IFT209 – Programmation système

Université de Sherbrooke

Devoir 3

Enseignant:	Michael Blondin
Date de remise:	mercredi 20 février 2019 à 23:59
À réaliser:	en équipe de deux
Modalités:	remettre en ligne sur Turnin; une seule remise avec vos noms/CIP en commentaires en en-tête du code

Problème. Le but de ce devoir est d'implémenter un calculateur composé d'un accumulateur et de trois celulles de mémoire. Ce calculateur permet de stocker des valeurs entières dans son accumulateur et sa mémoire, et d'effectuer des affectations, des additions signées et des soustractions signées.

L'accumulateur et la mémoire du calculateur sont respectivement dénotés par `acc` et `mem`. La mémoire possède trois celulles dénotées `mem`[0], `mem`[1] et `mem`[2]. L'accumulateur et les trois celulles mémoire stockent des entiers *signés* de *128 bits*. Lorsque le programme démarre, l'accumulateur et les celulles mémoires sont initialisés à 0, et leur contenu s'affiche à l'écran dans le format hexadécimal:

```
acc:    0000000000000000 0000000000000000
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 0000000000000000
```

L'utilisatrice peut ensuite entrer au clavier un code d'opération numérique compris entre 0 et 5, suivi d'un opérande, dont l'effet est le suivant:

code d'opération	**opérande**	**effet**
0	entier non signé n de 64 bits	`acc` $\leftarrow n$
1	entier non signé n de 64 bits tel que $n \in \{0,1,2\}$	`mem`$[n] \leftarrow$ `acc`
2	entier non signé n de 64 bits tel que $n \in \{0,1,2\}$	`acc` $\leftarrow$ `mem`$[n]$
3	entier non signé n de 64 bits tel que $n \in \{0,1,2\}$	`mem`$[n] \leftarrow$ `mem`$[n] +$ `acc`
4	entier non signé n de 64 bits tel que $n \in \{0,1,2\}$	`mem`$[n] \leftarrow$ `mem`$[n] -$ `acc`
5	aucun	quitte le programme

Après l'exécution d'une opération, l'état du calculateur est à nouveau affiché à l'écran et l'utilisateur peut à nouveau effectuer une opération.

Exemple. Si l'utilisatrice entre:

```
0
255
```

alors l'accumulateur prend la valeur 255, et ainsi l'état du calculateur devient:

```
acc:    0000000000000000 00000000000000FF
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 0000000000000000
```

Tant que l'utilisateur n'entre pas le code 5 afin de quitter, le programme lit un nouveau code suivi d'un opérande. Par exemple, si l'utilisateur entre:

```
1
2
```

alors la valeur de l'accumulateur est copiée dans `mem`[2], et ainsi l'état du calculateur devient:

```
acc:    0000000000000000 00000000000000FF
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 00000000000000FF
```

Si l'utilisatrice entre ensuite:

```
0
1
```

l'état du calculateur devient:

```
acc:    0000000000000000 0000000000000001
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 00000000000000FF
```

Le code d'opération 3 permet d'additionner le contenu de l'accumulateur à une cellule mémoire. Par exemple, si l'utilisateur entre:

```
3
2
```

l'état du calculateur devient:

```
acc:    0000000000000000 0000000000000001
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 0000000000000100
```

Le code d'opération 4 permet de soustraire le contenu de l'accumulateur d'une cellule mémoire. Par exemple, si l'utilisatrice entre:

```
4
1
```

alors `mem`[1] prend la valeur -1, et ainsi l'état du calculateur devient:

```
acc:    0000000000000000 0000000000000001
mem[0]: 0000000000000000 0000000000000000
mem[1]: FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
mem[2]: 0000000000000000 0000000000000100
```

Le code d'opération 2 permet de copier le contenu d'une cellule mémoire dans l'accumulateur. Par exemple, si l'utilisateur entre:

```
2
1
```

alors `acc` prend la valeur de `mem`[1], et ainsi l'état du calculateur devient:

```
acc:    FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
mem[0]: 0000000000000000 0000000000000000
mem[1]: FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
mem[2]: 0000000000000000 0000000000000100
```

Lorsque l'utilisatrice entre finalement le code d'opération 5, aucun opérande n'est lu et le programme se termine.

Directives.

- Votre programme doit être obtenu en complétant le code partiel de la page suivante;
- Votre programme doit être remis dans un seul fichier nommé `devoir3.s`;
- Ne modifiez pas le point d'entrée ainsi que le format des entrées et sorties;
- Vous pouvez supposer que les valeurs en entrée sont valides; en particulier seuls les codes d'opérations 0 à 5 seront entrés, vous n'avez pas à faire de validation;
- Vous n'avez pas à signaler le report ou le débordement d'une opération arithmétique, par exemple si l'accumulateur contient le plus grand entier signé positif de 128 bits, c'est-à-dire `7FFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF`, et que vous lui additionnez 1, alors l'accumulateur prend la valeur négative `8000000000000000 0000000000000000` et le débordement n'est pas signalé;
- L'espace dans l'affichage des nombres hexadécimaux n'a que pour seul but de faciliter la lecture; par exemple si l'accumulateur contient `0000000000000010 000000000000000F`, alors il s'agit du nombre $16^{17} + 15$ (et non de deux nombres distincts).

Pointage. Vous pouvez obtenir un maximum de 20 points. Vous obtenez:

- 2 points si votre programme lit correctement un code d'opération et son opérande;
- 2 points si votre programme affiche correctement l'état du calculateur;
- 2 points par opération bien implémentée (donc 12 points au maximum);
- 2 points si votre code est bien indenté (codes d'opération, opérandes et commentaires alignés);
- 2 points pour la présence de commentaires significatifs facilitant la lecture et compréhension du code.

Si l'implémentation d'une fonctionnalité est incorrecte et/ou partielle, vous pourrez tout de même obtenir jusqu'à 1,75/2 points selon la gravité du problème.

Bonus.

Vous obtenez des points bonus si vous implémentez correctement l'opération supplémentaire suivante:

code d'opération	**opérande**	**effet**
6	entier non signé n de 64 bits tel que $n \in \{0, 1, 2\}$	`mem`$[n] \leftarrow$ `mem`$[n] \cdot$ `acc`

Vous obtenez 0,25 point bonus (absolu) à votre note finale de la session si votre implémentation supporte la multiplication de nombres compris entre 0 et $2^{63} - 1$; et 0,5 point bonus (absolu) additionnel si votre implémentation supporte la multiplication de nombres compris entre -2^{63} et $2^{63} - 1$.

Code partiel.

```
.global main

main:
        /*
           code ici
                    */

        bl      exit

.section ".rodata"
fmtOpcode:      .asciz  "%lu"
fmtOperande:    .asciz  "%lu"
fmtAcc:         .asciz  "acc:    %016lX %016lX\n"
fmtMem:         .asciz  "mem[%lu]: %016lX %016lX\n"
```