|  |  |  |
| --- | --- | --- |
|  | | |
| **Compte-rendu de projet : développement de l’ISS du MiniMIPS** | | |
|  | ENSTA Bretagne  2 rue François Verny  29806 Brest Cedex 9, France | LANDAIS Erwann,  [erwann.landais@ensta-bretagne.org](mailto:erwann.landais@ensta-bretagne.org)  FOURNIOL Nathan,  [nathan.fourniol@ensta-bretagne.org](mailto:nathan.fourniol@ensta-bretagne.org) |

Table des matières

[Table des matières 2](#_Toc2539906)

[I) Généralités sur le langage assembleur choisi 2](#_Toc2539907)

[II) Description de l’assembleur et du codeur de données 3](#_Toc2539908)

[1) Description générale de l’assembleur 3](#_Toc2539909)

[2) Fonctions principales de l’assembleur 3](#_Toc2539910)

[3) Description et particularités du codeur de données 3](#_Toc2539911)

[III) Description de l’ISS 4](#_Toc2539912)

[IV) Évaluation de l’ISS 4](#_Toc2539913)

[V) Interface Graphique 5](#_Toc2539914)

[Conclusion 6](#_Toc2539915)

[Annexe 7](#_Toc2539916)

[Annexe 1 : Multiplication de matrices de toute taille 7](#_Toc2539917)

Un simulateur de jeu d'instructions (ou ISS en anglais : instruction set simulator) permet de simuler des programmes écrits en code « binaire », pour un processeur donné. Ce code binaire est issu d’un ensemble d’instructions écrites selon une syntaxe particulière, appelée code assembleur. Afin de se familiariser avec ces notions, le but de ce projet était de créer un simulateur de jeu d’instructions et d’évaluer son efficacité. Pour cela, nous avons commencé par choisir la syntaxe de notre code assembleur. Par la suite, nous avons créé un programme assembleur, capable de passer des instructions d’un format assembleur en un format hexadécimal ; nous avons également créé un programme de structure similaire capable de passer des instructions d’un format décimal à un format hexadécimal, afin de coder nos données. Ensuite, nous avons créé un simulateur de jeu d’instructions en python. Nous avons enfin pu évaluer l’efficacité de celui-ci.

# I) Généralités sur le langage assembleur choisi

Globalement, la syntaxe choisie est identique à celle utilisée en cours :

\*les registres sont écrits ainsi : r + numéro du registre

\*les valeurs entières sont écrites sans caractères spéciaux devant

\*la présence d’un ‘#’ désigne le début d’un commentaire, sur une ligne.

\*les boucles sont écrites ainsi : L\_ + nom de la boucle. Il n’est pas obligatoire de faire une tabulation pour les instructions présentes dans cette boucle, mais cela est recommandé pour des questions de lisibilité.

Un exemple est disponible en annexe, concernant la multiplication de matrices de toute taille.

# II) Description de l’assembleur et du codeur de données

## 1) Description générale de l’assembleur

L’assembleur utilisé est appelé asmtohex.py. Écrit en langage python, il permet de prendre les instructions d’un fichier assembleur et de les écrire selon un format hexadécimal, codé sur 32 bits.

Ces deux derniers fichiers sont des .txt. A l’origine, le fichier assembleur est appelé ‘asmInstructions.txt’, tandis que le fichier hexadécimal est appelé 'hexInstructions.txt' . Ces noms peuvent être modifiés au sein du programme.

Il faut également noter que les commentaires sont possibles au sein du fichier ‘asminstructions.txt’ ; ceux-ci doivent simplement commencer par un ‘#’.

## 2) Fonctions principales de l’assembleur

Ce programme correspond à l’enchaînement suivant de fonctions :

**-**l**oad\_ASM ,** permettant de charger un fichier .txt et de récupérer ses données sous forme de liste. Elle prend en entrée le nom du fichier à modifier, et renvoie une liste contenant les lignes du fichier non vides, sans commentaires. Ces commentaires sont repérés par la présence d’un ‘#’.

**-analyze\_instructions :** cette fonction transforme une liste d'instructions en code assembleur donnée en entrée en liste de nombres. La taille et le contenu de cette liste varient selon l’instruction sélectionnée, et correspondent aux consignes du projet.

Pour les instructions ‘branz’, ‘braz’ et ‘jmp’, l’assembleur supporte le fait que les adresses de saut associées ne soient pas écrites explicitement (par des valeurs décimales). Le code assembleur peut contenir des noms associés à chacune de ces adresses ; ces noms doivent cependant commencer par un L (L\_loop, L\_k, …). La fonction analyze\_instructions associe à chacun de ces noms une adresse décimale, qu’elle renvoie ensuite dans la liste de nombres de sortie.

**-compute\_hex\_instructions :** cette fonction transforme la liste de nombres données en entrée en liste d’instructions hexadécimales, renvoyées en sortie. Là encore, le codage de ces instructions diffère selon le type d'opération. Ces instructions hexadécimales sont codées sur 32 bits.

**-output\_hex\_instructions :** cette fonction crée le fichier hexadécimal (dont le nom est donné en entrée) à partir des données hexadécimales prises en entrée, sous forme de liste. Il associe également à chacune de ces données une adresse, qui est rajoutée dans le fichier hexadécimal : ainsi, les adresses de ces données se suivent.

## 3) Description et particularités du codeur de données

Afin de faciliter la manipulation des données utilisées au sein des programmes, il nous a semblé nécessaire de créer un programme capable de transformer les données décimales en données hexadécimales. Appelé datawriter.py, il possède une structure similaire au programme asmtohex.py : une fonction pour charger un fichier.txt et récupérer ses données (**load\_dec**) ; une fonction pour renvoyer les données du fichier sous forme de liste (**analyse\_data**) ; et une fonction pour renvoyer les données ainsi que leur adresse sous forme hexadécimale (**output\_hex\_data**).

A noter que l’adresse de la première donnée peut être modifiée selon les besoins. Par défaut, cette adresse est 1.

# III) Description de l’ISS

L’ISS est codée sous Python dans le fichier MIPS-X.py. Il se présente sous la forme d’une classe appelée **VM**. Au sein du constructeur, il est possible de définir le nombre de registres via la valeur n\_reg (initialement à 32) et la taille de la mémoire n\_mem (initialement à 1024). Le constructeur récupère aussi l’ensemble des instructions et des données hexadécimales, et les stocke dans des listes distinctes.

Le programme possède un mode continu et un mode pas à pas.

L’ISS s’exécute via la fonction **run**, qui ne s’arrête que lorsqu’elle rencontre l’instruction « stop ». Une itération de cette fonction se présente sous la forme de l’enchaînement suivant de sous-fonctions :

-affichage du contenu des registres (via **showregs** )

-sélection d’une instruction dans la liste correspondante (via **fetch** )

-décodage de l’instruction, attribution des valeurs aux variables correspondants à cette instruction (via **decode** ).

-choix de l’instruction et exécution de celle-ci (via **evalu** ).

Une fois l’instruction « stop » atteinte, le programme renvoie le temps d’exécution du programme.

L’ISS compte également le nombre de cycles ayant été effectué pendant toutes les instructions. Il possède également un cache, influençant le nombre de cycles effectué par le programme. On considère alors qu’un accès à une donnée dans le cache consomme un cycle, tandis qu’un accès à une donnée dans la mémoire consomme 10 cycles.

L’ISS possède enfin une interface graphique. Pour l’afficher, il est nécessaire d’utiliser le fichier InterfaceforMIPSX\_cache.py. Celui-ci utilise alors le fichier MIPS\_X\_Cache\_interface.py.

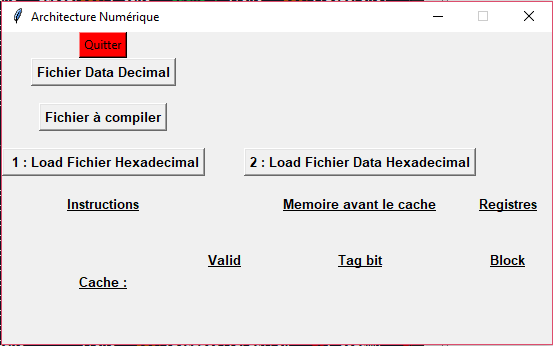
# IV) Évaluation de l’ISS

Afin d’évaluer les performances maximales de notre ISS, nous avons créé un programme appelé MIPS-X-max-perf. Il est similaire au programme MIPS-X.py, mais fonctionne uniquement en mode continu et n’affiche aucune information sur le déroulé du programme, afin d’évaluer les performances de notre ISS le plus exactement possible.

Nous avons pu évaluer notre ISS sur une série de tests (10000). Sans cache, en moyenne, l’ISS effectue 702593 cycles par seconde ; sa fréquence est donc de 0,7 Mhz, ce qui correspond à un cycle tous les 1,42 µs .

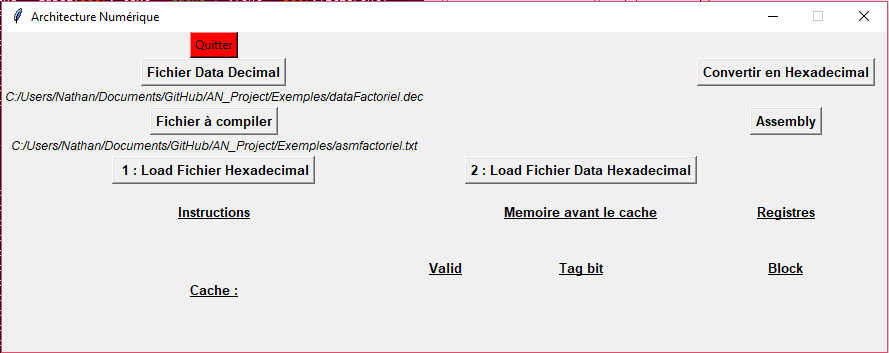
Via le fichier MIPS-X-max-perf-Cache, nous avons aussi pu évaluer la performance de notre ISS avec un cache implémenté. Celui-ci effectue alors 818685 cycles par seconde ; sa fréquence est de 0,8MHz, ce qui correspond à un cycle tous les 1,22 µs .

# V) Interface Graphique



Lors du lancement du fichier d’interface avec Python3, on génère la fenêtre ci-dessus codée avec la bibliothèque Tkinter.

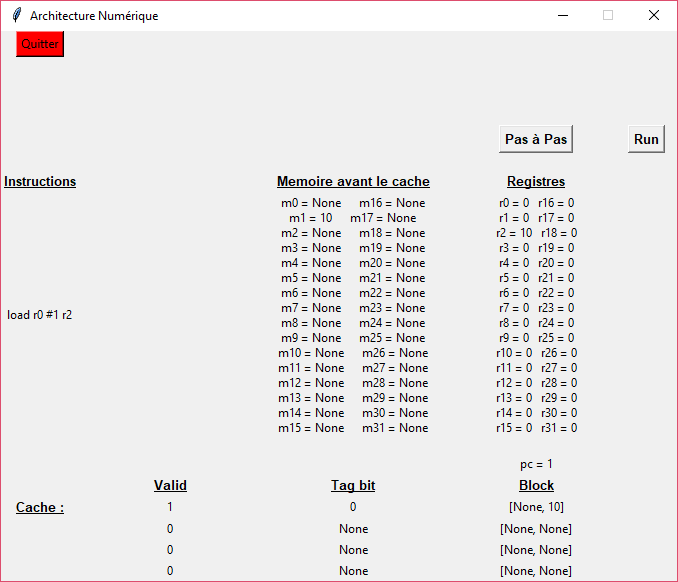
A partir de cette fenêtre, on peut charger un fichier de donnée en cliquant sur **Fichier Data Decimal** sous forme décimale, que l’on veut convertir en hexadécimal. De même, on peut charger un fichier écrit en assembleur grâce au bouton **Fichier à compiler**. L’interface va se modifier afin de laisser apparaitre les boutons permettant la conversion de ces deux fichiers. Il suffit de cliquer dessus pour que les fichiers soient convertis et automatiquement chargé pour l’exécution. Si on dispose des fichiers en hexadécimal on peut directement utiliser les boutons **Load Fichier Hexadécimal** pour le programme **et Load Fichier data Hexadecimal** pour le fichier de données.



Ensuite, une fois les fichiers chargées, les boutons **Pas a Pas** et **Run** s’affiche afin d’exécuter le programme. Les différentes variables s’affichent en dessous de leur identifiant.

Une fois l’exécution finis, un bouton **Réinitialiser** permet la réinitialisation de tout le programme afin de pouvoir exécuter un nouveau code.

Fenêtre en mode Run ou Pas à pas



# Conclusion

Même si cette performance est bien en dessous des performances des processeurs classiques (ce qui s’explique en partie par le langage de programmation choisi), ce projet nous aura permis de manipuler les notions d’assembleur et d’ISS. Il faudrait désormais comparer les performances de cet ISS avec celui d’un ISS codé en C, afin d’évaluer le gain de performances entre ces deux langages.

# Annexe

## Annexe 1 : Multiplication de matrices de toute taille

load r0,1,r1  
 load r0,3,r8  
 mult r8,r1,r8  
 add r8,3,r8  
L\_i: seq r1,0,r4  
 branz r4,L\_end  
 sub r1,1,r1  
 load r0,2,r2  
L\_j: seq r2,0,r4  
 branz r4,L\_i  
 add r0,0,r5  
 load r0,3,r3  
 load r0,2,r4  
 mult r4,r3,r4  
 add r4,r8,r4  
 load r0,1,r9  
 mult r9,r1,r9  
 add r9,r2,r9  
 add r9,r4,r9  
 sub r3,1,r3  
L\_k: slt r3,0,r4  
 branz r4,L\_k\_1  
 load r0,3,r4  
 mult r4,r1,r4  
 add r4,r3,r4  
 add r4,4,r4  
 load r4,0,r6  
 load r0,3,r4  
 mult r4,r3,r4  
 add r4,r2,r4  
 add r4,r8,r4  
 load r4,0,r7  
 mult r6,r7,r6  
 add r5,r6,r5  
 sub r3,1,r3  
 jmp L\_k,r0  
L\_end: stop  
L\_k\_1: store r9,0,r5  
 sub r2,1,r2  
 jmp L\_j,r0