et un autre pour les opérations en mémoire qui peut écrire et charger en mémoire.
Pour implémenter la mémoire nous avons créé une structure ainsi qu'une liste chaînée, comme ça même si la taille de la mémoire est fixe (4 Go), nous n'avons pas besoin d'allouer un si grand espace pour la mémoire lorsque nous lançons l'émulateur. Notre structure mémoire fonctionne donc en liste chaînée avec un compteur pour ne pas dépasser les 4 Go de mémoire.
Nous avons également diviser notre mémoire en deux parties : une partie pour stocker les instructions qui vont être lu en suivant le programm counter et une autre pour stocker les données. Ces deux parties incrémentent bien sur le même compteur pour ne pas dépasser les 4 Go.
Pour gérer la mémoire nous avons décidé de faire la lecture et l'écriture en mémoire par mot (4 octets) car le processeur lit en mémoire par mot.

Séance 5

Nous avons continué à écrire nos différentes fonctions du module registre.c qui correspondent à chaque instruction.
Nous avons bloqué un long moment pour les instructions de type immédiate qui utilise une valeur signé, car par exemple lorsque la valeur immédiate était négative, et que le registre rt était le 4ème registre, c'était le 3ème registre (n-1) qui était pris en compte. Nous avons regardé d'où venait l'erreur : du mask utilisé pour isoler la valeur de rt ou alors il y avait une erreur dans notre code pour écrire l'instruction en hexadécimal qui nous avait échappée. Le mask étant correct, nous avons vu que la valeur de rt dans le code hexadécimal était n-1, si n est le registre que nous demandions. Après vérification avec ce que nous donne l'émulateur fourni, on a bien le même code hexadecimal, donc si l'on traduit directement ce que donne l'émulateur fourni, on a pour une valeur immédiate négative rt=n-1.
Comme nous n'avons pas réussi à comprendre d'où venait cette erreur et qu'elle était présente aussi dans l'émulateur fourni, nous avons donc décidé de faire une disjonction de cas dans les instructions de type immédiate, lorsque la valeur immédiate est positive ou négative. Lorsqu'elle est négative nous allons chercher le registre rt+1.
Nous avons le même problème pour rs cette fois ci pour bgtz et blez, nous avons effectué les mêmes vérifications que précédemment et donc effectué la même disjonction de cas pour pallier à ce problème, mais cette fois sur rs.

Séance 6

Nous avons commencé à implémenter la fonction interactif et l'exécution pas à pas.
Pour la fonction interactif, nous avons créé une nouvelle fonction dans le module de conversion, qui du coup ne lit plus les instructions dans un fichier mais sur l'entrée standard et donc n'écrit plus le résultat en hexadécimal dans un autre fichier mais le garde dans une variable pour pouvoir ensuite la passer à la fonction qui décode et exécute les instructions.

Séance 7

Nous avons terminé l'implémentation de la nouvelle fonction de conversion sur l'entrée standard, nous l'utilisons avec deux autres fonctions que l'on a défini dans le module main, au dessus de la fonction main (nous n'avons pas crée un nouveau module) qui nous permettent : pour l'une de vider le buffer de l'entrée standard avant de prendre une nouvelle ligne d'instruction de la part de l'utilisateur (pour par exemple éviter qu'un retour à la ligne soit pris en compte lors d'une nouvelle saisie sur l'entrée standard) et pour l'autre de vérifier si le nom de l'instruction (ADD, ADDI, SW…) est valide. Nous avons ajouté aussi la possibilité pour l'utilisateur de demander à afficher la mémoire et/ou les registres.

Séance 8

Nous avons ajouté la fonction pas à pas à notre programme et effectué des tests sur cette dernière et sur l'utilisation en mode interactif afin de vérifier leur bon fonctionnement.
Puis nous avons rédigé le bilan.

BILAN

Avant de préciser quelles tâches ont été effectuées par chaque membre du groupe, nous allons expliciter quelles fonctionnalités du processeur MIPS nous n'avons pas implémentées :

- -Étiquettes
- -Directives
- -Les instructions J, JAL et SYSCALL
- -Noms mnémoniques des registres

Puis une présentation de notre code pour l'assembleur constitué de 4 modules et d'un main :

Convert :

Ce module permet de traduire la forme textuelle des instructions MIPS en leur forme hexadécimal.

Register :

Ce module s'occupe de toutes les opérations sur les registres : il contient une fonction principale qui décode une instruction sous forme hexadécimal et des fonctions annexes qui permettent d'exécuter l'instruction décodée.

Memory :

Ce module nous permet de gérer la mémoire : écrire et lire la mémoire mais aussi supprimer tout ce qui s'y trouve lorsque l'on arrête l'émulateur.

Pour implémenter la mémoire nous avons créé une structure pour pouvoir utiliser une liste chaînée, comme ça même si la taille de la mémoire est fixe (4 Go), nous n'avons pas besoin d'allouer un si grand espace directement pour la mémoire lorsque nous lançons l'émulateur.
Notre liste chaînée est triée par ordre croissant des adresses de la mémoire, chaque élément de la liste contient donc une variable pour l'adresse et une variable contenant ce qui est stocké à cette adresse.

Notre structure mémoire fonctionne donc en liste chaînée avec un compteur pour ne pas dépasser les 4 Go de mémoire. Ce compteur est incrémenté à chaque fois que l'on vient écrire quelque chose en mémoire dans une adresse encore jamais utilisée.

Nous avons également divisé notre mémoire en deux parties pour faciliter son utilisation : une partie pour stocker les instructions qui vont être lu en suivant le compteur programme et une autre pour stocker les données. Ces deux parties incrémentent bien sur le même compteur pour ne pas dépasser les 4 Go.

Pour gérer la mémoire nous avons décidé de faire la lecture et l'écriture en mémoire par mot (4 octets) car toutes les instructions que nous utilisons fonctionnent par mot, le processeur fonctionne par mot.

Enfin, on a décidé de faire la pile dans la mémoire de donnée, en la faisant commencer à

une certaine adresse, adresse qui sera initialisée dans le main au registre $29, le pointeur de pile.

Affichage :

Ce module nous permet d'afficher les valeurs contenues dans les registres ainsi que les cases de la mémoire de données qui sont utilisées.

Main :

Dans le main on initialise les registres (tous à 0 sauf le $29 qui contient l'adresse du début de la pile) et la mémoire pour stocker les données.
Pour le mode non-interactif, on initialise aussi la mémoire pour stocker les instructions, puis on lit le fichier sous forme textuelle et on la traduit en hexadécimal et on stock cette traduction dans la mémoire pour les instructions. Et enfin, on exécute toutes ces instructions en suivant le compteur de programme. A la fin de l'exécution on affiche les registres et la mémoire de données.
Pour le mode interactif, on vient attendre une entrée de l'utilisateur, et on traite sa demande si il s'agit d'une instruction correcte ou d'une demande d'affiche des registres et/ou de la mémoire.

Il y a aussi deux autres fonctions en plus du main que l'on utilisent pour le mode interactif : une fonction pour vider le buffer lors d'une nouvelle entrée de l'utilisateur et une fonction qui vérifie si le nom de l'instruction donnée est pris en compte par notre émulateur.

Rapport de Tache :

Module Convert

convert.c

- Pierre & Hugo: Implémentation du code pour la lecture et l'écriture des fichiers (fscanf / fprintf / fread).
- Hugo: Implémentation de l'ouverture du fichier d'entrée et du fichier de sortie avec le bon path (fopen).
- Hugo: Création des tableaux d'instructions.
- Pierre: Implémentation de la gestion des cas en fonction de l'instruction lue.
- Pierre & Hugo: Conversion des instructions en code hexadécimal.
- Pierre & Hugo: Création de la fonction convert_interactive pour le mode interactif. Réflexion au niveau de la lecture de flux pour lire l'entrée de l'utilisateur.

convert.h

- Pierre: calcul de l'opcode pour chaque instruction.

Mode Affichage

affichage.c

- Pierre: Implémentation de la fonction afficherMemoire qui affiche la mémoire.
- Hugo: Implémentation de la fonction afficherRegistres qui permet d'afficher les 32 registres ainsi que HI et LO.

Module Register

register.c

- Pierre: Implémentation des cas pour chaque instruction.
- Hugo: Implémentation des instructions pour mettre à jour les registres et la mémoire.
- Hugo & Pierre: Réflexion à propos de la gestion du PC et des arguments des fonctions.
- Hugo: Test des fonctions avec le code MIPS et le mode interactif.

registre.h

- Pierre: Calcul des masks.

Module Memory

memory.c

- Pierre: Implémentation des fonctions LW et SW ainsi que vider_memoire.
- Hugo: test des fonctions sw et lw avec des instructions MIPS.

memory.h

- Pierre: Création de la structure case_mem et mémoire pour gérer la mémoire.

Main

- Pierre: Implémentation de la fonction vider_buffer.
- Hugo & Pierre: Implémentation de la fonction dinguerie pour vérifier que l'instruction est correcte.
- Hugo: Création de la structure du main en fonction de la commande entrée par l'utilisateur (0 ou plusieurs arguments).
- Hugo: Implémentation de l'ouverture des fichiers. Création du tableau de registre et de la mémoire.
- Pierre: Implémentation des boucles pour lire le fichier contenant le code en hexadécimal, charger les instructions en mémoire et les décoder.

Pour le mode interactif:

- Hugo: Gestion des commandes dans le mode interactif (afficher registre et/ou mémoire, EXIT).

Makefile

- Hugo: Écriture du Makefile.