ASR1 2016
Florent de Dinechin, Nicolas Louvet, Antoine Plet

Devoir à la maison

Assembleur et simulateur

Rendus

1. Le 21 octobre, un reply au mail annonçant le DM, avec comme sujet "ASR1 : groupe" contiendra la composition de votre binôme, avec les deux membres du binôme en CC. Pas de trinôme, un monôme sera accepté.
2. Avant le dimanche 13 novembre 23h59, un reply à ce mail avec comme sujet "ASR1 : rendu1", et en attachement un ficher .tgz qui se décompresse en un répertoire Nom1-Nom2, qui est une copie améliorée du répertoire fourni.
3. Avant le dimanche 27 novembre 23h59, un reply à ce mail avec comme sujet "ASR1 : rendu2", et en attachement un ficher .tgz qui se décompresse en un répertoire Nom1-Nom2.

Le travail demandé pour chaque rendu est précisé page suivante.

Documents et code fournis

L'ISA du processeur, corrigée et mise à jour :
`http://perso.citi-lab.fr/fdedinec/enseignement/2016/isa2016.pdf`
Les squelettes du simulateur et de l'assembleur :
`http://perso.citi-lab.fr/fdedinec/enseignement/2016/src_etudiants.tgz`
Tout ce qui suit marche bien sous Linux, le SAV n'est pas assuré pour les autres systèmes.

Le simulateur

Pour obtenir le simulateur il faut le compiler en tapant `make`.
Puis `./simu` vous donnera un mode d'emploi sommaire.

L'assembleur

Essayez `python asm.py toto.s` où `toto.s` est l'un des programmes du TD4. Si cela ne marche pas, réparez `asm.py`.

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, ce n'est pas bien méchant.
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.

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).

— 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.

Terminez ces routines par `jump 0` : ainsi on peut les lancer par `simu r mult.obj` et observer dans la console les valeurs des registres r0, r1 et r2.

Rendu 2

Simulateur et assembleur

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

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 0xb000). Chaque pixel est défini par un mot de 16 bits.

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 la moitié basse de 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.