Architecture, Système, Réseau 2019-2020
julien.braine@ens-lyon.fr,mateusz.skomra@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

Devoir à la maison : assembleur et simulateur

1 Rendus attendus

Il y a 3 rendus incrémentaux du même DM. Je vous invite à renégocier collectivement les dates de rendus en cas de conflit avec d'autres coups de bourre.

1. **Avant le mercredi 23 octobre à 23h59**, envoyez par mail la composition de votre binôme
   — à `Florent.de-Dinechin@insa-lyon.fr`
   — avec dans le sujet "ASR1 : groupe",
   — avec les deux membres du binôme en CC.
   Ceux qui n'auront pas répondu seront binômés de force. Pour l'éventuel singleton, on verra.
2. **Avant le mercredi 14 novembre à 23h59**, envoyez par mail votre rendu 1
   — à `Florent.de-Dinechin@insa-lyon.fr`
   — avec dans le sujet "ASR1 : rendu 1",
   — avec les deux membres du binôme en CC (le second membre du binôme est responsable de vérifier ce que le premier envoie),
   — avec en attachement un ficher .tgz qui se décompresse en un répertoire Nom1-Nom2, qui est une copie améliorée du répertoire fourni
   — et rien d'autre. En particulier, si vous avez des explications à donner, un mode d'emploi, etc, mettez-les dans un README dans le tgz, pas dans le mail.
3. **Avant le dimanche 1 décembre 23h59**, envoyez par mail le rendu 2 selon les mêmes modalités que le rendu 1, avec dans le sujet "ASR1 : rendu 2"
4. **Avant le dimanche 15 décembre 23h59**, envoyez par mail le rendu 3 selon les mêmes modalités que le rendu 1, avec dans le sujet "ASR1 : rendu 3"

Le travail demandé pour chaque rendu est précisé page suivante. Chaque rendu est un sur-ensemble du rendu précédent.

2 Documents et code fournis

L'ISA du processeur (qui sera corrigée et mise à jour si nécessaire) s'appelle `isa2019-v1.pdf` sur : `http://perso.citi-lab.fr/fdedinec/enseignement/2019/ASR1`

Vous trouverez également sous le même lien une archive `src_etudiants.tgz` qui contient un squelette de simulateur (écrit en C++) et un squelette d'assembleur (écrit en python)

Tout ce qui suit marche bien sous Linux, le SAV n'est pas assuré pour les autres systèmes.

2.1 Le simulateur

Il y a besoin d'une bibliothèque graphique, voir `Readme.md`. Pour obtenir le simulateur il faut le compiler en tapant `make` dans `simu.src/`.

Puis `./simu` vous donnera un mode d'emploi sommaire.

2.2 Pour démarrer

Créez un fichier `test.obj` qui contient

```
341
341
341
000
```

Par gentillesse j'ai codé chaque instruction sur une ligne. Comprenez quelles sont ces instructions en vous référent à l'ISA.

Lancez `./simu -d -s test.obj` et appuyez plusieurs fois sur entrée. Observez ce qui se passe.

Modifiez dans test.obj la dernière instruction pour qu'elle saute au début, et relancez le simulateur. C'est très facile mais il y a un piège petit-boutiste.

2.3 L'assembleur

Écrire de l'hexadécimal cela va bien 5mn. Essayez `python asm.py` puis `python asm.py test.s`

Comprenez ce qui sort.

2.4 C'est du sabotage

Un script de sabotage a effacé tout ce qui se trouvait entre `begin sabote` et `end sabote` dans `asm.py` et `processor.cpp`. Par bonheur pour vous, il a laissé ces marqueurs : cela vous dit où vous devez travailler.

Cela dit vous êtes encouragés à comprendre tout le code.

Par pure méchanceté, vous devrez donc écrire du python, du C++ et de l'assembleur de notre processeur. Le travail sur `asm.py` et `processor.cpp` est vraiment complémentaire : choisissez qui s'occupe de quoi, et travaillez en même temps sur le support des mêmes instructions.

3 Rendu 1

Vous nous rendrez un tgz dans lequel le simulateur et l'assembleur seront assez réparés pour que la multiplication et la division binaire marchent. Il suffit d'une poignée d'instruction, par exemple pas besoin des accès mémoire, mais vous êtes bien sûr encouragés à avancer plus que cela...

Comme programmes assembleur, vous rendrez

- a minima, une multiplication des deux entiers positifs disponibles dans R2 et R3, résultat dans R4.
- a minima, la division d'un entier positif de 16 bits par un autre de 8 bits (dont l'octet de poids est nul) (r1), résultat dans r2
- en bonus, la multiplication de deux entiers signés, et/ou la multiplication de deux entiers positifs de 16 bits avec le résultat sur 32 bits (r2 et r3)

Vous aurez écrit dans votre répertoire ASM un fichier mult.s et un fichier div.s qui contiennent votre implémentation de la multiplication et de la division.

En commentaire dans ces fichiers, vous indiquerez le nombre total de mioches d'instruction lus dans l'exécution d'une mutiplication.

Vous n'êtes pas obligé de gérer les étiquettes (labels) dans le rendu 1. Cela vous fera même du bien d'assembler des distances de saut à la main une fois dans votre vie.

Si vous avez des critiques constructives sur le jeu d'instructions, écrivez-les dans un fichier `remarques.txt` dans votre tgz.

4 Rendu 2

4.1 Simulateur et assembleur

Tout le jeu d'insruction devra être implémenté dans ses moindres détails. Nous testerons votre simulateur et votre assembleur sur des programmes à nous.

Un support des étiquettes est demandé.

4.2 Une API graphique

Le simulateur fourni inclut une sortie graphique sur un écran de 320x256 pixels qui occupe le haut de la mémoire (adresse exacte dans `screen.h`, et elle s'affiche au lancement avec l'option `-g`). Chaque pixel est défini

par 3 mioches qui encodent une couleur au format RGB (allez lire `screen.cpp` pour comprendre où sont les bits de chaque couleur).

Vous donnerez dans le répertoire ASM un programme qui contient plusieurs routines graphiques (appelables par `syscall`, et il faudra initialiser les vecteurs correspondant). Le système de coordonnées utilisé a le point (0,0) en bas à gauche de l'écran. Voici les routines attendues.

— *syscall 1* est `clear_screen` : efface tout l'écran (la couleur est passée dans r2).
— *syscall 2* est `plot`, qui allume le pixel de coordonnées (r2,r3) à la couleur r4. L'écran faisant 320 pixels de large, il y a une multiplication par 320 à réaliser. Inutile de dégainer votre `mult`, cette multiplication se fait plus vite en une poignée de décalages et additions.
— *syscall 3* est `fill` qui remplit un rectangle de couleur r6, du point de coordonnée (r2, r3) au point de coordonnées (r4, r5) inclus. De préférence, plus vite qu'en utilisant `plot`.
— *syscall 4* est `draw` qui trace une droite du points de coordonnée (r2, r3) au point de coordonnées (r4, r5), toujours à la couleur r6. Documentez vous sur l'algorithme de Bresenham pour cela. Pour le coup vous pouvez utiliser `plot`.
— *syscall 5* est `putchar` qui écrit le caractère dont le code ASCII est dans r4 au point de coordonnée (r2, r3), avec la couleur r5. Deux options ici : soit vous trouvez quelquepart un fichier de bitmap 8x8 (8 octets par caractère) que vous intégrerez dans votre fichier assembleur au moyen d'un script ad-hoc. Soit vous fabriquez une police vectorielle à coups de `draw`. Dans ce cas, putchar prend un argument de plus (la taille) et renvoie dans r2 la position ou l'on peut placer le prochain caractère.

Je demande pour tester tout ceci un programme qui affiche au moins une droite en diagonale de l'écran (pas à 45 degrés mais d'un coin à l'autre), un rectangle et vos initiales dedans.

5 Rendu 3

Avec tout cela vous m'offrirez pour Noël une jolie démonstration graphique que je laisse à votre imagination. Ou un jeu, ou un compilateur, ou tout cela à la fois.

Tout ceci vous permettra d'argumenter (toujours dans `remarques.txt`) combien ce processeur est tout pourri, et comment que vous auriez fait de meilleurs choix si on vous avait laissé vous exprimer.