Marion Fillaudeau et Ekaterina Galkina
Groupe 5

Simulation d’une machine virtuelle

Sommaire

Introduction / méthode de travail

Afin de mettre en pratique notre machine virtuelle, nous avons principalement travaillé côte à côte dans le but de pouvoir s'entraider pour comprendre les différentes subtilités du problème et comment les mettre en place. Par ailleurs, ce travail en groupe nous a également appris à travailler avec Git et GitHub pour la gestion des versions.

Avant toute chose, nous avons réfléchi ensemble sur le déroulement théorique de notre programme (De quelles fonctions allons-nous avoir besoin ? Comment traduire une ligne du fichier texte en une instruction en hexadécimal ? Comment structurer notre processeur virtuel ? Comment décoder une instruction en hexadécimal ? Comment stocker les instructions en mémoire ? etc...)

Nous avons décidé de débuter par l'écriture du code qui s'occupera de la traduction de langage d'assembleur en langage machine et essayer de déjà gérer tous les bugs possibles. Cette partie a d'abord été traitée en nous appuyant sur l'exemple *inversion.txt* donné dans le sujet, puis une fois cet exemple bien compris nous avons essayé d'envisager les erreurs de syntaxe qui pourraient se trouver dans les programmes donnés lors d'un appel à notre simulateur. Ainsi, dès que nous commencions à coder, pour essayer de limiter la possibilité que notre programme tombe en panne et d'indiquer avec une précision les erreurs possibles à l'utilisateur, on notait toutes les erreurs possibles, les cas particuliers à gérer dans un fichier pour ne pas les oublier et y revenir après. Puis, nous avons partagé ces petits problèmes de la manière la plus équitable possible, pour les résoudre et bien tester.

N.B : Dans le code source, nous avons commenté la majorité des fonctions/variables, mais si vous utilisez un environnement de codage, vous pouvez également obtenir des informations rapidement sur une fonction/variable en la survolant avec la souris.

Partie 1 : Génération du fichier en langage machine

1. Les difficultés rencontrées

La partie la plus compliquée, à notre avis, était celle-ci. Il fallait s'assurer de gérer toutes les issues possibles, c'est-à-dire de prédire tous les inputs d'utilisateur et comment les interpréter. Comme dit plus haut, nous avons passé beaucoup de temps à gérer tous ces cas. Nous les avons traités plus ou moins indépendamment, ce qui peut expliquer la lourdeur et la longueur de notre fonction traduction.

2. Fonctionnalités

L'utilisateur se servant de notre programme doit avant tout lire la documentation rédigée. Nous avons essayé d'indiquer avec le plus de précision possible le type des erreurs de syntaxe rencontrées. Les cas traités sont pour l'instant :

- Le programme n'autorise pas à l'utilisateur de passer une ligne vide;
- Les étiquettes doivent respecter le format indiqué (lettres et chiffres, et deux points immédiatement après), ne peuvent être utilisées que dans certaines fonctions spécifiques;
- Il ne peut y avoir deux étiquettes portant le même nom;
- L'utilisateur peut mettre autant d'espaces qu'il veut entre les valeurs passées;
- Le programme contrôle les noms de commandes et indique si un nom de commande n'existe pas;
- Nous vérifions si une ligne est bien formatée, entre autres la présence ou non de virgules et parenthèses;
- Le programme contrôle et indique si le nombre d'arguments passés ne correspond pas à celui la fonction appelée, ou si ceux-ci sont mal positionnés (notamment les valeurs immédiates);
- Les valeurs passées en paramètres peuvent être écrites en décimal, hexadécimal et peuvent être négatives et positives;
- Pour la majorité des erreurs, l'utilisateur peut voir des indications avec les suggestions de correction de la ligne où il y a un problème.

Pour construire, le fichier *hexa.txt*. La structure de notre programme est la suivante :

1. Nous récupérons d'abord toutes les étiquettes présentes dans le fichier, et vérifions leur syntaxe et la présence de doublons.
2. Dans le même temps, nous stockons dans un tableau le contenu de toutes les lignes
3. Nous parcourons le tableau de lignes et pour chaque ligne, nous appelons la fonction traduction. Cette fonction modifie l'élément du tableau passé par adresse en argument pour y stocker la valeur numérique de la ligne ainsi traduite. La fonction traduction renvoie 1 ou 0 pour signaler une erreur sur la ligne.
4. Une fois avoir traduit toutes les lignes, si aucune erreur n'a été détectée sur l'ensemble du fichier, on peut lancer l'écriture dans le fichier *hexa.txt.*

3. Les améliorations envisageables

Nous avons probablement omis certains cas lors du listing de toutes les erreurs possibles, il faudrait nous assurer que celles-ci soient correctement gérées.

La fonction "*traduction*" de cette première partie, qui stocke la valeur de la ligne dans un tableau, pourrait être découpée en plusieurs fonctions pour alléger le code, dans la mesure où certains passages se répètent au sein de la fonction. Cependant nous avons préféré nous concentrer sur la bonne gestion des erreurs et avancer sur la seconde partie du projet (au détriment de la légèreté du code).

Partie 2 : Exécution du programme

1. Les difficultés rencontrées

Nous avons tout d'abord construit assez rapidement les petites fonctions exécutant les différentes instructions. Puis en nous penchant dessus plus attentivement, nous nous sommes rendu compte de certaines erreurs qui pourraient arriver avec certaines valeurs négatives et/ou limites. Nous avons donc dû ajouter quelques lignes de code pour s'assurer de la bonne représentation de chaque entier.

De plus, lors de notre première version du code simulant l'exécution nous n'avions pas choisi les bons types pour nos variables (par exemple, nos registres étaient du type int, et après chaque fonction nous nous assurions qu'ils soient dans l'intervalle donnée avec une fonction *int216bits*). Cependant, nous avons décidé de conserver la fonction *int216bits* pour être sûres de conserver les bons intervalles et garantir la portabilité.

Nous avons également passé un certain temps à finaliser les fonctions pour les instructions de transfert (ldb, ldw, stb et stw). D'abord, nous avons mis du temps à comprendre leur fonctionnement notamment l'inversion des octets de poids faible et forts, dans le cas de ldw et stw. De plus, elles posaient un problème lorsque nous essayions de passer en valeur immédiate des valeurs négatives, et il était plutôt difficile de les tester. Après une révision nous avons trouvé l'erreur.

L'autre difficulté majeure de cette deuxième partie résidait dans les tests : toutes les petites fonctions devaient être bien vérifiées pour qu'elles puissent donner des résultats attendus. Et bien sûr, les tests de l'entièreté du programme depuis l'algorithme passé en langage assembleur jusqu'à son exécution. Pour effectuer ceci, nous nous sommes inspirés des exercices proposés en travaux pratiques et ceux des sujets d'examen d'architecture des ordinateurs, mais aussi des programmes inédits pour vérifier des choses plus spécifiques telles que l'accès à la mémoire.

Nous traduisions ces exercices dans le langage assembleur du projet pour ensuite les tester avec notre traducteur en langage machine puis simulateur.

2. Fonctionnalités

Pour cette partie, voici le déroulement de notre code :

1. Nous récupérons d'abord le fichier *hexa.txt* puis le stockons dans notre mémoire grâce à la fonction *programInMem*. Au début de la mémoire se trouvent donc nos instructions : 4 cases mémoires représentent 1 instruction. Nous stockons donc 2 caractères hexadécimaux dans chaque case.
2. Nous commençons l'exécution avec PC = 0. Pour exécuter une instruction à l'adresse i on appelle *ligne* en passant différents pointeurs en arguments : un vers la case i de la mémoire, un pour charger le code opération, un pour charger la destination, un pour charger la source 1, un pour charger la présence d'une valeur immédiate et un pour charger la source 2. La fonction se charge ensuite de modifier ces différentes variables pour pouvoir les utiliser dans la commande appelée.
3. Grâce à un tableau de pointeurs vers des fonctions on peut appeler la fonction correspondant au code opération. La fonction appelée va effectuer son opération et va mettre à jour si besoin le registre d'état.
4. On continue ainsi le déroulement du programme en se laissant guider par le registre PC, dès qu'il est nul, on stoppe l'exécution.

Dans cette partie du projet, des erreurs ne pouvant pas être décelées à la compilation (traduction du fichier) peuvent aussi survenir. Nous les avons traitées de la manière suivante :

- La division par 0 met évidemment un terme à l'exécution du programme
- L'instruction "in" ne donne pas l'occasion de stocker autre chose qu'un entier en valeur décimale.
- Si l'adresse d'un saut est une valeur immédiate qui n'est pas valide (pas un multiple de 4) alors l'exécution du programme se termine.
- Nous signalons si l'utilisateur tente d'accéder (avec "ldb" ou "ldw") à la portion de la mémoire où sont stockées les instructions du programme, s'il demande de stocker (avec "stb" ou "stw") une valeur dans une case mémoire occupée par une instruction alors le programme s'arrête pour motif d'erreur de segmentation.

Annexe : Exemples de programmes en langage machine

Nous mettons à disposition certains programmes que nous avons conçus pour tester notre machine virtuelle. Certains de ces fichiers sont disponibles dans le fichier tests du dossier FillaudeauGalkina.

- *Exemple 1.* Test des instructions de transfert.

```
add r5, r0, #-513
add r6, r0, #500
stw (r6)#0, r5
ldw r9, (r6)#0
out r9
ldb r7, (r6)#0
out r7
ldb r8, (r6)#1
out r8
add r4, r0, #900
add r10, r0, #255
add r11, r0, #253
stb (r4)#0, r10
stb (r4)#1, r11
ldw r30, (r4)#0
out r30
hlt
```

.

- *Exemple 2.* Boucle for (int i = 0; i<=19; i++) { printf(i);}

```
add r7, r0, #0
add r8, r0, #20
loop: out r7
add r7, r7, #1
sub r9, r7, r8
jnc fin
jmp loop
fin: hlt
```

- *Exemple 3.* Affichage des valeurs de 0 à 20 en double. Principalement, pour tester la robustesse de notre traducteur.

```
add r7   , r0, #0
101     0001    add r8, r0   , #20
add r5, r0, #52
loop: out r7
        stb (r7)   r5, r7
ldb r10, (r7   )  r5
  out r10
add r7, r7, #1
1001sub r9 , r7,  r8
jnc fin
jmp   loop
fin: hlt
```