INF1600

Travail pratique 3

Lien C++ et Assembleur

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal

1 TABLE DES MATIERES

2 INTRODUCTION ET SOMMAIRE

Ce travail pratique vise à approfondir votre compréhension de l'assembleur IA-32, en utilisant la syntaxe AT&T. Il comprendra des exercices portant sur la manipulation de différents types de données en assembleur pour mieux comprendre le lien entre le langage C et l'assembleur. Un autre exercice consistera à explorer les concepts avancés de l'assembleur en relation avec le C++, notamment le name mangling, l'héritage, et les méthodes virtuelles.

Dans la première partie de ce travail pratique, vous serez amené à écrire des programmes en assembleur impliquant la manipulation de différents types de données, mettant en lumière les nuances de la représentation des données au niveau du matériel.

Ensuite, vous implémenterez des méthodes de classe en assembleur, en vous appuyant sur les spécifications fournies en C++. Vous explorerez ainsi des aspects avancés tels que le name mangling pour comprendre comment les noms de fonctions sont décorés et gérés lors de la compilation.

Enfin, la dernière partie de ce travail pratique portera sur la mise en œuvre de concepts de classes parentes et enfants en assembleur, vous permettant de comprendre comment l'héritage et les méthodes virtuelles sont réalisés au niveau du langage machine.

2.1 REMISE

Voici les détails concernant la remise de ce travail pratique :

- Méthode : sur Moodle, une seule remise par équipe, incluant les **fichiers sources** que vous modifiez en un seul fichier compressé. Une pénalité de 0,5 pt sera appliquée si votre code ne figure pas dans des fichiers séparés (**un fichier assembleur par exercice**).
- Format des fichiers sources : Modifiez les fichiers assembleurs demandés dans les dossiers désignés à chaque question, puis compresser le tout en un seul fichier source compressé en format .ZIP que vous devez nommer comme suit : `<matricule1>-<matricule2>-<tp3>.<zip>`

Attention :

L'équipe de deux que vous avez formé pour le TP1 est la même pour ce TP et la suite des TPs de cette session.

2.2 BARÈME

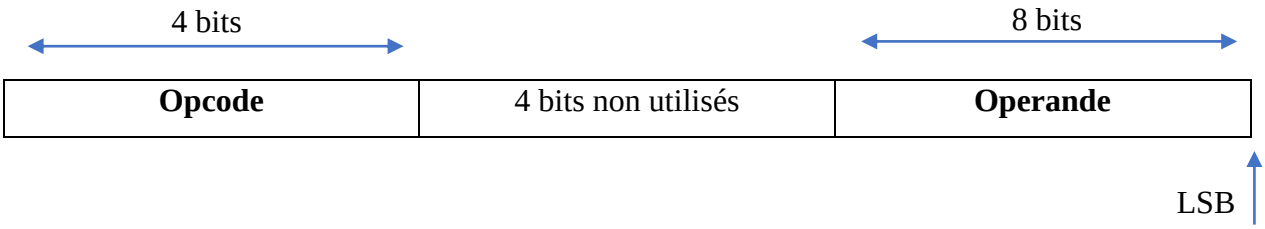
Les travaux pratiques 1 à 5 sont notés sur 4 points chacun, pour un total de 20/20. Le TP3 est noté selon le barème suivant. Reproduisez ce tableau dans le document PDF que vous remettrez.

TP 3				**/4,00**
PROC ACC				/0,75
	Q1		/0,75	
MATRICE OP				/2,75
	Q1		/0,75	
	Q2		/2.0	
		Q2.1	/1,5	
		Q2.2	/0,5	
DETERMINANT				/0,50
	Q1		/0,5	

3 PROCESSEUR ACCUMULATEUR

Pour ce premier exercice, il sera question d'implémenter une simulation du processeur accumulateur comme vous l'avez vu durant les TP 0 et 1, permettant de trouver l'état de la mémoire après exécution:

Pour ce faire nous avons décidé d'encoder nos instructions sur 16 bits (2 octets), les 4 derniers bits ont été réservé pour l'opcode c'est le nombre qui déterminera le type d'action à réaliser, les 8 premiers bits ont été réservé pour les opérandes (adresses) et les 4 bits entre l'opcode et l'opérande sont à négligé.
Voici donc le format de vos instructions encodées :



Dans le répertoire proc_acc vous trouverez un fichier execute_acc.s que vous devez compléter, ce fichier permet la simulation du processeur accumulateur. Vous trouverez un autre fichier aussi, editor.cm ce fichier contiendra le code qui sera simulé. Initialement le fichier contient un programme code machine qui permet de calculer la somme de 1 jusqu'à 10 inclusivement.

Pour exécuter votre code assembleur, nous avons fait appel aux fonctions correspondantes dans le fichier `proc_acc/proc_acc.c`. **Vous ne devez pas modifier ce fichier**. Pour lancer le programme, exécutez les commandes suivantes dans le terminal, en vous trouvant dans le répertoire courant de l'exercice :

```
$ make clean
$ make
$ make run
Ou pour debugger:
$ make debug
```

Q1/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `execute_acc.s` qui doit accéder à un tableau qui représente la mémoire et le modifier selon les instructions faites.

Lorsque vous lancez le programme, le code en C fourni est exécuté en simultané avec votre implémentation en assembleur. Ce code permet de lire du fichier editor.cm, puis de créer une mémoire avec les instructions encodées sur 16 bits qu'il passera en paramètre à la fonction assembleur execute_acc. Vérifiez sa signature qui est déclarée dans le ficher proc_acc.h.

Voici un exemple d'exécution:

editor.cm	mémoire avant exécution	mémoire après exécution
.text	0xF000	0xF000
start:	0xF000	0xF000
ld sum	0x400F	0x400F
add top	0x000D	0x000D
st sum	0x300F	0x300F
ld top	0x400D	0x400D
sub one	0x100E	0x100E
brz end	0x600A	0x600A
st top	0x300D	0x300D
br start	0x5001	0x5001
end:	0xF000	0xF000
stop	0x8000	0x8000
.data	0xF000	0xF000
top: 10	0x000A	0x0001
one: 1	0x0001	0x0001
sum: 0	0x0000	0x0037

4 MATRICES

Une matrice est une structure de données rectangulaire composée de nombres disposés en lignes et en colonnes. Chaque élément de la matrice est identifié par ses coordonnées, qui correspondent à sa position dans la ligne et la colonne.

Soit A une matrice de taille $m \times n$, dans cette matrice A, a_{ij} représente l'élément situé à la i-ème ligne et la j-ème colonne. Les matrices peuvent avoir différentes dimensions, telles que $m \times n$, où m est le nombre de lignes et n est le nombre de colonnes.

Dans cette partie vous allez devoir implémenter des méthodes qui permettent de manipuler des matrices.

Pour exécuter votre code assembleur, nous avons fait appel aux fonctions correspondantes dans le fichier `matrix_computing/main.c`. Ce fichier contient déjà des fonctions de test qui permettent de vérifier l'implémentation de vos fonctions assembleur. Pour lancer le programme, exécutez les commandes suivantes dans le terminal, en vous trouvant dans le répertoire courant de l'exercice :

```
$ make clean
$ make
$ make run
```

4.1 MATRICES
PARTIE 1 : OPÉRATIONS MATRICIELLES

Nous allons définir uniquement trois opérations pour les matrices.

L'opération d'addition :

Soit A une matrice $m \times n$ et B une matrice de $p \times q$, alors l'opération d'addition n'est définie que pour m = p et n = q, en d'autres mots on ne peut additionner que des matrices de même taille. Si les deux matrices sont de même taille $m \times n$, alors l'opération d'addition produit une matrice C de taille $m \times n$, tel que chaque élément c_{ij} de C est défini de la sorte:

$$c_{ij} = a_{ij} + b_{ij}$$

L'opération de multiplication par un scalaire :

Soit A une matrice $m \times n$ et α un scalaire (un nombre réel), alors l'opération de multiplication par un scalaire produit une matrice C de taille $m \times n$, tel que chaque élément c_{ij} de C est défini de la sorte:

$$c_{ij} = \alpha \times a_{ij}$$

L'opération de multiplication par une autre matrice :

Soit A une matrice $m \times n$ et B une matrice de $p \times q$, alors l'opération de multiplication $A \times B$ n'est définie que pour n = p, en d'autres mots on ne peut multiplier une matrice par une autre si le nombre de colonnes de A est égal au nombre de lignes de B. **Attention: *$A \times B$! = $B \times A$. (en général).*** Si n = p, alors l'opération de multiplication produit une matrice C de taille $m \times q$, tel que chaque élément c_{ij} de C est défini de la sorte:

$$c_{ij} = \sum_{k=0}^{n} (a_{ik}\, b_{kj})$$

Dans le cadre de ce TP, il sera question d'implémenter des méthodes qui permettent de réaliser ces opérations. Les trois opérations devront être implémentées en surchargeant les opérateurs de + et de * qui servent pour l'addition et la multiplication respectivement. La surcharge de l'opérateur d'addition vous est fournie sous forme d'assembleur dans le fichier operator_matrix_pl.S, pour vous aider à comprendre les conventions d'appels/retours utilisées (la majorité des lignes du fichiers sont commentées pour vous aider à comprendre).

La fonction « findAddrElem » qui est disponible dans le fichier find_addr_elem.S vous est également fournie, cette fonction permet de trouver l'adresse d'un élément à partir d'un tableau de tableaux, elle est notamment utilisée lors de la surcharge de l'opérateur d'addition en assembleur. Voici la signature équivalente de la fonction en C :

```
float* findAddrElem(float** matrix, int i, int j);
```

Q1/ Vous devez implémenter la surcharge de l'opérateur de multiplication par un scalaire en assembleur en complétant le fichier source operator_scalar_ml.S.

Q2/

Q2.1/ Dans cette section, il s'agit d'implémenter la surcharge de l'opérateur de multiplication par une autre matrice en assembleur en complétant le fichier source operator_matrix_ml.S.

Q2.2/ On vous demande aussi, de gérer les cas non possibles et de juste appeler la fonction erreur correspondante : notMultiplicableError, qui prends en paramètre le nombre de ligne de la deuxième matrice.

Note : Voir annexe pour bien comprendre les conventions d'appels ou le name mangling.

Questions intéressantes à se poser :

1. Quelle est la taille d'une instance de Matrix ? Comment sont repartis les données dans cet objet ?
2. Pourquoi la fonction findAddrElem utilise imul pour trouver le prochain élément ? Quel serait l'alternative naturelle à cette fonction ?
3. Comment pourrait-on retourner un pointeur vers l'objet sans que l'appeleur ai alloué de l'espace sur la pile d'avance ? Est-ce que c'est possible sur la pile ?

4.2 MATRICE
PARTIE 2: DÉTERMINANT D'UNE MATRICE

Dans cette partie nous nous intéressons uniquement aux matrices carrées de taille 2.

Définition :

Soit A une matrice carrée de taille $n \times n$, le déterminant determinant(A) est défini comme une somme pondérée des produits des éléments de A qui s'ajuste avec un certain modèle d'arrangements. Dans le cas d'une matrice 2×2 :

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a} & \boldsymbol{b} \\ \boldsymbol{c} & \boldsymbol{d} \end{bmatrix}$$

determinant(A) est définie par : ad – bc.

Une classe dérivée SquareMatrix de la classe Matrix est utilisée pour créer des matrices carrées 2×2. Cette classe contient deux méthodes virtuelles, une de ces méthodes est à implémenter, c'est celle du déterminant.

Q1/ Vous devez compléter la méthode *determinant* de SquareMatrix qui est définie dans le fichier assembleur determinant.S. À chaque fois que cette méthode est appelée vous devez changer l'attribut _determinant.

Questions intéressantes à se poser :

Quelle est la taille d'une instance SquareMatrix ? Comment ses attributs sont ordonnés dans la mémoire ?

ANNEXE:

1. Pour les conventions d'appels et de retours nous utilisons celles de cdecl, celles-ci spécifient que pour pouvoir retourner un objet c'est à l'appeleur de faire de l'espace localement sur la pile et ensuite de passer le pointeur vers cet objet en le passant comme premier paramètre. Pour plus d'informations, lire les conventions d'appels liées à cdecl, notamment dans la section « Varias ».

2. Pour le name mangling, si la méthode est une méthode **const** de classe, alors un **K** s'ajoute au name mangling avant le nombre de lettre de la classe, par exemple : _ZNK au lieu de _ZN. Pour plus d'informations sur le name mangling, je vous invite à vous référer au cours. Pour plus de détail voire le code source ou cette documentation open source.

3. Pour plus d'informations sur les instructions du x86 il y'a cette documentation non-officielle qui contient pas mal d'information. Sinon pour plus de détail voir la documentation officielle.