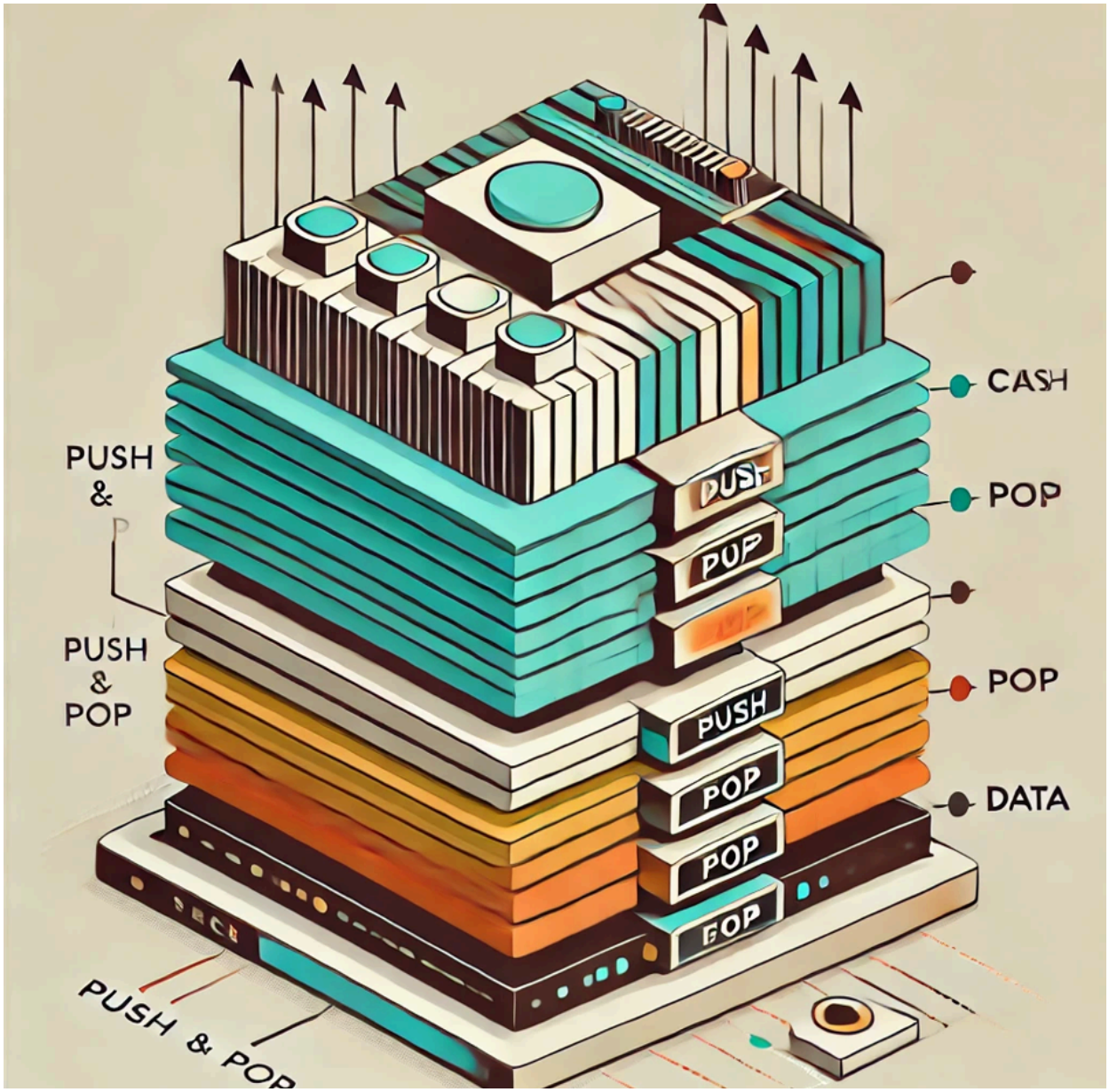

Machine à pile

11.02.2025

Samuel Nonon

Joshua Lozano

TD4

Répartition du travail

Dans ce projet, nous devions simuler une machine à pile. Pour cela, nous avons séparé le travail en deux. Joshua s'est occupé de la première partie récupérer le fichier, le traduire et regarder les différentes erreurs de compilations possibles et Samuel a créé les différentes fonctions permettant l'exécution de la machine à pile, en traitant les différentes erreurs d'exécution.

Objectifs

1. Pour récupérer le fichier je m'étais fixé plusieurs objectifs, le faire proprement, éviter au maximum de faire un code redondant et bien sûr de faire en sorte que cela soit le plus efficace possible (cf. partie 1 : récupération du fichier et traduction).
2. Pour l'exécution, après avoir récupéré les instructions en hexadécimal dans le fichier « hexa.txt », le but était d'implémenter les différentes fonctions en testant les erreurs possibles, afin de renvoyer un message d'erreur plutôt que notre programme ne marche pas ou renvoie un résultat incohérent.

Rapport

I. Récupération du fichier et traduction

Récupération des instructions, arguments et étiquettes

Pour récupérer le fichier, deux choix se sont offerts à moi, le récupérer ligne par ligne ou mot par mot. J'ai choisi ligne par ligne pour plusieurs raisons, je trouvais cela bien plus pratique pour délimiter chaque instruction et ses arguments avec l'utilisation de strtok. En effet, toutes les instructions ne prenant pas le même nombre d'argument, la présence d'une étiquette m'aurait compliqué la tâche car je n'aurais pas pu déterminer avec certitude où était la fin de la ligne et à quoi chaque chaîne de caractères correspondait.

Pour illustrer, si on a "fin: halt" le premier mot est une étiquette, le deuxième l'instruction et il n'y a pas d'argument, si la ligne suivante est "read 1000" le premier mot est l'instruction et le deuxième l'argument. Je trouvais que cela portait à

confusion et ne pas pouvoir déterminer avec certitude où la ligne se finissait me déplaisait. J'ai utilisé "fgets" pour récupérer la ligne complète et pouvoir la découper en plusieurs morceaux.

Pour cela, j'ai utilisé la fonction prédéfinie "strtok" qui permet de découper la chaîne en morceaux à partir d'un indicateur qu'on lui donne comme un espace (ce que j'ai utilisé). La prise en main de cette fonction fut assez compliquée au premier abord, car elle modifiait aussi la chaîne de caractère initial et me renvoyait souvent des segment-fault. Mais après plusieurs tests pour bien comprendre son utilisation, elle m'a été très utile.

Je commence en premier lieu par récupérer les étiquettes s'il y en a. Si l'étiquette et l'instruction sont séparées d'un espace, je regarde si le dernier caractère du premier "token" pris par "strtok" se finit par deux points si c'est le cas je stocke l'étiquette dans un tableau étiquette au numéro de sa ligne: `strcpy(etiquette[ligne], etiq);` sinon, je regarde si l'étiquette est collée à l'instruction grâce à une fonction annexe "etiquette_collée" qui retourne la position de la fin de l'étiquette dans la chaîne de caractère ou -1 si il n'y a pas d'étiquette collée. Si l'étiquette est collée, je récupère l'étiquette et la stocke dans le tableau étiquette, je récupère l'instruction et la place dans une structure créée au préalable.

Pour les instructions j'ai décidé de créer une structure "instr" pour les stocker temporairement. Ce n'est pas ce que j'avais fait dès le début. Initialement, je me contentais de récupérer la ligne et de la mettre dans une fonction "convers_machine" (plus présente dans le code) très longue et très répétitive qui effectuait une série de tests afin de vérifier si l'instruction ainsi que son argument étaient corrects. Cependant, la fonction était extrêmement longue et répétitive. De plus, cette fonction n'était pas du tout pratique pour le calcul d'adresse des étiquettes. J'ai refait une fonction plus compact et concise que j'ai directement intégrée dans "recup_fichier" qui est la fonction actuelle, et j'ai créé une structure me permettant de stocker temporairement l'instruction et l'argument de la ligne.

Cette alternative m'a réellement permis d'aller bien plus vite et d'avoir un code bien plus efficace. Mon premier code était tellement long, rempli de fonctions annexes, d'astuces pour récupérer les instructions/arguments/étiquettes que je m'y perdais et n'arrivais plus à avancer.

Gestion des erreurs et calcul d'adresse

Pour la gestion des erreurs, j'ai essayé d'identifier le maximum d'erreurs possibles. J'ai aussi demandé à chatgpt de me générer un code similaire à celui donné par M.Lazard pour être sûr de n'oublier aucune erreur. Pour cela, j'ai divisé les instructions en trois

différents types : celles qui ne demandent pas d'arguments, celle qui en demandent,et celle qui pourraient nécessiter un calcul d'adresse.

Pour les instructions qui ne demandent pas d'argument (halt, ret, dup...), les vérifications étaient plus faciles : j'ai vérifié que l'instruction était valide et qu'il n'y avait pas d'argument.

Pour les instructions qui demandent des arguments (read, op, pop...), j'ai vérifié que l'instruction était correcte, qu'il y avait bien un seul argument et que cet argument était valide. J'ai aussi veillé à ce que l'argument ne sorte pas de la mémoire et pour op qu'il appartienne bien à l'intervalle [0, 15].

Enfin, pour les instructions qui demandent un calcul d'adresse, cela devient plus délicat, j'ai d'abord récupéré les étiquettes dans un tableau puis dans un second temps calculé les adresses si elles devaient être calculées. Un problème logique arrive dans le cas où une étiquette n'est pas valide et qu'il y a une autre erreur dans le code, la deuxième erreur sera renvoyée car le calcul d'adresse se fait dans un second temps. Pour illustrer cela, voici un exemple (le programme n'a pas de sens):

```
ici: read 1000
push# 0
op 0
jnz fini  // problème ici étiquette inconnue
push -1000 // push ne prend que des entiers positifs
op 15
pop 1000
jmp ici
fin: halt
```

Dans ce cas là, l'erreur renvoyée sera celle de push et ensuite, si elle est corrigée celle de l'étiquette. J'ai ce problème car je ne peux pas calculer les adresses des étiquettes directement, je parcours le fichier ligne par ligne et si il y a une étiquette dans un jmp/jnz/call qui n'est pas encore apparue (comme ci-dessus avec jmp fin et fin:), je ne peux pas la calculer. J'aurai aimé pouvoir corriger cela en récupérant en premier lieu les étiquettes puis ensuite la traduction du code et la gestion d'erreur, en testant mon programme sur linux, j'ai découvert de nombreux bugs qui m'ont pris beaucoup de temps, je n'ai pas osé changer mon programme de peur de rajouter d'autres bugs que j'aurai pu ne pas voir ou ne pas avoir le temps de les corriger.

Pour conclure

Les principaux problèmes que j'ai rencontrés ont été comment aborder la récupération des éléments du fichier, la prise en main de "strtok" et le calcul des adresses des étiquettes. Les améliorations possibles de ma partie, seraient une description plus détaillée des erreurs rencontrées ainsi que corriger le problème d'ordre des erreurs avec les calculs d'adresses.

II. Exécution

La partie exécution se divise en quatre fichiers : le fichier « read », le fichier « programme », le fichier « instructions » et le fichier « opération », ainsi que leurs headers respectifs. Dans ces fichiers, l'utilisation de shorts permet de résoudre tous les problèmes d'overflow quant à la taille des entiers.

La partie du main correspondant à l'exécution appelle dans un premier temps les fonctions permettant la lecture du fichier contenant le code machine. Ensuite, tant qu'aucune erreur n'est signalée ou que la fin du programme source n'est pas atteinte, ce dernier est exécuté en parcourant la liste d'instruction.

Les fichiers « programme »

Dans le header de cette partie se trouve la définition de la structure « Programme ». Celle-ci est composée de trois shorts, d'une liste de shorts et de 2 pointeurs sur des shorts :

- La liste de 5000 shorts « mémoire » permet de stocker les différentes valeurs tout au long de l'exécution . Elle contient notamment la pile de la machine.
- Le short « SP » correspond au sommet de la pile.
- Les pointeurs « instructions » et « donnees » font référence aux listes contenant les instructions et les données du programme lu.
- Le short « nb_instructions » correspond aux nombres d'instructions dans le fichier « hexa.txt ». Il permet d'initialiser les listes « instructions » et « donnees » à la bonne taille.
- Le short « PC » fait référence à la prochaine instruction à exécuter.

La fonction « creer_programme » permet d'initialiser correctement la structure programme, et « supprimer_programme » assure la bonne libération de la mémoire à la fin de l'exécution.

La fonction « sélectionner_instruction » appelle l'instruction correspondant à celle à la position PC-1 dans la liste d'instructions.

Le fichier « read »

La fonction « compter_instructions » permet de connaître le nombre d'instructions dans le fichier « hexa.txt » afin d'initialiser les listes de données de d'instructions à la bonne taille.

La fonction « lire_fichier » lit le fichier « hexa.txt » ligne par ligne et met les instructions et données lues dans leurs listes respectives.

Le fichier « instruction »

Dans ce fichier se trouve l'implémentation des différentes instructions du langage machine. Chacune de ces fonctions vérifie que l'accès à la mémoire est correct afin de ne pas créer d'erreur, ainsi qu'il y ait suffisamment d'éléments dans la pile et dans certains cas qu'elle ne soit pas remplie.

La fonction « op » appelle l'opération correspondant à celle de sa donnée.

Le fichier « opération »

Dans ce fichier se trouve l'implémentation des différentes opérations du langage machine. Ces fonctions ne font pas de vérifications de validité de leurs appels, celles-ci ayant été faites au préalable dans la fonction « op ».

La seule faisant une vérification est la fonction « division » qui empêche la division par 0.

Difficultés rencontrées

Afin de produire le meilleur code possible, il était important d'optimiser au mieux notre programme. Une difficulté souvent rencontrée, a été de choisir entre optimiser l'espace mémoire ou optimiser la vitesse d'exécution.

Un exemple notable est celui de la taille des listes contenant les instructions et les données. En effet, celles-ci doivent être initialisées à l'avance, mais quelle taille choisir ?

La première piste explorée était celle d'une structure de liste chaînée, permettant de contourner le problème. Cependant, cette solution rendait le parcours des instructions compliqué pour les sauts d'adresse et ne correspondait pas réellement aux consignes du sujet.

Les autres options étaient les suivantes : connaître le nombre d'instructions pour initialiser des listes d'exactement à la bonne taille, ou initialiser des listes de taille arbitrairement assez grande en faisant la supposition « naturelle » qu'il n'y aura pas plus de 500 instructions. Cette dernière solution à le désavantage d'être particulièrement coûteuse en mémoire, notamment car nous utilisons deux listes différentes, une pour le code des instructions et une pour les données. La première solution, quant à elle, nécessite de lire le fichier contenant le code machine deux fois. En effet, le parcours du fichier pour récupérer les instructions nécessite que les listes soient déjà initialisées. Malgré ce coût en temps de calcul, c'est cette solution qui a été retenue, car le coût en mémoire de la précédente solution semble plus impactant.

Un autre exemple est celui de certaines fonctions dans le fichier opération, comme les fonctions « égal » et « non_egal ». En effet, ces deux fonctions étant assez similaires, la question était de savoir s'il valait mieux éviter la redondance, et donc d'optimiser la mémoire, ou bien écrire deux fonctions différentes afin de faire le moins d'opérations possibles. Dans ce cas, c'est l'optimisation de la vitesse de calcul qui a été faite.