Documentation développeur - Architecture des Ordinateurs

Cours de Emmanuel Lazard

MISEROLLE William, DE PELLEGARS MALHORTIE Nicolas

9 février 2025

Table des matières

1 Introduction **2**

2 Description de notre projet **2**
2.1 Un peu d'histoire . . . 2
2.2 Description résumée du projet . . . 2
2.3 Description précise . . . 2
2.3.1 Un menu interactif et stylisé dans le terminal . . . 2
2.3.2 Opcode . . . 3
2.3.3 Compilation . . . 3
2.3.4 Exécution : . . . 4

3 Explication du code C **5**
3.1 Principe de la pile dans le simulateur . . . 5
3.2 Description des fonctions principales . . . 5
3.3 Complexité des Fonctions . . . 6

4 Bugs, erreurs, solutions, améliorations.... **6**
4.1 Exemples de bugs rencontrés & Nos solutions . . . 6
4.2 Pistes d'améliorations du simulateur . . . 7

5 La construction du projet **8**
5.1 Les outils . . . 8
5.1.1 Les sites et applications : . . . 8
5.1.2 Les livres : . . . 8
5.2 Ordre et répartition des tâches réalisées . . . 8

6 Conclusion **8**

1 Introduction

Ce document présente le projet de traduction d'un programme écrit en assembleur en son équivalent en code machine, puis son exécution. L'objectif est d'implémenter un traducteur capable de convertir chaque instruction assembleur en sa représentation hexadécimale ainsi qu'un exécuteur capable de réaliser ces instructions.

L'objectif est de fournir un compte rendu structuré permettant de comprendre facilement le code et de le faire évoluer pour un potentiel développeur ou même un étudiant aventurier. Nous détaillerons ici les choix techniques, les défis rencontrés, et la manière dont nous avons organisé le projet.

2 Description de notre projet

2.1 Un peu d'histoire

Voyons ensemble rapidement ce qu'est l'hexadécimal pour mieux comprendre notre projet. L'héxadécimal est un système de numération en base 16, utilisant les chiffres de 0 à 9 et les lettres de A à F. Cette norme informatique est utilisée par les processeurs. Chaque chiffre hexadécimal correspond à quatre bits, rendant la conversion entre binaire et hexadécimal plus simple et plus lisible pour les développeurs.

L'assembleur, quant à lui, est un langage de bas niveau directement interprété par le processeur, on dit "bas niveau" car il est proche du code machine. Il utilise des abréviations pour représenter les instructions machine, qui sont ensuite traduites en code hexadécimal.

Dans ce projet, nous avons développé un **compilateur** qui convertit un programme assembleur donné par l'utilisateur en son équivalent hexadécimal, et un **exécuteur** capable d'interpréter ce code hexadécimal pour simuler le comportement du processeur à l'aide d'une pile.

2.2 Description résumée du projet

Le projet consiste en plusieurs fichiers :

— **main.c** : C'est le programme principal : permettant d'interagir avec l'utilisateur.
— **executeur.c** : Contient les fonctions qui permettent d'interpreter le code hexacimal pour l'exécution.
— **assembleur_to_machine.c** : Contient les fonctions de la traduction.
— **assembleur_to_machine.h / executeur.h** : Fichiers headers qui déclarent les structures et fonctions utilisées par le projet.

2.3 Description précise

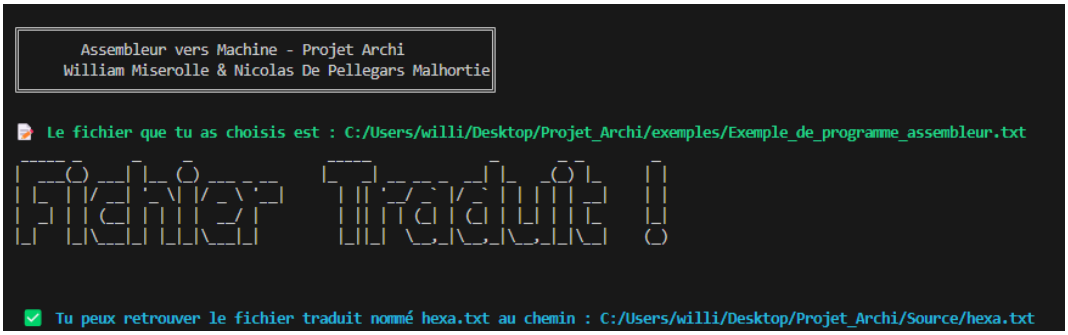
2.3.1 Un menu interactif et stylisé dans le terminal

Pour rendre l'exécution du programme plus agréable, nous avons mis en place comme un menu dans le terminal. Il affiche une boîte encadrée contenant le titre du projet, ainsi que nos noms. Une fois le fichier sélectionné, un message ASCII s'affiche pour signaler la traduction du fichier, suivi d'un message confirmant la création du fichier hexadécimal. C'est au même endroit qu'apparaîtra un message d'erreur s'il existe.

Effets visuels et feedback utilisateur Nous avons intégré des couleurs et des animations ASCII pour dynamiser l'affichage :

— Une mise en couleur des messages d'erreur et de validation grâce aux séquences ANSI.
— Un affichage ASCII du message "Fichier Traduit !" pour montrer clairement la réussite de la traduction.
— Une confirmation finale indiquant le chemin où se trouve le fichier hexadécimal généré.

Capture d'écran du menu Ci-dessous, une capture d'écran illustrant notre menu en action :



2.3.2 Opcode

Un **opcode**, abréviation de *code d'opération*, est un nombre binaire ou hexadécimal qui représente une instruction spécifique qu'un processeur doit exécuter. Par exemple, un opcode de `0001` peut signifier une addition entre deux nombres, tandis qu'un opcode de `0010` peut correspondre à une soustraction. Les opcodes de ce projet sont définis dans l'énoncé par exemple `(2)` correspond à : *push x*. Nous avons fait le choix de ne pas les laisser tels des nombres entiers mais de les écrire sous forme hexadécimale pour mieux correspondre aux attentes de traduction.
Par exemple `(2)` devient `(0X02)`. Idem pour des valeurs supérieures à 10 comme *op i* qui passe de `(11)` à `(0X0B)`.

2.3.3 Compilation

La première partie est la compilation du fichier. Le gros du travail consiste à détecter si le fichier lu est conforme à la syntaxe assembleur. On peut ensuite procéder à l'édition du fichier hexadécimal.

Lecture et analyse du code assembleur Le programme commence par ouvrir en mode "lecture" le fichier assembleur, c'est-à-dire qu'il ne peut pas modifier le texte mais seulement le lire, contenant le code. Pour chaque ligne, on vérifie si elle est correcte avant d'en extraire l'instruction ainsi que, si elle existe, une étiquette associée.

- Si une étiquette est présente, elle est vérifiée selon les règles suivantes :
  - Elle ne doit pas dépasser 30 caractères. Si c'est le cas, un message d'erreur est affiché : `Erreur dans l'étiquette : elle dépasse 30 caractères`.
  - Elle doit commencer par une lettre. Sinon, le programme affiche : `Erreur dans l'étiquette : le premier caractère de l'étiquette n'est pas une lettre`.
  - Elle ne peut contenir que des caractères alphanumériques et le tiret du 8 `_`. Si une erreur est détectée, un message est affiché : `ERREUR a cause de X invalide à la pos Y`.
- Si une instruction inconnue est rencontrée, nous avons le message d'erreur `ERREUR !! : l'instruction X n'est pas connue`.
- Si un paramètre existe, il est vérifié :
  - Si ce paramètre est censé être un entier mais contient un caractère invalide, un message s'affiche : `Le paramètre n'est pas un entier, X à la pos Y n'est pas un chiffre`.
  - Si le paramètre est une étiquette, elle est vérifiée. Si elle n'a pas été déclarée auparavant, l'erreur suivante est générée : `Le paramètre X est invalide`.
- Si une ligne contient une erreur de syntaxe globale (ex. une mauvaise structure d'instruction ou d'étiquette), l'erreur suivante est retournée : `ATTENTION !!! erreur de syntaxe à la ligne : X`.
- Si une étiquette est déclarée plusieurs fois dans le programme on a le message : `ATTENTION l'étiquette X existe déjà !`.

Pour verifier les types des caractères on utilise la bibliothèque ctype. Dans la fonction valider_etiq par exemple.

Extraction des étiquettes Un premier passage est effectué pour collecter toutes les étiquettes présentes dans le programme et les stocker dans un tableau dédié. Ces étiquettes sont enregistrées avec leur adresse mémoire afin de permettre le bon fonctionnement des instructions de saut comme `jmp`, `jnz`, `call`.

Traduction en hexadécimal Nous faisons un deuxième passage de lecture. Chaque instruction assembleur est convertie en son équivalent hexadécimal en utilisant une table regroupant leurs "op codes". Les instructions nécessitant un paramètre (valeur entières par exemple ou adresse d'étiquette) sont calculées grâce à une adresse courante incrémentée au fur et à mesure des lignes, pour assurer une conversion correcte.

Génération du fichier hexadécimal Les instructions traduites sont écrites dans un fichier de sortie `hexa.txt`. Ce fichier contient l'équivalent hexadécimale du programme. Il servira de fichier d'entrée pour l'exécuteur.

2.3.4 Exécution :

La seconde partie de ce projet est l'exécution du programme. Pour cela, on recueille tout d'abord les instructions du fichier hexadécimal que l'on sauvegarde dans un tableau d'instructions avant de parcourir ce tableau selon la valeur du registres PC. L'executeur est construit de manière à être capable de reperer certaines erreurs de programmation commises par l'utilisateur.

Lecture du fichier hexadecimal : L'exécuteur lit le fichier grâce aux fonctions `savecode()` et `nombre de ligne()` et enrengistre les informations de chaque lignes dans un tableau d'instruction. L'instruction à l'élément i du tableau désigne l'instruction à la ligne i+1 (car le tableau commence à 0, et cela n'a pas de sens de parler de ligne 0 d'un fichier.txt).

Execution : L'execution du programme est faite suivant la boucle infinie while(1) présente dans le main. Cette boucle tourne jusqu'a ce qu'on arrive à un appel à halt. A chaque itération de la boucle, PC est incrémenté, car PC représente l'instruction suivante. C'est pourquoi la fonction executeligne accède à l'instruction à l'indice PC-1. La fonction executeligne() permet d'accéder à la fonction assembleur que l'on souhaite exécuter en fonction de la valeur du code opération de l'instruction.

Verification d'erreurs : La difficulté de la conception de l'exécuteur est de configurer la détection d'erreurs. Notre programme repère plusieurs types d'erreurs différentes :

- Débordement négatif de la pile. Cela arrive quand on veut dépiler un élement, mais que la pile est vide.
- Débordement positif de la pile. Cette erreur a priori plus rare peut arriver si l'utilisateur saisie un programme exrêmement lourd, que notre mémoire ne peut supporter.
- Tentative d'acceder à une adresse inexistante, par exemple write 5001.
- Tentative d'accéder à une instruction inexistante. Cela arrive si PC<0, ou si PC >= nombre de ligne. Ces détections d'erreurs sont placées dans la boucle while, afin de vérifier en permanence si le programme peut continuer normalement, mais également dans les fonctions JMP, JNZ et CALL afin de reperer où se trouve l'erreur pour faciliter le débuggage.
- Si le résultat d'une opération n'est pas représentable sur 16 bits.

Nous sommes capables d'identifier la ligne de l'erreur à l'aide de la valeur de PC, ce qui facilite le débuggage pour l'utilisateur. Ces erreurs arrêtent immédiatement le programme à l'aide de exit(EXIT_FAILURE), et sont notifié en gras et rouge à l'utilisateur, selon la synthaxe suivante : on écrit "\033[1 ;31m" avant le texte et "\033[0m" après. \033[est l'échappement ANSI pour changer la couleur, 1 ;31m permet de mettre le texte en gras (1) et en rouge (31). \033[permet de reinitaliser la couleur normale du texte. Nous avons également intégré quelques warning dans le programme pour détecter ce qui pourrais être des erreurs dans certains cas. Les warnings sont :

- Si la pile peut écraser des données : Si l'utilisateur sauvegarde une donnée à une adresse trop proche de 0, il est possible qu'on empile une valeur à cette même adresse. Ainsi, un warning est généré si, lors d'un appel à une des fonction push, la valeur à l'adresse SP est non nulle.
- S'il y a un nombre différents de call et de ret

Ces warning sont écrits en orange de la même manière que les erreurs sont écrites en rouge.

3 Explication du code C

3.1 Principe de la pile dans le simulateur

Une pile est une structure qui suit le principe du LIFO : Last In, First Out, ce qui veut dire que le dernier élément ajouté est le premier à être retiré. Dans le cadre de notre projet de machine à pile, cette structure est utilisée pour gérer efficacement les opérations arithmétiques, les sauts conditionnels et les appels de procédures.

Fonctionnement de la pile La pile est manipulée à l'aide d'un registre SP (Stack Pointer), qui indique la première case mémoire libre de la pile. Elle fonctionne de la manière suivante :

- **push** Ajoute un élément en haut de la pile et incrémente SP
- **pop** : Retire l'élément du sommet de la pile et décrement SP

Avantage de l'utilisation d'une pile

- **Simple à manipuler** : Pas besoin d'adresser explicitement des variables en mémoire.
- **Facilite l'écriture du code :** Les fonctions et les boucles (JNZ, CALL) utilisent directement les valeurs du haut de la pile, et donc pas besoin de leur adresser explicitement les adresses des données à utiliser.

3.2 Description des fonctions principales

Le fichier `assembleur_to_machine.c` contient les fonctions de la traduction. Voici une description des fonctions utilisées :

obtenir_opcode(const char* instruction) : Cette fonction recherche dans la table des opcodes, défini dans le fichier, l'instruction assembleur mise en paramètre et retourne son équivalent en hexadécimal. Si on ne trouve pas l'instruction demandée, la fonction renvoie `-1` *(PS : l'utilisation de const précise que la variable ne vas pas changer).*

valider_etiq(const char* etiq) : Cette fonction vérifie si une étiquette est conforme aux règles données dans l'énoncé (et suppositions naturelles) - en renvoyant 1 si oui et 0 sinon.

est_un_entier(const char* str) : Cette fonction détermine si la chaîne de caractères fournie représente un nombre entier valide. Elle prend en compte les signes + et - en début de chaîne. Elle parcourt toute la chaine et vérifie si chaque caractère est un chiffre. En renvoyant 1 si oui et 0 sinon.

collecter_etiq(const char* fichier_assembleur, Tableau_detiqs* tbl_etq) : Cette fonction effectue un premier passage sur le fichier assembleur en lecture pour sauvegarder les étiquettes et leurs adresses. Elle stocke ces informations dans un tableau d'étiquettes afin de permettre la bonne gestion des sauts.

traduire_instruction(const char* ligne, char* code_hexa, int adresse_courante, Tableau_detiqs* tbl_etq) : La fonction prend en entrée une ligne assembleur, reconnait l'instruction et son paramètre s'il y en a un. Ensuite elle génère la représentation hexadécimale correspondante en mettant bien en forme selon la norme héxadécimal : 2 caractères, espace, 4 caractères. Elle gère aussi les références aux étiquettes en les remplaçant par leur adresse mémoire. Elle renvoie 0 si tout c'est bien passé.

traducteur(const char* fichier_assembleur, const char* fichier_hexa) : Cette fonction réalise grâce aux autres fonctions la traduction de tout un programme assembleur. Elle :

- Lit le fichier assembleur ligne par ligne.
- Vérifie et collecte les étiquettes.
- Convertit chaque instruction en hexadécimal.
- Écrit le résultat dans le fichier `hexa.txt`.

Grâce à ces fonctions, le traducteur est capable de convertir efficacement un code assembleur en instructions machines utilisables par un simulateur.

int nombreDeLigne(const char* nomfichier) : Calculer le nombre de ligne dans le fichier est utile pour plusieurs raisons. Cela permet de créer le tableau d'insruction avec malloc ainsi que de verifier si le registre PC est toujours dans des valeurs correctes, pour détecter des erreurs. Pour compter le nombre de ligne, on compte le nombre de caractère "\n" en ajoutant 1 à la fin car notre fichier hexadecimal ne contient pas ce caractère à la dernière ligne.

void savecode(instruction* tab[], const char*nomfichier) : La rôle de cette fonction est de remplir le tableau d'instructions, à la base vide, avec les instructions contenue dans le fichier. On lit pour cela chaque ligne du fichier dans un chaine de caractères, et que l'on recopie en 2 chaines : une contient le code opération et l'autre la donnée. On peut ainsi, avec la fonction strtol de la bibliothèque stdlib, obtenir la valeur en base 10 de la valeur hexadécimale contenue dans une chaine de caractère. On effectue après, le calcul permettant d'obtenir un nombre signé pour la donnée.

Executeligne : On execute le programme ligne par ligne. Tab_ins[*pPC-1] contient le code opération de la ligne à executer. On sélectionne ainsi l'instruction assembleur à executer.

3.3 Complexité des Fonctions

Fonction	Fichier	Complexité
charger_fichier()	assembleur_to_machine.c	$O(L)$
verifier_syntaxe()	assembleur_to_machine.c	$O(L \times I)$
traduire_instruction()	assembleur_to_machine.c	$O(I)$
ecrire_fichier_machine()	assembleur_to_machine.c	$O(L)$
analyser_operande()	assembleur_to_machine.c	$O(1)$
obtenir_opcode()	assembleur_to_machine.c	$O(1)$
generer_code_machine()	assembleur_to_machine.c	$O(L \times R)$
nombreDeLigne()	executeur.c	$O(L)$
savecode()	executeur.c	$O(L)$
main()	assembleur_to_machine.c	$O(L \times I + L \times R)$

TABLE 1 – *L : nb. de lignes | I : nb. d'instructions | R : nb. d'opérations*

L'ensemble des autres fonctions du fichier executeur.c présentent une complexité O(1)

4 Bugs, erreurs, solutions, améliorations....

4.1 Exemples de bugs rencontrés & Nos solutions

Ligne vide à la fin du fichier

Problème rencontré :

Le code de toute la partie exécution par du principe que le fichier héxadécimal comporte à chaque ligne 2 caractères, un espace, et 4 autres caractères sans vérification d'erreurs. Or, lors du premier test du programme, notre fichier hexadécimal généré comportait une ligne en trop. Ce qui a posé problème était qu'aucun message d'erreur n'était généré, le programme se contentant de s'arrêter de suite rendant le problème difficile à identifier

Solution apportée :

Pour corriger ce problème, nous avons ajouté une variable `nb_de_ligne_fichier_assembleur` qui comptabilise le nombre total de lignes dans le fichier assembleur d'entrée. Lors de l'écriture du fichier hexadécimal, nous avons modifié la gestion des sauts de ligne en appliquant la condition suivante :

```
adresse_courante++;

// Empêcher l'écriture d'un saut de ligne supplémentaire à la fin
if (adresse_courante < nb_de_ligne_fichier_assembleur) {
```

```
        fprintf(sortie, "%s\n", code_hexa);
    } else {
        fprintf(sortie, "%s", code_hexa);
    }
```

Ainsi, si l'adresse courante est inférieure au nombre total de lignes du fichier assembleur, un saut de ligne est ajouté. En revanche, pour la dernière ligne, l'instruction est écrite sans , évitant ainsi la ligne vide en fin de fichier.

Difficulté pour tester le programmme Afin de tester le simulateur sur d'autres programmes que celui donné dans le sujet, nous avons demandé à ChatGPT de générer des programmes en assembleur. Cela a permis de corriger quelques erreurs qu'il y avait dans notre programme, comme par exemple de ne pas l'interrompre lorsque le résultat d'une opération n'est pas représentable sur 16 bits. Ce qui était difficile était, en cas d'erreur, de comprendre si le problème venait de notre simulateur ou du programme généré par l'ia, qui n'est pas encore infaillible.

4.2 Pistes d'améliorations du simulateur

Compteur dans les boucles : Nous avions par ailleurs pour objectif de préciser, dans le cas ou le programme contient des boucles, de préciser à quelle itération de la boucle est intervenue l'erreur afin de faciliter le débuggage par l'utilisateur, en créeant des entiers CPT_JMP, CPT_JNZ, et CPT_CALL, et d'incrémenter leur valeur dans les fonctions JMP, JNZ et CALL par des pointeurs. Cet ajout est très simple à incoporer dans des programmes qui utilisent des boucles simple, mais dès qu'il y a des boucles imbriquées, il nous est apparu impossible de rendre cette fonctionalité disponible. Nous avons donc choisit de ne pas incorporer ces compteurs.

Correction automatique des étiquettes : Une autre idée que nous avons eue pour améliorer le programme est la correction automatique des fautes d'orthographe dans les étiquettes. L'idée serait de détecter les erreurs d'écriture dans les noms d'étiquettes, de les signaler à l'utilisateur, puis de les rectifier automatiquement pour éviter toute erreur d'exécution.

Nous pensions à par exemple la méthode de la distance de Hamming*(cf. références)*, pour comparer les chaines de caractères. Une fois l'erreur détectée, le programme proposerait une correction et remplacerait l'étiquette incorrecte par sa version correct.

Cependant :

— La distance de Hamming n'est pas idéale car elle se contente de compter le nombre de caractères différents entre deux chaines *de même longueur*. Pour que la correction soit réellement efficace, il aurait fallut utiliser un algorithme de calcul de distance plus perfectionné, comme celui de Levensthein, ce qui aurait complexifié le code.

— Aussi une correction automatique pourrait poser problème si plusieurs étiquettes avec des noms presque identiques existent, entraînant des modifications indésirées.

Autoriser des commentaires : Nous voulions inclure la possibilité d'écrire des commentaires dans le programme après le caractère ';'. On aurait pu pour cela lire la ligne en la pourcourant caractère par caractère dans la fonction traduire_instruction(), et les detecter à l'aide d'un if. Dans notre projet, nous avons supposé qu'une ligne d'assembleur ne dépasserait pas 100 caractères. Cependant, si un programmeur bavard ajoute un commentaire après un point-virgule ; et que ce commentaire est très long, la ligne risque de dépasser cette limite, ce qui pourrait provoquer des erreurs de lecture.

Pour résoudre ce problème, une solution aurait été d'utiliser une allocation dynamique pour gérer la longueur des lignes. Cependant, cette approche aurait considérablement complexifié le code, ce qui a motivé notre choix de ne pas l'implémenter dans cette version du projet. Nous avions aussi pensée à une alternative qui serait d'imposer explicitement une longueur des commentaires et d'afficher un avertissement si un commentaire dépasse une certaine taille. Cependant par manque de temps, nous n'avons pas pu implémenter cette fonctionnalitée.

5 La construction du projet

5.1 Les outils

5.1.1 Les sites et applications :

— Google docs pour rédiger et noter nos idées
— Discord pour pouvoir discuter et s'appeler afin de transmettre les idées.
— Visual Studio Code le logiciel de code nécessaire pour développer le projet.
— Overleaf pour réaliser le compte rendu du projet
— Couleurs Ascii Art pour trouver des couleurs et comprendre comment les utiliser dans le terminal.
— Wikipedia.org pour trouver des informations sur les piles.
— Github pour se renseigner sur du code informatique et sauvegarder notre code sur un repository.

5.1.2 Les livres :

— Initiation à l'algorithmique et à la programmation en C - Malgouyres, Rémy (Disponible à la BU. Editions Dunod, 2015)

— Ebook Learn Latex - Jérémy Just (learnlatex.org)

5.2 Ordre et répartition des tâches réalisées

Tâches	Nom(s) du/ des réalisateurs	Statut
- Définition des objectifs du projet (compilateur et exécuteur)	Nicolas et William	Terminée
- Structuration des fichiers et organisation du projet	Nicolas et William	Terminée
- Étude des instructions assembleur de l'énoncé	Nicolas et William	Terminée
- Création de la table des opcodes	William	Terminée
- Développement du compilateur assembleur vers hexadécimal	William	Terminée
- Gestion des étiquettes et calcul des sauts	Nicolas et William	Terminée
- Vérification et validation des instructions en entrée	William	Terminée
- Génération et écriture du fichier hexadécimal	William	Terminée
- Développement de l'exécuteur pour interpréter le fichier hexadécimal	Nicolas	Terminée
- Développement du menu interactif et gestion des entrées utilisateur	Nicolas et William	Terminée
- Tests unitaires sur la traduction et l'exécution	Nicolas	Terminée
- Correction des bugs et amélioration des performances	Nicolas	Terminée
- Documentation et commentaires du code	Nicolas et William	Terminée
- Rédaction du compte rendu en LaTeX	Nicolas et William	Terminée

Le tableau de répartition

6 Conclusion

En conclusion ce projet nous a permis de mettre en pratique les connaissances en programmation C et d'appronfondir notre connaissance de l'assembleur.

Au fil de ce projet, nous avons pris conscience de l'importance de l'organisation, en particulier sur une durée de plusieurs semaines. L'utilisation d'outils de versioning tels que GitHub nous a grandement facilité la collaboration et la gestion du code. La rédaction de ce compte rendu en LaTeX nous a aussi aidés à structurer nos idées de manière claire et à documenter notre démarche nous permettant de prendre du recul et de mieux comprendre ce que nous avons faits.

Ce projet de traduction et d'execution nous a permis de perfectionner nos connaissance sur le système de pile et la manière dont un programme est executée à l'interieur de l'ordinateur.

Au-delà de l'aspect informatique, ce projet nous a appris à travailler efficacement en binôme : un exercice peu fréquent à l'université, où les examens sont individuelles. Nous espérons que ce projet vous

a intéressé et qu'il pourra servir de référence aux étudiants souhaitant s'initier à la programmation d'un traducteur et d'un executeur.

On pensait que notre pire cauchemar était l'algèbre... puis on a découvert les segmentations faults

Références

[1] Algorithme de hamming. https://irem.univ-lille.fr/~site/IMG/pdf/142_distance_entre_les_mots.pdf.

[2] C tutoriel. https://www.w3schools.com/c/index.php.

[3] Ctype library. https://koor.fr/C/cctype/Index.wp.

[4] Latex FAQ. Comment afficher toutes les entrées d'un fichier .bib ? https://www.latex-fr.net/3_composition/annexes/bibliographie/afficher_toutes_les_entrees_d_un_fichier_bib#:~:text=si%20%5Cnocite%7B*%7D%20est%20plac%C3%A9e,entr%C3%A9es%20cit%C3%A9es%20dans%20le%20document.

[5] Latex FAQ. Comment citer une url avec bibtex ? https://www.latex-fr.net/3_composition/annexes/bibliographie/citer_une_url.

[6] Msys2. How to install msys2 on windows, Janvier 2021. https://www.msys2.org/wiki/MSYS2-installation/.

[7] Overleaf. Bibliography management with bibtex. https://fr.overleaf.com/learn/latex/Bibliography_management_with_bibtex.

[8] Overleaf. Documentation. https://fr.overleaf.com/learn.