lab 1
Un peu d'assembleur

M. Briday

1 Objectif

L'objectif du TP est de se familiariser avec l'assembleur, et en particulier celui du RISC-V. Cette première étape est indispensable pour bien comprendre l'architecture interne du µC.

2 Installation

Le TP utilise l'éditeur VSCode [1], avec l'extension : `RISC-V Venus Simulator`. C'est un simulateur de jeu d'instruction RISC-V, aucun cross-compilateur n'est nécessaire pour ce premier TP.

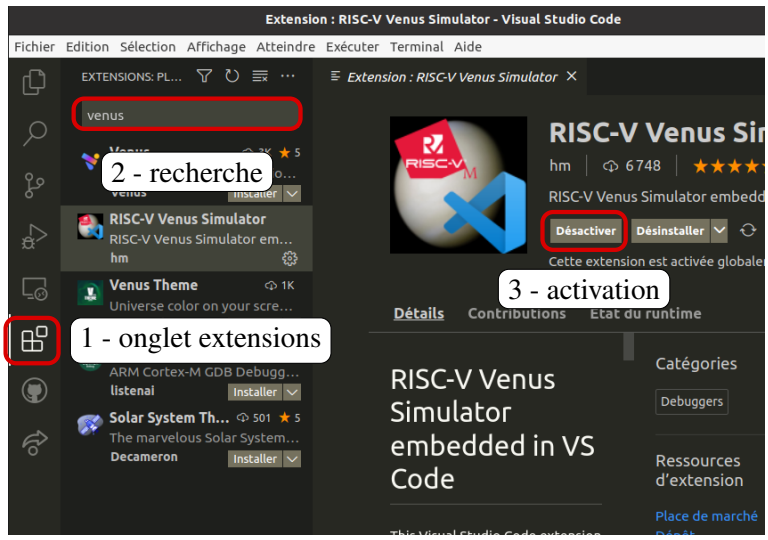

FIGURE 1 – Installation de l'extension RISC-V Venus Simulator.

Pour installer/activer l'extension, voir figure 1, il suffit de :

1. `https://code.visualstudio.com/`

— cliquer sur l'onglet de choix des extensions
— taper le nom de l'extension dans la barre de recherche (il faut du réseau...)
— activer l'extension

Pour démarrer, il faut créer un fichier texte avec l'extension `.s` et de l'ouvrir avec VSCode (figure 2)

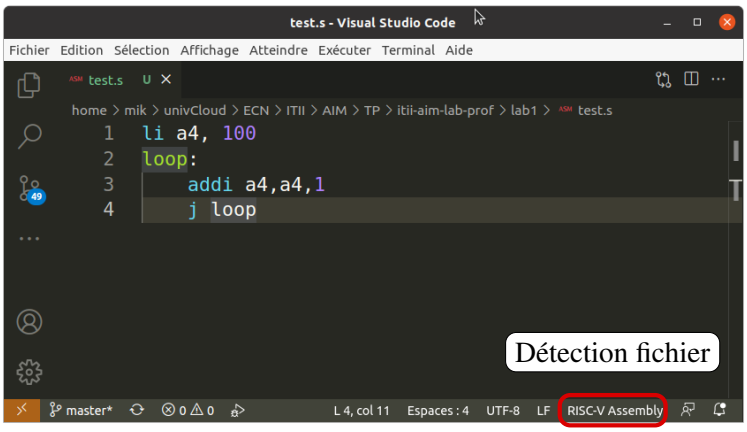

FIGURE 2 – Détection du format de fichier.

On peut ensuite démarrer la simulation en appuyant sur `F5` (figure 3). Il peut être nécesssaire de cliquer sur l'onglet `debug` à gauche pour afficher l'état des registres. On peut à partir de là simuler une application.

Note : Il y a une autre extension pour avoir la coloration syntaxique de l'assembleur. Il suffit de chercher `risc-v` dans les extensions : l'extension s'appelle `RISC-V Support`.

3 Application

3.1 Fonctions

▷ implémenter une fonction : `void toUpperCase(char *str);` qui transforme une chaine de caractère ASCII en majuscule. [2]

La fonction doit respecter l'ABI (passage des paramètres, sauvegarde des registres si besoin (*callee-saved registers*)).

Pour définir une zone de données en assembleur, on utilise la directive `.data` :

```
# pour récupérer l'adresse de 'chaine':
la a0, chaine
```

2. On pourra remarquer que les codes ascii d'une majuscule et de la minuscule associé diffèrent d'une valeur constante (32).

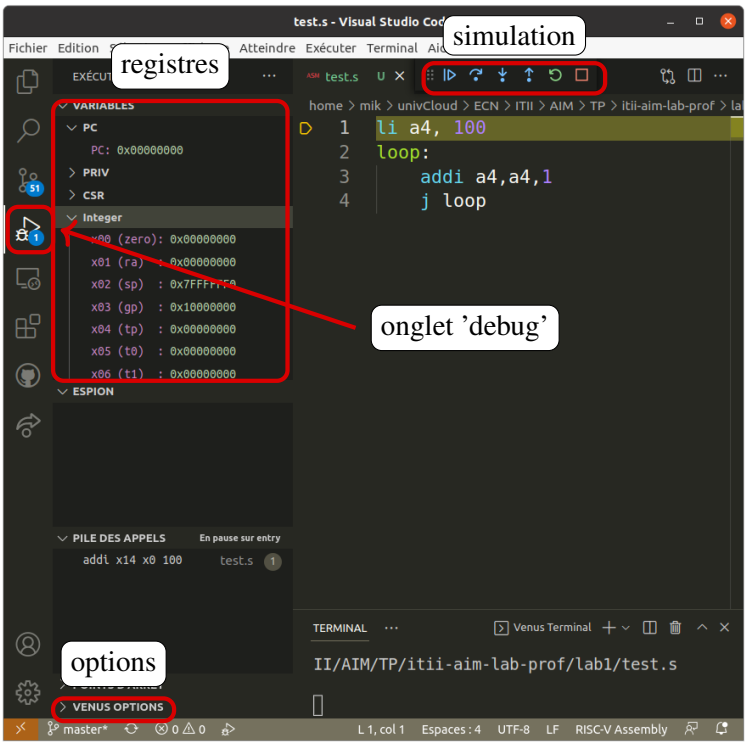

FIGURE 3 – Simulation en cours. Les options permettent d'avoir d'autres vues (mémoire, assembleur, ...)

```
.data
chaine:
.string "Hello, est-ce OK?"
```

On peut utiliser le compilateur en ligne `https://godbolt.org/`, en choisissant un cross-compilateur RISC-V (bien sûr !), et avec les drapeaux de compilation `-march=rv32i -mabi=ilp32` pour comparer l'assembleur produit avec votre solution. il faut rajouter le drapeau `-Ox` qui correspond au niveau d'optimisation, de `-O0` (pas d'optimisation) à `-O3` (optimisations au maximum).

- ▷ avez-vous fait mieux que le compilateur (en terme de nombre d'instructions exécutées) s'il n'y a pas d'optimisation ? si les optimisations sont activées ?
- ▷ valider le fonctionnement en appelant la fonction avec une chaine de caractère (elle se termine par `\0`), et en visualisant la mémoire.

3.2 Des Leds ! !

Le simulateur Venus permet l'utilisation d'un panneau de Led (voir figure 4). Pour accéder à une led, il faut utiliser l'instruction `ecall` avec 3 paramètres :

- `a0` contient la valeur `0x100` (`Set a Led`)
- `a1` contient la position en (x,y). Les 16 bits de poids fort pour `x`, les 16 bits de poids faible pour `y`. Origine en haut à gauche (0,0). (ici `0x00020004` : x=2, y=4)
- `a2` contient la couleur, l'octet de poids fort est à `0`, le suivant indique le `rouge` (256 valeurs), le suivant le `vert`, puis le dernier le `bleu`. (ici `0x0000FF00` :vert)

Le code de la figure 4 donne un exemple d'utilisation.

plus d'info dans l'onglet `Venus Options -> Open documentation`, ou sur `https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus`

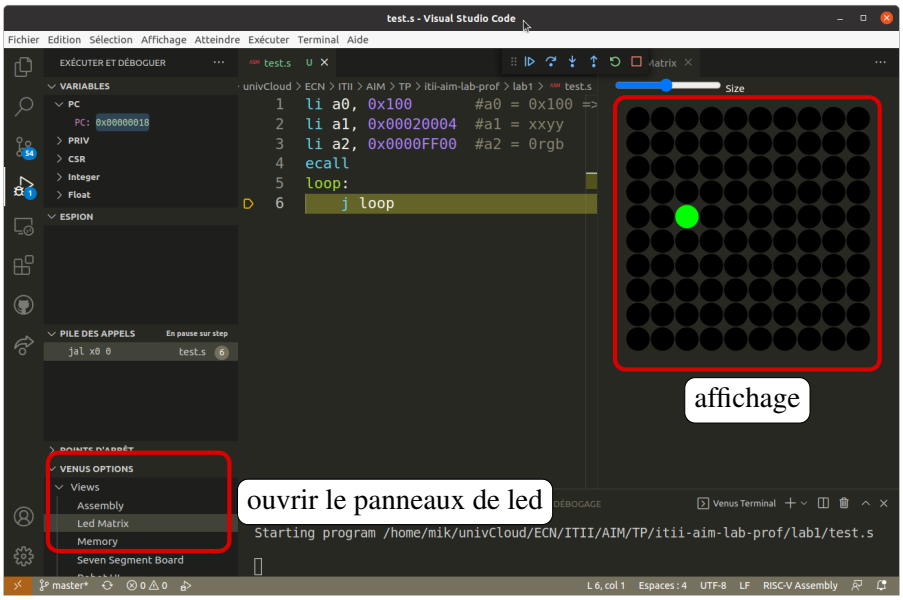

FIGURE 4 – Utilisation de la matrice de leds

▷ Proposer un programme pour faire un dégradé de couleur comme la figure 5, en utilisant des boucles, bien entendu ! L'extension M (multiplication) n'est plus autorisée.

3.3 Séparer flot de contrôle et données...

On souhaite rendre indépendante la partie algorithmique (flot de contrôle) des données affichées (qui seront stockées dans un tableau). Pour définir une zone de données en assembleur, on utilise la directive `.data` :

```
# pour récupérer l'adresse de 'buffer':
la a0, buffer
```

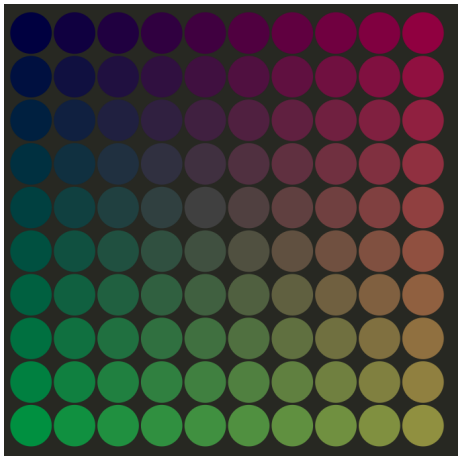

FIGURE 5 – Dégradé de couleur avec la matrice de led

```
.data
buffer:
.byte 0x01 0x23 0x45 0x67 0x89 0xab 0xcd 0xef
```

Ainsi, le code précédent défini un tableau constant de 8 octets, dont l'adresse de départ est `buffer`.

▷ afficher le logo de l'ECN (ou du RISC-V, ou ce que vous voulez...) sur les leds, en utilisant cette approche.

Vous pouvez modifier la taille de la matrice de led si besoin, en créant un fichier `.vscode/launch.json` de la forme suivante :

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "venus",
            "request": "launch",
            "name": "Debug current file",
            "program": "${file}",
            "stopOnEntry": true,
            "openViews": ["LED Matrix"],
            "ledMatrixSize": {
                "x": 10,
                "y": 10
            }
        }
```

```
    ]
}
```

3.4 Fonction récursive...

- ▷ implémenter une fonction qui renvoie la factorielle d'un nombre : `int fact(int n);`. La fonction sera récursive. (il ne faut pas commencer par l'approche avec le compilateur ici !). On autorisera ici l'extension RISC-V M (multiplications).
- ▷ simuler le code pour visualiser l'utilisation de la pile.

3.5 Fonction racine carrée

On souhaite implémenter une fonction de calcul de la racine carrée en assembleur pour des raisons de performances. On autorisera ici l'extension RISC-V M (multiplications).

L'algorithme utilise la méthode de Newton-Raphson, et la convergence ($\sqrt{x} = \lim_{n\to+\infty} U_n(x)$) de la suite :

$$\begin{cases} U_0 = x \\ U_{n+1} = \frac{U_n + \frac{x}{U_n}}{2} \end{cases}$$

On pourrait prendre une autre valeur U_0 pour converger plus vite...

Une implémentation en langage C est disponible en annexe, avc un test pour un nombre en virgule fixe (16.16), c'est-à-dire un codage sur 32 bits, avce 16 bits pour la partie entière et 16 bits pour la partie fractionnaire.

Rappel : avec le calcul en virgule fixe, le "$\times 2^{-16}$" est implicite. Lors du calcul de la racine carrée, cet exposant est modifié (en "$\times 2^{-8}$"). Il faut alors faire un décallage pour ré-encoder la valeur en (16.16).

- ▷ implémenter la fonction `square_root` en assembleur. Vous pouvez valider votre code en comparant vos résultats avec ceux obtenus avec la version en C.