Bobo Jules
Maozhen Fan

Notice Développeur

Sommaire

Introduction 2

Fonctionnement général 2

1. Traducteur 2
    a. Les structures

    b. méthode et description des fonctions
        i. les étiquettes
        ii. isolation des instructions
        iii. récupération du code opératoire et des numéros de registre

    c. Gestion des erreurs

2. Interpréteur 5
    a. Les différents types de listes
    b. Méthode et description des fonctions
        i. L'exécution du programme
        ii. Les instructions

Amélioration possibles 8

Répartition du travail 9

Bobo Jules
Maozhen Fan

Introduction

Notre programme lit un fichier d'instruction assembleur donné par l'utilisateur et le traduit en hexadécimal dans un fichier hexa.txt avant d'exécuter les instructions.

Fonctionnement général

1. Traducteur

a. Les structures

On utilise deux structures :

- Instruction : sert à stocker les instruction et contient le nom de l'instruction, une liste des arguments, le nombre d'argument ainsi que un nombre 0 ou 1 désignant s'il y a une valeur immédiate
- Étiquette: sert à stocker les étiquettes, contient le mot désignant l'étiquette et la ligne à laquelle elle fait référence

b. Méthode et description des fonctions

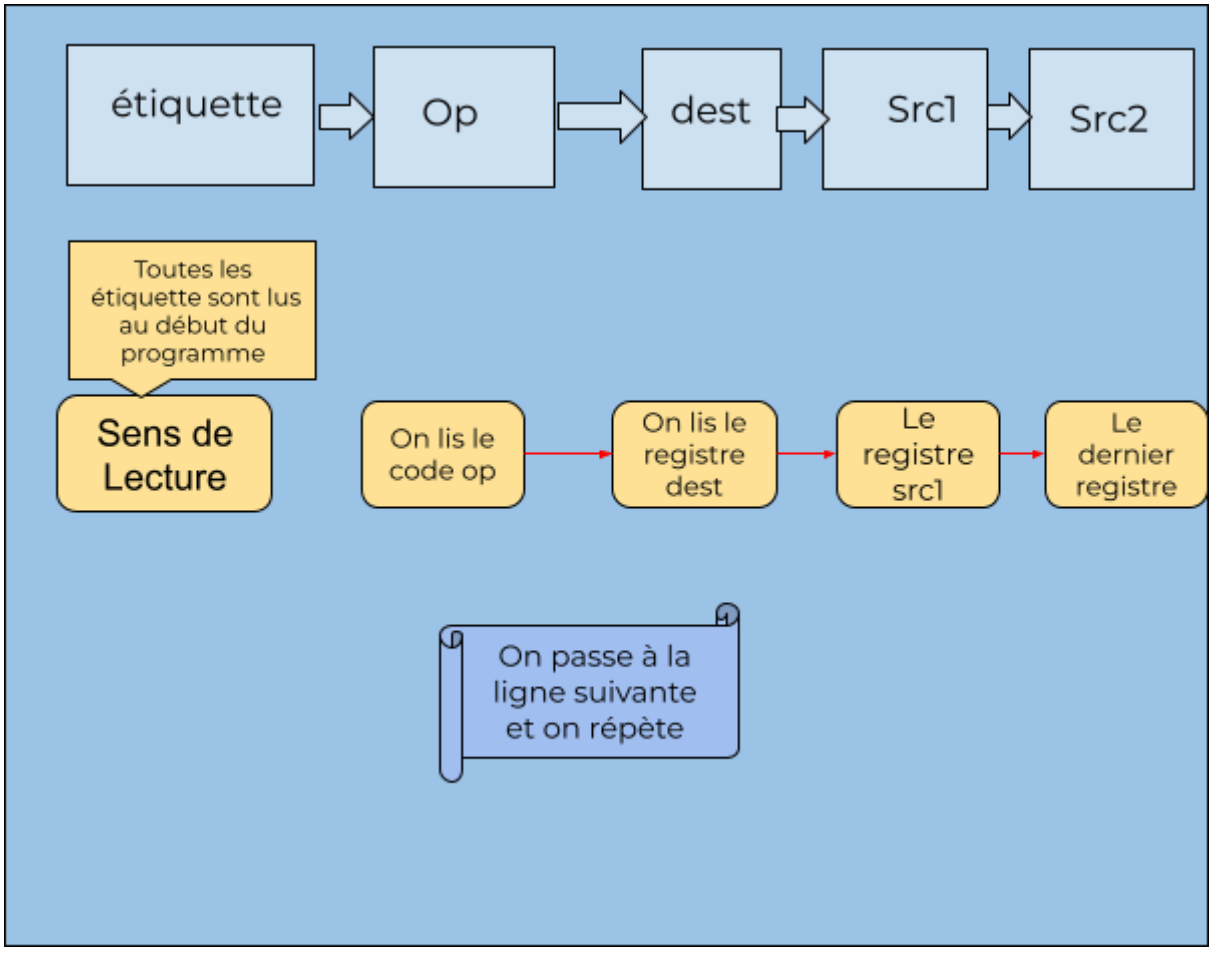
Voici comment on a effectué la traduction du document:

chaque ligne est de la forme
etiquette: nom : dest src1 S (la présence de l'étiquette et des 3 arguments varie)

1) On lit le fichier ligne par ligne, grâce à notre fonction ***lecture fichier*** appelant ***lecture ligne*** à chaque ligne, on vérifie alors s'il y a une étiquette en regardant la présence du caractère ':' et la stockant dans le cas échéant dans un tableau avec ***etiquettechr*** les caractères avant le tiret. On enregistre alors le reste de la ligne dans un tableau de ligne et on passe à la ligne suivante.

2) On cherche ensuite à isoler les instructions et leurs arguments.
On appelle alors notre fonction ***LigneVersInstruction*** qui transforme nos lignes en instructions. On fonctionne alors ligne par ligne sur tout le document, on stocke le nom de l'instruction puis on sépare les argument on en les stocke grâce à strtok. S'il y a un # On incrémente à 1 l'indice indiquant que la valeur de S est immédiate et on note le nombre d'arguments. On stocke toutes ces instructions dans un tableau.

Schéma récapitulatif du processus :



3) On récupère ensuite le code opératoire associé au nom de l'instruction avec ***Codeop.***

   On récupère le numéro du registre destination et des sources avec nos fonctions ***recupRegistreDest***, ***recupSrc1***, ***recupSrc2*** ,s'il y a une valeur immédiate on la récupère en entier décimal ou hexadécimal selon le cas. On fait alors un décalage de chacun de ces nombre et on les somme de façon à avoir le code sur 32 bits, on décale le nom de 27, dest de 22, src1 de 17, et l'indice immédiat de 16. On les somme avec src2 et on écrit dans le document hexa.txt ce code convertit en hexadécimal.

   On a des cas particulier suivant le code opératoire pour associer l'argument de les arguments de l'instruction avec les bon registre, puisque l'ordre et la présence des argument est variable selon l'opération effectuée.
   Dans le cas des saut avec étiquette on retourne la ligne associée à l'étiquette multipliée par 4.

c. Gestion des erreurs

C'est ici la partie la plus complexe de la traduction, vérifier que les instructions sont "légals", renvoyer le message de l'erreur provoqué ainsi que la ligne correspondante.
On a pour chacune des étapes de la traduction des fonctions pour vérifier l'intégrité des inputs :

- D'abord on vérifie qu'un fichier à été importé et qu'il n'est pas vide , pour cela on vérifie qu'un argument a été donné et que notre fonction nbligne donnant le nombre de ligne, utile lors de la lecture de fichier pour allouer la mémoire nécessaire, ,ne rend pas 0.

- cas des étiquettes : On vérifie que l'on a pas de doublon avec ***testetiq*** et on vérifie qu'elle est non vide.

- cas du nom de l'instruction: On vérifie dans ***Codep*** la correspondance de l'opération avec celles que l'on a implémenter, si elle n'existe pas on le signal.

- Cas des registres: On effectue des vérifications à chaque fois que l'on récupère un registre.
  On vérifie que le registre existe avec ***CheckRegistre*** (donc si on a r+nombre entre 0 et 31)
  On vérifie que le nombre d'arguments donné s'accorde avec le nombre attendu.

On vérifie pour S que la valeur immédiate est bien un nombre décimal ou hexadécimal selon le cas avec ***CheckvaleurImmediate***
Dans le cas des saut on vérifie aussi que l'étiquette associée à l'argument existe.

Puisque l'on fonctionne ligne par ligne il n'y a pas de difficulté à renvoyer la ligne d'erreur, c'est l'itération où l'on est. On écrit aussi un message d'erreur personnalisé selon l'erreur que l'on vient de détecter.

2. Interpréteur

a. Les différents types de listes

Utilisation de deux types de listes :

- Liste de **char** : Nous avons deux listes de char, l'une de taille 65536 et l'autre de taille 3. Ces deux listes servent à simuler la mémoire de la machine et à représenter les bits des états, car la mémoire de la machine est en octets et un char est de 1 octet.
- Liste de **short** : Elle sert à simuler les registres de 16 bits car un short est de 2 octets.

Ces deux choix facilitent nos tâches.

b. Méthode et description des fonctions

1) Voici comment nous avons effectué l'exécution du document :

Chaque instruction est codée sur 32 bits, et on les répartit en 5 codes : code opération,registre destination, premier registre source, immédiate et second registre source. De plus, le code immédiat nous permet de savoir si le second registre est une valeur immédiate ou un registre. Ensuite, on les stocke dans **une matrice de dimension nombre de lignes * 5** (un tableau de int de dimension nombre de lignes * 5), puis on parcourt toutes les lignes de la matrice et lit le code opération pour savoir quelle instruction la machine va exécuter et les autres codes nous donnent les valeurs dont on a besoin pour effectuer l'instruction. L'adresse de l' instruction est représentée par **le numéro de ligne de la matrice**.

2) Description des fonction:

a) Le code opération de 0 à 7 indique les instructions arithmétiques et logiques : **add, sub, mul, div, and, or, xor, shl.**

La fonction **add** : Elle effectue une addition entre les valeurs du premier registre source et du second registre, puis on la stocke dans le registre destination.

La fonction **sub** : Elle effectue une soustraction entre les valeurs du premier registre source et du second registre, puis on la stocke dans le registre destination.

La fonction **mul** : Elle effectue une multiplication entre les poids faibles de la valeur du premier registre source et du second registre grâce à la fonction **PoidsFaible** et **ConvertirBinaire**, puis on la stocke dans le registre destination.

1) La fonction **ConvertirBinaire** prend une valeur et la convertit en binaire représenté par une chaîne de caractères.

2) La fonction **PoidsFaible** prend une valeur et donne ses poids faibles grâce à la fonction **ConvertirBinaire**.

La fonction **div** : Elle effectue une division entre les valeurs du premier registre source et du second registre, puis on la stocke dans le registre destination.

Les fonctions logiques **and**, **or**, **xor** : Elles effectuent des opérations logiques entre les valeurs. On convertit les valeurs en binaire grâce à la fonction **ConvertirBinaire**, puis on effectue l'opération logique correspondante pour tous les bits et on la stocke dans le registre destination.

La fonction **shl** : Elle effectue des décalages de bits par rapport au signe du second registre source. Si le second registre est positif, alors on effectue les décalages à gauche, sinon à droite et les nouveaux bits sont remplacés par 0. Pour le cas particulier où on devrait effectuer un décalage à droite pour une valeur négative, on peut utiliser la fonction **DecalageDroiteNeg**. On la stocke ensuite dans le registre destination.

- La fonction **DecalageDroiteNeg** permet de faire le décalage à droite pour la valeur et remplace les nouveaux par 0.

Toutes ces fonctions prennent 4 arguments : **registre destination, premier registre source, immédiate, second registre source.**

b) Le code opération de 10 à 13 indique les instructions de transfert de données : **ldb, ldz, stb, stw.**

La fonction **ldb** : Elle effectue un transfert de données de 1 octet. On transfère la valeur de l'adresse mémoire **premier registre source + second registre source** au registre destination.

La fonction **ldz** : Elle effectue un transfert de données de 2 octets. On transfère la valeur de l'adresse mémoire **premier registre source + second registre source** au registre destination comme les poids faibles de la valeur du registre destination, puis on transfère la valeur de la mémoire **premier registre source + second registre source + 1** au registre destination comme les poids forts de la valeur du registre destination.

La fonction **stb** : Elle effectue un transfert de données de 1 octet. On transfère les poids faibles de la valeur du registre destination à l'adresse mémoire premier registre source + second registre source.

La fonction **stw** : Elle effectue un transfert de données de 1 octet. On transfère les poids faibles de la valeur du registre destination à l'adresse mémoire **premier registre source + second registre source**, puis on transfère les poids forts de la valeur du registre destination à l'adresse mémoire **premier registre source + second registre source + 1**.

Toutes ces fonctions prennent 4 arguments : **registre destination, premier registre source, immédiate, second registre source.**

c) Le code opération de 20 à 26 indique les instructions de sauts : **jmp, jzs, jzc, jcs, jcc, jns, jnc.**

La fonction **jmp** : Elle renvoie l'adresse de saut, plus clairement le numéro de la ligne de la matrice qui stocke tous les codes d'instructions.

Les fonctions **jzs, jzc, jcs, jcc, jns, jnc** : Elles renvoient l'adresse de saut selon certaines conditions spécifiques.

d) Le code opération de 27 à 28 indique les instructions d'entrée-sortie : **in, out.**

La fonction **in** : Elle permet à l'utilisateur d'entrer une valeur et la stocke dans le registre destination.

La fonction **out** : Elle permet d'afficher la valeur du registre destination.

e) Le code opération de 29 indique l'instruction diverse : **rnd**

La fonction **rnd** : Elle permet d'obtenir une valeur aléatoire entre la valeur du premier registre et de la valeur du second registre - 1.

f) Les fonctions qui permettent de mettre à jour les bits Z et C : **bitZ** et **bitC**.

g) La fonction **verifieLimites** permet de savoir si le résultat pourra bien être représenté par 16 bits.

h) La fonction **verifierEntier** permet de savoir si la valeur est un entier, donc on l'utilise pour vérifier si la valeur du second registre est un entier.

i) La fonction **verifierMemoire** permet de verifier si l'adresse de mémoire indiquée est correcte. Mais finalement on n'a pas besoin de cette fonction car les valeurs sont en 16 bits.

La nature de second registre source dépend du code immédiat:
si le code immédiat vaut 1 alors le second registre source est une valeur immédiate sinon il représente un registre

Améliorations possibles

Malheureusement notre gestion des erreurs n'est pas sans faille. Certain cas très particulier qui devrait afficher des erreurs ne le font pas et compile sans problème, c'est dû à la façon dont les argument sont séparer par strtok : les virgule et parenthèse sont vu comme des espaces et donc passé par le programme qui fait que certaines instruction de la forme suivant ne pose pas de soucis:

add r1,,,,r3,,,r4,
add r1,(r2),r5

Le programme peut être modifié pour résoudre ces soucis mais il faudrait réécrire la fonction ***LigneVersInstruction*** en profondeur . L'utilisation de strtok pour gagner du temps a donc créer des failles, bien que l'on soit au courant de ces

erreurs puisqu'elles sont très isolées et ne font pas planter le programme nous le laissons ainsi.

On a tout de même incorporé quelques sécurité pour éviter des aberrations comme:
add (r1),(r2),r3 en comptant que le nombre de () ne dépasse pas 2, de même avec #

On aurait aussi pu incorporer un système de code d'erreur de fonction affichage d'erreur pour simplifier le programme et séparer les parties car c'est ici mélangé et brouillon.

Enfin il n'y a pas la gestion de la mémoire malloc-free puisque ce n'est pas demandé dans le sujet mais les fonctions pour la libération sont dans le programme, on ne les utilise pas mais ce ne demanderait pas tant d'effort que de les incorporer dans le programme.