PARTIE 1 - Traduction

Pour la traduction, nous avons crée une fonction « trad » qui remplit un tableau de chaine de caractères nommé « Machine » qui, à chaque indice « i» allant de 0 au « nombre de lignes de notre fichier-1 », associe la traduction en binaire de l'instruction située à la ligne « i-1 ». On considérera par la suite que les lignes commencent à 0 (donc Machine[i] traduit bien l'instruction à la ligne i).

« trad » prend en arguments :

- Un tableau de chaine de caractères « Result » qui lit un fichier texte et met dans « Result[i] » la ligne numéro « i » .
- Le tableau de chaine de caractères « Machine » introduit précédemment
- Un tableau de chaine de caractères « Etiquette » qui, à chaque indice « i », associe le nom de l'étiquette située à la ligne numéro « i ».
- « nbLignes », un entier renvoyé par la fonction « recupLinks » (qui initialise le tableau « Result[i] » )
- Un pointeur sur un entier « temoin », initialisé à 0 dans le « main » et modifié à 1 dans la fonction « trad » si un saut vers une étiquette inexistante est demandé.

La fonction « trad » repose sur une disjonction de cas pour traduire les instructions en binaire. Nous avons également du créer des fonctions auxiliaires :

- Une fonction « decimal2binaireCOMP » qui renvoie une valeur décimale traduite en binaire (complément à 2)
- Une fonction « hexToBin » qui traduit de l'hexadécimal au binaire
- Une fonction « pos » qui renvoie l'indice de la chaine de caractère « s » passée en argument dans le tableau « Etiquette », ou -1 si elle n'y apparait pas.
- Une fonction « completerBinDroite » qui complète une chaine de caractères binaires avec des 0 à droite jusqu'à ce que la chaine contienne le nombre de caractères demandé.

Nous avons également introduit des témoins : « val », qui compte le nombre de valeurs qui doivent être saisies selon les différentes instructions : par exemple, « hlt » n'est suivi d'aucun argument tandis que « and » est suivi de 3 arguments. On compare « val » avec notre compteur « cpt » pour s'assurer qu'il n'y a pas d'erreur. Le témoin « temoin » sert à repérer si le programme assembleur fait appel à une étiquette inexistante, ce qui provoque une erreur. Le témoin « transfert » est mis à 1 si on est dans le cas d'une instruction de transfert. Et pour finir, le témoin « hash » vérifié si notre ligne contient un #.

Nous avons rencontrés plusieurs difficultés lors de cette étape. Par exemple, nous avons du faire face à des « Bus error », erreur que nous n'avions jamais vu avant. Nous avions également du apprendre à manier correctement les fonctions de la bibliothèque <string.h> telles que « strcat », « strdup » , qui n'ont ont causé beaucoup de soucis lorsqu'elle n'était pas accompagné d'un « strdup ».

Notre traduction comporte toutefois plusieurs défaut : beaucoup de cas d'erreurs ne sont pas traitées, comme par exemple si il n'y a pas d'hlt à la fin.
De plus, nous n'avons pas réussi a traduire certaines instructions, comme par exemple st* (rd)#n, rn.

PARTIE 2 - Creation des fonctions auxiliaires :

Après avoir créer notre fonction « powbis(x,y) », qui nous renvoie x à la puissance y, nous avons commencer par écrire des fonctions de conversion être les nombres binaires (en complément à 2), les nombres hexadécimaux et les nombres décimaux.
Puis, nous avons commencé par créer notre fonction « add2binaires », pour cela nous nous sommes inspirés des premiers cours d'architecture des ordinateurs. Nous avons donc créé un demi-additionneur (« demiAdd ») , un additionneur complet («addComplet ») et enfin

« add2binaires » qui additionne 2 nombres écrit en binaire. Nous avons entré en argument un pointeur vers une retenue pour pouvoir gérer le registre d'etat C, à la fin de notre simulation.
Pour soustraire 2 nombres binaires nous avons utiliser l'égalité « a-b = a+(-b) ». Nous avons donc pris l'inverse du second nombre binaire et l'additionner avec le premier.

Pour le « and », le « or » et le « xor » nous avons juste comparé bit par bit nos 2 nombres.

Puis, on s'est occupé de la fonction « shr », pour laquelle nous avions eu du mal à bien actualiser le registre d'Etat C.

Une des fonctions qui nous a bien servi est « completerBin » qui prend un nombre binaire de taille quelconque et un entier « taille » qui met des 0 à gauche de notre nombre dans que la taille de notre nombre est pas égale à « taille ».

Pour la division binaire, une des fonctions qui nous a pris le plus de temps, on a différencier les cas ou le résultat doit être positif du cas négatif pour pouvoir ensuite convertir nos 2 nombres en valeur absolue. On a ensuite effectuer la division binaire, pour cela on s'est inspiré de l'algorithme d'Euclide.

PARTIE 3 - Exécution :

Pour l'exécution, nous avons récupérer toutes les lignes du fichier en hexadécimal que nous avons stocké dans un tableau de structures Instructions : cette structure décompose une instruction écrite en binaire en les 5 parties données par l'énoncé (code opératoire (5bits), dest (5bits), src1 (5bits), Imm (1bit) et src2 (16bits).

Quand à la simulation de la machine virtuelle, nous avons crée 3 tableau : un tableau de 32 registres généraux allant de r0 à r31, un tableau de registres d'états comportant 3 caractères correspondant aux registres Z, C et N, et enfin un tableau simulant la mémoire, de 65536 éléments simulant les octets.

Nous avons choisit de stocker tout en binaire, donc en char car que ce soit en binaire, en hexadécimal ou en décimal, beaucoup de conversions doivent se faire.

Pour l'exécution même du programme, nous avons crée un pointeur PC sur la première case du tableau de structures instructions.

Afin de récupérer et gérer plus facilement les données demandées par l'instruction, nous avons fait appel a une seconde structure PCbis qui prennent le code bis en décimal (seule exception au stockage en binaire) pour faciliter le switch d'exécution, les valeurs soit des registres, soit des #.

L'exécution d'une instruction passe donc par un switch en fonction du code opératoire présent dans PCbis->codebis.

Un témoin saut a été mis en place pour savoir si un saut par un JMP, J*… a été effectué, si ce témoin est à 0 à la fin, alors PC est avancé d'une case dans le tableau d'instructions.

Remarques supplémentaires :

Concernant la répartition des taches lors de ce projet, Claire s'est chargé de la traduction du fichier assembleur, Abde-Rrahman s'est occupé de la plupart des fonctions auxiliaires et Nathan a réalisé la simulation, bien que nous nous soyons tous entraidés pour nos taches respectives, surtout à la fin pour pouvoir tout débuger.