Architecture, Système, Réseau 2017-2018
{alice.pellet___mary,fabrice.mouhartem}@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

Devoir à la maison : assembleur et simulateur

1 Rendus attendus

1. Avant le dimanche 22 octobre à 23h59, envoyez par mail la composition de votre binôme
   — par *reply* au mail annonçant le DM,
   — 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 dimanche 12 novembre 23h59, envoyez par mail votre rendu 1
   — par *reply* au mail annonçant le DM,
   — 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 29 novembre 23h59, envoyez par mail le rendu 2 selon les mêmes modalités que le rendu 1, avec dans le sujet "ASR1 : rendu 2"

Le travail demandé pour chaque rendu est précisé page suivante.

2 Documents et code fournis

L'ISA du processeur, corrigée et mise à jour :
`http://perso.citi-lab.fr/fdedinec/enseignement/2017/ASR1/isa2017.pdf`
Les squelettes du simulateur et de l'assembleur :
`http://perso.citi-lab.fr/fdedinec/enseignement/2017/ASR1/src_etudiants.tgz`
Tout ce qui suit marche bien sous Linux, le SAV n'est pas assuré pour les autres systèmes.

2.1 L'assembleur

Essayez
`python asm.py`
puis
`python asm.py ASM/test.s`.
Comprenez ce qui sort. Dans `test.s`, remplacez le -17 par la bonne valeur qui créera une belle boucle infinie comptant dans `r3`.

2.2 Le simulateur

Pour obtenir le simulateur il faut le compiler en tapant `make` dans `simu.src/`.
Puis `./simu` vous donnera un mode d'emploi sommaire. Par exemple, essayez
`./simu -d -s ASM/test.obj`

2.3 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...

— a minima, une multiplication de deux entiers positifs contenus dans r0 et r1, renvoie le résultat dans r2, et fonctionne tant que le résultat tient sur 16 bits.
— a minima, la division d'un entier positif de 16 bits (r0) 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 bits d'instruction lus dans l'exécution d'une itération.

Terminez ces routines par `jump -13` : ainsi on peut les lancer par `simu -d mult.obj` et observer dans la console les valeurs des registres r0, r1 et r2.

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. Il a le droit d'être dramatiquement sous-optimal (tous les sauts sur 16 bits), ou de demander au programmeur sur combien de bits un saut est encodé. Si vous le faites mieux, c'est des points de bonus.

Ajoutez au simulateur des compteurs des différentes instructions, des compteurs de nombre de bits d'instruction, de nombre d'instruction, de nombre de bits qui passent sur la mémoire, etc. Bref, tout ce qui vous permettra d'argumenter (toujours dans `critiques.txt`) que ce processeur est tout pourri et que vous auriez fait de meilleurs choix si on vous avait laissé vous exprimer.

4.2 Une API graphique

Le simulateur fourni inclut une sortie graphique sur un écran de 160x128 pixels qui occupe le haut de la mémoire (à partir de l'adresse 0x10000). Chaque pixel est défini par un mot de 16 bits qui encode une couleur au format RGB (allez lire `screen.cpp` pour comprende où sont les bits de chaque couleur).

Vous donnerez dans le répertoire ASM un programme qui contient plusieurs routines graphiques (appelables par `call`). Le système de coordonnées utilisé a le point (0,0) en bas à gauche de l'écran.

— `clear_screen` efface tout l'écran (la couleur est la valeur de r0).

— `plot` allume le pixel de coordonnées (r1, r2) à la couleur r0. L'écran faisant 160 pixels de large, il y a une multiplication par 160 à réaliser. Inutile de dégainer votre `mult`, cette multiplication se fait (vite) en deux décalages et une addition.
— `fill` remplit un rectangle de couleur r0 du point de coordonnée (r1, r2) au point de coordonnées (r3, r4). De préférence, plus vite qu'en utilisant `plot`.
— `draw` trace une droite du points de coordonnée (r1, r2) au point de coordonnées (r3, r4), toujours à couleur r0. Documentez vous sur l'algorithme de Bresenham pour cela. Pour le coup vous pouvez utiliser `plot`.
— `putchar` écrit le caractère dont le code ASCII est dans r3 au point de coordonnée (r1, r2) et de couleur r0. Vous trouverez quelquepart un fichier de bitmap 8x8 (8 octets par caractère) que vous intégrerez dans votre fichier assembleur au moyen d'un script ad-hoc.

Avec tout cela vous me ferez une jolie démonstration graphique que je laisse à votre imagination. C'est sur cette démo que vous ferez des statistiques constructives.