IFT209 – Programmation système

Université de Sherbrooke

Devoir 3

Enseignant:	Michael Blondin
Date de remise:	mercredi 15 février 2023 à 23h59
À réaliser:	en équipe de deux
Modalités:	remettre en ligne sur Turnin
Pointage:	sur 20 points + 1,5 point bonus

Le but de ce devoir est d'implémenter un calculateur rudimentaire afin de mettre en pratique l'accès mémoire et l'arithmétique signée sur des entiers excédant 64 bits.

Problème. Considérons un calculateur composé d'un accumulateur et de quatre celulles de mémoire. Ce calculateur stocke des valeurs entières et peut effectuer des affectations, des additions et des soustractions.

L'accumulateur et la mémoire sont respectivement dénotés par « `acc` » et « `mem` ». La mémoire possède quatre celulles dénotées `mem`$[0]$, `mem`$[1]$, `mem`$[2]$ et `mem`$[3]$. L'accumulateur et les quatre celulles mémoire stockent des entiers *signés* de *128 bits*. Lorsque le calculateur démarre, son accumulateur et ses celulles mémoires sont initialisées à 0, et leur contenu s'affiche à l'écran sous leur valeur hexadécimale:

```
acc:    0000000000000000 0000000000000000
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 0000000000000000
mem[3]: 0000000000000000 0000000000000000
```

Après le démarrage, on peut entrer au clavier un *code d'opération* numérique compris entre 0 et 5 (inclusivement), suivi d'un *opérande*, dont l'effet est le suivant:

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

Après l'exécution d'une opération, l'état du calculateur est à nouveau affiché à l'écran et on peut à nouveau effectuer une opération (à moins d'avoir entré « 5 » auquel cas le programme termine).

Exemple. Si l'on 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
mem[3]: 0000000000000000 0000000000000000
```

Le programme lit un nouveau code suivi d'un opérande. Par exemple, si l'on 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
mem[3]: 0000000000000000 0000000000000000
```

Si l'on entre ensuite:

```
0
1
```

l'état du calculateur devient:

```
acc:    0000000000000000 0000000000000001
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 00000000000000FF
mem[3]: 0000000000000000 0000000000000000
```

Le code d'opération 3 additionne l'accumulateur à une cellule mémoire. Par exemple, si l'on entre:

```
3
2
```

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

```
acc:    0000000000000000 0000000000000001
mem[0]: 0000000000000000 0000000000000000
mem[1]: 0000000000000000 0000000000000000
mem[2]: 0000000000000000 0000000000000100
mem[3]: 0000000000000000 0000000000000000
```

Le code d'opération 4 soustrait l'accumulateur d'une cellule mémoire. Par exemple, si l'on 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
mem[3]: 0000000000000000 0000000000000000
```

Le code d'opération 2 copie le contenu d'une cellule mémoire dans l'accumulateur. Par exemple, si l'on 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
mem[3]: 0000000000000000 0000000000000000
```

Lorsqu'on 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;
— Supposez que les valeurs en entrée sont valides;
— Ne signalez pas les reports et les débordements. Par exemple, si l'accumulateur contient le plus grand entier signé positif de 128 bits, c.-à-d. `7FFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF`, et qu'on lui additionne 1, alors l'accumulateur prend la valeur négative `8000000000000000 0000000000000000`;
— 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 $16^{17} + 15 = 295147905179352825871$ (et non de deux nombres distincts).

Remarque.

Si vous allouez $8n$ octets dans la section « `.bss` » sous une étiquette « `tab:` », alors vous pouvez accéder à son contenu comme suit:

```
adr  x19, tab
ldr  x20, [x19]      // x20 = 8 premiers octets de tab  (octets  0 à  7)
ldr  x21, [x19, 8]   // x21 = 8 octets suivants de tab  (octets  8 à 15)
ldr  x22, [x19, 16]  // x21 = 8 octets suivants de tab  (octets 16 à 23)
// ...               // ...
```

Pointage. Vous pouvez obtenir jusqu'à 20 points répartis ainsi:

- 0,5 point pour la lecture des codes d'opération et des opérandes;
- 1 point pour l'affichage de l'état du calculateur;
- 2,5 points pour chaque opération fonctionnelle (donc 15 points au maximum);
- 1 point pour l'indentation du code (étiquettes, codes d'opération, opérandes et commentaires alignés);
- 2,5 points pour la qualité et la lisibilité du code (commentaires significatifs, organisation du code, pas de code répétitif, pas de « code spaghetti », etc.)

Bonus.

Vous obtenez des points bonus si vous implémentez correctement les opérations supplémentaires suivantes:

code d'opération	opérande	effet
6	entier non signé i de 64 bits tel que $i \in \{0, 1, 2, 3\}$	$\texttt{mem}[i] \leftarrow \texttt{mem}[i] \cdot \texttt{acc}$
7	entier non signé i de 64 bits tel que $i \in \{0, 1, 2, 3\}$	$\texttt{mem}[i] \leftarrow \texttt{mem}[i] \div \texttt{acc}$

Dans les deux cas, on suppose que $\texttt{mem}[i]$ est représentable sur 64 bits. Plus précisément, vous obtenez 0,75 point bonus si votre implémentation supporte la multiplication de nombres compris dans l'intervalle $[-2^{63}, 2^{63}-1]$; et 0,75 point bonus si elle supporte la division de nombres compris dans l'intervalle $[-2^{63}, 2^{63}-1]$. Vous ne pouvez pas invoquer d'instruction de multiplication ou de division pour y arriver (c'est le défi). Dans certains cas, il pourrait être pratique d'utiliser des manipulations de bits qui n'ont pas encore été couvertes, mais qui apparaissent au sommaire ARMv8.

Code partiel.

```
.global main

// ...
main:
    /*
        code ici
                  */

    mov     x0, 0
    bl      exit

.section ".rodata"
fmtAcc:         .asciz  "acc:    %016lX %016lX\n"
fmtMem:         .asciz  "mem[%lu]: %016lX %016lX\n"
/*
    autres données ici
                       */
```