Développement et retour:

Après de nombreux tests et fastidieuses séances de debugs, nous sommes fiers de vous présenter notre rapport de projet. Sans tout de même croire que l'on peut à présent créer notre propre language de programmation et succéder à Haskell Curry, nous avons quand même eu une agréable expérience et profondément saisi les différents concepts liés à l'assembleur et aux registres, et avons eu une certaine conceptualisations de la création des divers IDE que l'on peut utiliser.
Notre programme effectue donc deux tâches ; la première correspond à la traduction du langage assembleur en langage hexadécimal. La deuxième tâche correspond à la simulation de ce programme après l'avoir récupéré en langage machine.

Étape 1 : Traduction du langage assembleur en hexadécimal

La première étape de ce projet correspond à la conversion du code assembleur en fichier hexadécimal.
Pour ce faire, nous avons pensé dès le départ au cas où le fichier pourrait contenir des sauts de lignes. Étant donné qu'il faut les ignorer, nous avons créé un nouveau fichier qui les enlève à l'aide de la fonction « removeEmptyLines ». Pour implémenter cette dernière nous avons également eu besoin de la fonction « isLineEmpty « qui elle, retourne 1 si la ligne est vide et donc doit être enlevée.
Ensuite, on vérifie à l'aide de notre fonction checkSyntax les cas d'erreurs de syntaxe qu'un utilisateur peut insérer. Nos deux principales difficultés étaient d'abord de changer la manière avec laquelle on récupérait les chaînes de caractère, tantôt il s'agissait de fscanf, tantôt c'était avec un fgets suivi d'un strtok. Puis, on est resté ahuris pendant une heure, car on avait simplement oublié de rajouter le \r au niveau des séparateurs !
Ensuite, à partir de ce fichier « propre », nous commençons la conversion hexadécimale. La fonction « translatetohexa » nous permet donc de le faire. Pour cela, la création de fonctions intermédiaires tels que « DownWeGo » était indispensable.
Cette dernière parcourt le fichier assembleur en entier pour stocker l'ensemble des étiquettes. Il est primordial de différencier les étiquettes des autres instructions lors de la conversion car le fichier hexadécimal n'affiche pas les étiquettes. Pour cela nous avons créé une structure Node :

```
typedef struct Node{
int address;
char name[20];
struct Node* next;
struct Node* prev;}Node;
```

Cette structure prend comme attributs le nom de son étiquette ainsi que l'adresse à laquelle elle se trouve, ce qui sera utile plus tard. Elle prend également des pointeurs sur les éléments suivants et précédents pour pouvoir insérer les nouvelles étiquettes au fur et à mesure du parcours du fichier assembleur. La fonction « InsertendNode » nous permet donc de le faire. Le parcours du fichier se fait donc à l'aide de la fonction fgets et strtok à l'aide desquels on récupère la première chaine de caractères de chaque ligne . On peut ainsi vérifier s'il s'agit bien d'une instruction ou non à l'aide de la fonction « verifyinstruction »qui retourne 1 dans le cas où une chaine de caractère correspond bien à une instruction.
Ainsi l'ensemble des étiquettes du programme sont stockées dans une liste.
Après ceci, nous entamons la lecture du fichier depuis le début grâce à la fonction « rewind » et nous commençons la traduction. Pour cela nous prenons bien soin d'évaluer chacun des cas pouvant se présenter.
Nous bouclons ainsi sur l'ensemble des lignes du fichiers calculées à l'aide de la fonction « height ». Pour chaque ligne, nous commençons par lire la première chaine de caractères grâce à la fonction dédiée à la lecture de fichier « fscanf ».Si cette dernière correspond à une instruction nous la convertissons en hexadécimal. En effet, nous avons créé une fonction « hexavalue » retournant l'entier correspondant à chaque instruction. Étant donné que l'instruction se présente sous 1 octet nous l'écrivons sur le fichier à l'aide de la fonction « fprintf » et « "%.2x " » qui nous permet de faire la conversion hexadécimal sur 1 octet.
Ensuite nous venons à tester si cette instruction admet naturellement une donnée ou non. Pour cela nous avons implémenté une fonction «hasdata» nous permettant de savoir en fonction de la valeur de retour si la prochaine chaine de caractère à lire correspond à un nombre ou une chaine de caractère (dans le cas des « jmp », « jpc » , « call »).
Ainsi si il s'agit d'un nombre on le convertit en hexadécimal de la même façon mais cette fois ci sur 2 octets avec "%.4x ".Dans le cas où il s'agit d'une chaine de caractère, nous faisons appel à la fonction « findetiquettevalue ». Elle nous retourne l'adresse de l'étiquette correspondante afin de calculer la nouvelle adresse à laquelle nous devons nous déplacer.
Encore une fois on traduit cette dernière en hexadécimal à l'aide « %.4x ».
Bien sûr dans le cas où la chaine de caractères de départ ne correspond pas à une instruction, nous testons qu'il s'agit bien d'une étiquette que nous repérons à l'aide des « : ».
Si c'est le cas, nous continuons la traduction de la ligne comme nous l'avions fait dans le cas précédent.
Après avoir parcouru l'ensemble des lignes de notre fichiers , nous obtenons finalement bien notre traduction hexadécimale!

Étape 2 : Simulation de la machine à Pile

Vient maintenant la simulation de notre programme à partir du fichier en hexadécimal.

Tout d'abord, nous avons commencé par récupérer les données du fichier en séparant les instructions des données dans deux tableaux , instructionab[] et datatab[] grâce à la fonction « filltab » et « "%x " » nous permettant ainsi de stocker les hexadécimaux en entiers. L'indice des tableaux pour la donnée lié à une instruction est donc le même.
Il fallait tout de même faire attention au cas où l'hexadécimal correspondait à un entier négatif et gérer ce cas lors de son stockage dans le tableaux de données.
Nous avons eu également besoin de créer un tableaux de 4000 adresses qui jouera le rôle de la mémoire. Les indices de ce tableaux correspondront donc aux adresses mémoires des éléments à ces indices. Ainsi la pile correspondra à une partie de notre mémoire commençant à l'adresse zéro. Il sera donc possible de la modifier tout au long du programme à l'aide du pointeur SP, le pointeur de Pile indiquant le premier emplacement libre sur la pile.

Ensuite viens l'implémentation des fonctions correspondant aux différentes instructions prédéfinies (« op, push, pop, jpc »…). Bien sûr, nous arrêtons toute simulation dans le cas où il pourrait y avoir un accès à une zone mémoire extérieure aux dimensions prédéfini(0 à 3999) .Cela est donc visible à travers les « printf » effectués sur chacun de ces cas. Cela évite ainsi au programme de planter.
Nous pouvons à présent maintenant commencer à exécuter notre programme. Pour cela nous commençons à PC=0 et SP =0.
Ainsi pour chaque indice (représenté par PC) de nos tableaux nous faisons appel à notre fonction «execute » récupérant notre instruction et notre donnée et faisant donc appel à la bonne instruction du programme à exécuter. Encore une fois nous prenons soin d'arrêter le programme dans le cas ou PC pourra prendre un valeurs extérieur au dimensions de notre tableaux d'instruction .
Ceci sera donc fait jusqu'à ce que le programme reconnaisse l'instruction « halt « annonçant la fin du programme.

Erreurs et difficultés rencontrées :

Concernant la conversion en hexadécimal, la principale difficulté aura été de gérer le cas des étiquettes. En effet ,il fallait les reconnaître et les stocker pour les utiliser plus tard . Après mûre réflexion, la solution la plus astucieuse aura été de commencer par les stocker avec leurs adresses dans une liste. Ceci nous a énormément aidé pour la suite.
Nous avons aussi dû forcer la conversion de nos valeurs numériques pour les adresses afin de les écrire sur 2 octets à l'aide du %04hx.
Le plus difficile résidait également dans le détail. En effet ne pas gérer l'ensemble des cas particuliers pourrait s'avérer fatal pour la suite du programme. Notre programme utilisant les « scanf » pouvait être handicapant lors de la gestions de ces cas particuliers. Ce qui aurait peut-être été plus simple à gérer avec des « fgests » et

« strtok ». C’est donc pour cela qu’il a fallu créer une nouvelle fonction spéciale utilisant les dernières fonctions mentionnées gérant donc les cas tel que : une instruction sans donnée par exemple.
Lors de la simulation, nous avons rencontré à plusieurs reprises le même segmentationfault : « global buffer overflow ». C’est à partir ce moment que nous avons fait attention à la gestion de notre mémoire en testant à chaque fois qu’il n’y avait pas d’accès mémoire hors dimensions définies. En effet, il n’est pas possible de faire un pop sur une pile vide ou encore un push sur une pile pleine.

Notre projet ainsi que l’ensemble des enseignements du semestre nous auront appris l’importance de l’organisation interne de l’ordinateur : négliger ou ignorer sa structure amènera tôt ou tard à des déconvenues et des erreurs.