

INF1600 — TP5

Assembleur en ligne et mémoire

Giovanni Beltrame giovanni.beltrame@polymtl.ca
Carlo Pincioli carlo.pincioli@polymtl.ca
Luca G. Gianoli luca-giovanni.gianoli@polymtl.ca

Remise

Voici les détails concernant la remise de ce travail pratique :

- **Méthode** : sur Moodle (une seule remise par groupe).
- **Échéance** : avant 23h55 ; le 11 avril 2017 pour la section B1, le 18 avril 2017 pour la section B2 ;
- **Format** : un seul fichier zip, dont le nom sera <matricule1>-<matricule2>.zip. Exemple : 0123456-9876543.zip. L'archive doit contenir les fichiers rapport.pdf et tp5.c ; le rapport doit comporter une page titre où figurent les noms et matricules des membres de l'équipe ;
- **Langue écrite** : français.
- **Distribution** : les deux membres de l'équipe recevront la même note.

Barème

Contenu	Points du cours
Assembleur en ligne	3
Mémoire - cache	4
Français écrit erroné	jusqu'à -1
Illisibilité du code (peu de commentaires, mauvaise structure...)	jusqu'à -1
Format de remise erroné (irrespect des noms de fichiers demandés, fichiers superflus, etc.)	jusqu'à -1
Retard	-0,025 par heure

Exercice 1 Assembleur en ligne

Nous voulons envoyer des données à un serveur distant. Pour des raisons de sécurité, certaines données sont très sensibles et donc ont été crypté. Une clé de chiffrement a été ajouté à chaque mot en divisant la clé en deux et mettre chaque moitié à la fin du mot. Votre travail consiste à déchiffrer le code en récupérant les deux parties de la clé et de la mettre au début.

Cependant, nos ordinateurs fonctionnent en mode little-endian. Donc, en plus de décrypter le code, nous avons besoin de convertir le mot de 16 bits à partir big-endian à little-endian.

Une façon de faire en C serait la suivante :

```
unsigned int Decryption_fct(unsigned int le)
{
    return = (le & 0xff000000) | (le&0xff) << 16 | (le & 0xff00) | (le & 0xff0000) >> 16;
```

Malheureusement le compilateur `gcc` génère trop d'instructions pour que notre programme ait de bonnes performances. Vous pouvez voir le code généré en tapant la commande :

```
gcc -m32 -S tp5.c
```

Un fichier `tp5.s` sera généré, il contient le code assembleur correspondant au programme.

Votre travail est d'implémenter la routine `Decryption_fct()` plus efficacement que le compilateur (en utilisant moins d'instructions) via quelques lignes d'assembleur en ligne dans le code C.

Le fichier `tp5.c` qui contient la routine à modifier vous est fourni. Vous pouvez compiler le programme avec la ligne de commande suivante :

```
gcc -Wall -m32 -gdwarf-2 -o tp5 tp5.c
```

et l'exécuter en faisant `./tp5` pour vérifier que votre implémentation est correcte.

Une bonne référence pour l'assembleur en ligne est GCC-Inline-Assembly-HOWTO : <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>

Exercice 2 Mémoire cache

Nous possédons un système embarqué qui comporte une mémoire principale de 1024Ko et une mémoire cache de 16Ko octets.

Nous considérons trois différentes versions de placement :

- direct,
- associative par ensemble de 2 blocs,
- associative par ensemble de 4 blocs.

Dans les trois cas, on considère :

- lignes des 16 octets
- politiques *write-back* et *write-allocate*
- algorithme de remplacement LRU

1. Vous devez écrire comment l'adresse est divisée dans ce système (**pour les 3 versions de placement**). Donnez le nombre de bits réservés pour le tag, l'ensemble et l'octet (**pour les 3 versions de placement**).
2. Remplissez ensuite le tableau suivant pour chaque accès mémoire (**pour les 3 versions de placement**). Les colonnes «Tag» et «Set» doivent comporter les numéros de tag et d'ensemble pour l'accès en question. La colonne «Hit» doit être cochée s'il y a un succès d'accès à la cache. La colonne «w-b» doit être cochée lorsque, pour l'accès de la même rangée, il y a un write-back vers la mémoire principale d'implique.

N'oubliez pas que, étant donné qu'il s'agit d'une cache associative par ensemble de deux/-quatre blocs, il y a un *dirty bit* pour chacun des deux/quatre blocs.

La cache est initialement invalide (*dirty bits* tous à 0). Un accès mémoire RD **x** signifie «lire la mémoire à l'adresse **x**» et WR **x** signifie «écrire en mémoire à l'adresse **x**».

Accès	Direct				2 blocs				4 blocs			
	Tag	Set	Hit	w-b	Tag	Set	Hit	w-b	Tag	Set	Hit	w-b
WR 0x5EF1D												
WR 0x19C7C												
RD 0x5EF1B												
RD 0x8CDB0												
WR 0x3CDB3												
WR 0x5EF15												
RD 0x68DBF												
WR 0xCAF1C												
RD 0x39C7E												
WR 0xCAF1A												

3. Donnez l'état de la cache à la fin de ces accès. Le format devrait ressembler au tableau (où l'astérisque * représente la présence du *dirty bit* pour un bloc). **Un tableau pour chaque version de placement. Modifier le nombre de colonnes «TagX» en conséquence.**

Set	Tag0	Tag1
FC	56*	02
89	09	14*

4. En supposant qu'un succès d'accès à la cache prenne 8 ns et qu'un défaut ou un accès à la mémoire principale prenne 100 ns, donnez le temps d'accès effectif de la cache pour la totalité de cette série d'accès pour chaque politique de placement.
5. Expliquez comment la structure de l'adresse aurait changée si la politique de placement de la cache était complètement associative ? Pourquoi ?