INF1600

Travail pratique 2

Architecture IA-32 et ASM

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal

1 Introduction et sommaire

Ce travail pratique a pour but de vous familiariser avec les instructions d'assembleur IA-32 selon la syntaxe AT&T. Vous aurez à manipuler des chaines de caractères et effectuer des calculs en virgule fixe et flottante. Les problèmes considérés traitent l'adressage de la mémoire, la gestion de la pile et les instructions arithmétiques de division et multiplication.

1.1 Remise

Voici les détails concernant la remise de ce travail pratique :

- Méthode : sur Moodle, une **seule remise par équipe**, incluant un rapport **PDF**. **Seules des équipes de deux (2) étudiants sont tolérées**, sauf avis contraire.
- Format: un dossier compressé intitulé `<matricule1>-<matricule2>-<tp1_section_X>.ZIP`, incluant les sources de vos programmes en C et en assembleur et vos réponses aux questions.
- **Attention :** L'équipe de deux que vous formez pour ce TP sera **définitive** jusqu'au TP5. Il **ne sera pas possible** de changer d'équipe au cours de la session (sauf avis contraire).

1.2 Barème

Le travaux pratiques 1 à 5 sont notés sur 4 points chacun, pour un total de 20/20. Le TP2 est noté selon le barème suivant. Reproduisez ce tableau dans le document PDF que vous allez remettre.

TP 2			**/4,00**
Partie 2			/2.1
	Q 2.2.1	/0,25	
	Q 2.2.2	/0,50	
	Q 2.2.3	/0.10	
	Q 2.3.1	/0,50	
	Q 2.3.2	/0,25	
	Q 2.4	/0.50	
Partie 3			/2,25
	Q 3.1.1	/0,75	
	Q 3.1.2	/0,15	
	Q 3.2.1	/1,00	

2 Manipulation des chaines de caractères

Cette partie traite l'utilisation des instructions assembleurs de la famille IA-32 pour manipuler les chaines de caractères.

2.1 Famille d'instructions IA-32 pour les chaines de caractères.

Intel a créé une famille complète d'instructions à utiliser pour travailler avec des données de type « String ». Cette section introduit un nombre d'instructions assembleur pour manipuler les chaînes de caractères de l'IA-32.

Ce type d'instructions utilisent les registres `%ESI` et `%EDI` comme pointeurs pour effectuer les opérations : Cela veut dire que ces registres doivent pointer vers les emplacements mémoires des chaines de caractères à vouloir traiter. À chaque fois qu'une instruction est utilisée (Voir tableau ci-dessous), les registres `%ESI` et `%EDI` sont automatiquement modifiés. La modification implique soit une incrémentation ou une décrémentation automatique de leurs valeurs, par la taille des données spécifiées, selon la valeur du Flag `DF` dans le registre `EIFLAGS.`

- Si le Flag `DF` est effacé (mise à 0), `%ESI` et `%EDI` sont incrémentés après chaque instruction de chaine de caractère, par la taille des données spécifiées.
- Si le Flag `DF` est activé (mise à 1), `%ESI` et `%EDI` sont décrémentés après chaque instruction de chaine de caractère, par la taille des données spécifiées.

Le tableau 1 explique les 5 instructions principales pour le traitement des chaînes de caractères : Les suffixes *B*, *W* et *L* spécifient la taille des données à traiter : *B* pour *Byte* (8 bits), *W* pour *Word* (16 bits) et *L* pour *Long* (32 bits).

Table 1: Instructions assembleur(IA-32) pour le traitement des chaines de caractères

Mnémonique	Signification	Utilisation

LODS	Déplace la valeur de la chaîne en mémoire d'adresse `%ESI` dans le registre `%AL/%AX/%EAX` selon la taille des données spécifiées.	`LODSB`
		`LODSW`
		`LODSL`

STOS	Déplace la valeur de la chaîne en `%AL/%AX/%EAX` selon la taille des données spécifiées dans la mémoire d'adresse `%EDI`.	`STOSB`
		`STOSW`
		`STOSL`

MOVS	Déplace la valeur de la chaine en mémoire d'adresse `%ESI`, vers l'emplacement mémoire d'adresse `%EDI.`	`MOVSB`
		`MOVSW`
		`MOVSL`

SCAS	Compare la valeur de la chaine en mémoire d'adresse `%EDI`, avec le registre `%AL/%AX/%EAX` selon le type spécifié.	`SCASB`
		`SCASW`
		`SCASL`

CMPS	Compare la valeur de la chaine en mémoire d'adresse `%ESI`, avec la valeur de la chaine en mémoire d'adresse `%EDI`.	`CMPSB`
		`CMPSW`
		`CMPSL`

La mise à zéro ou à un du drapeau DF se fait par biais des instructions suivantes :

Instruction	Description
`CLD`	`DF` = 0 (routine d'incrémentation de `%esi` et `%edi`)
`STD`	`DF` = 1 (routine de décrémentation de `%esi` et `%edi`)

Une autre instruction souvent utilisée pour manipuler les chaines de caractères est l'instruction `REP`. Elle est utilisée pour répéter une instruction de chaîne un nombre spécifique de fois, contrôlé par la valeur du registre `%ECX`, de la même manière qu'une boucle, mais sans l'instruction `LOOP` supplémentaire.

L'instruction `REP` répète l'instruction de chaine qui la suit immédiatement jusqu'à ce que la valeur du registre `%ECX` soit égale à zéro. Elle est généralement appelée un préfixe. L'exemple suivant illustre son utilisation :

```
movl $23,%ecx    # Nombre d'itérations
rep movsb        # Répéter l'instruction movsb 23 fois
```

Il existe des instructions `REP` qui vérifient l'état du Flag zéro (ZF) pour répéter l'instruction. Le tableau suivant décrit d'autres instructions `REP` qui peuvent être utilisées :

Table 2: Autres instructions de type REP

Instruction	Description
`REPE`	Répéter tant que égale
`REPNE`	Répéter tant que ce n'est pas égale

Le code ASCII permet de définir 128 codes numériques, donc 128 caractères. Les 32 premiers codes, de 0 à 31, ne sont pas des caractères imprimables mais des caractères "de contrôle". À partir du code 32, suivent des signes de ponctuation et quelques symboles mathématiques comme ! ou + ou /, puis les chiffres arabes de 0 à 9, ainsi que les 26 lettres de l'alphabet latin, en capitales puis en minuscules.

Voici la table des codes ASCII d'utilité pour cette partie :

Caractères	ASCII (en hexadécimal)
Lettres minuscules : de '`a`' à '`z`'	`0x61` à `0x7a`
Lettres majuscules : de '`A`' à '`Z`'	`0x41` à `0x5a`

La manipulation des chaines de caractères demande dans certains cas de manipuler les codes *ASCII* pour parvenir à faire certaines tâches. L'une d'elles serait de pouvoir aller chercher n'importe quel caractère à partir d'un caractère initial.

2.2 GDB:

GDB est un déboggeur permettant d'observer ce qui se passe dans notre programme assembleur. Il donne accès aux valeurs de tous les registres à chaque ligne de code exécutée par le fichier ASM.

Voici une vidéo expliquant rapidement comment utilisée GDB :

Dans la partie 2.2, vous devez fixer un code assembleur qui ne fonctionne pas ou qui n'effectue pas ce qui est demandé. Voici ce que le code effectue :

- Le programme possède une étiquette de type ASCII avec deux mots séparés par un espace
    - Ex: strings: .asciv "Allo Hi"

- Pour l'étiquette, on calcule la somme de chaque lettre de chaque mot
    - Allo → 65 + 108 + 108 + 111 = …

- Ensuite on effectue le rapport de proportion suivant : $\frac{string1}{string2}$. Donc avec l'exemple : $\frac{Allo}{Hi}$.
- Si le rapport est supérieur à 1, cela veut dire que la valeur de Allo est plus grande que Hi, donc on « print » Hi.

Q2.2.1/ Identifier ce que font chacune des étiquettes suivantes dans le programme.

Indice : Chaque lettre correspond au début du mot en anglais. Ex : s_c pourrait être « scalar_product ».

Note : si un « 2 » est dans à la fin de l'étiquette cela veut juste dire qu'on effectue une deuxième fois le même principe que l'étiquette avant. Ex : s_c2 veut dire scalar_product2.

n_c :

d :

r :

e_n:

Q2.2.2/ Identifier les 4 erreurs qui font que le programme ne fonctionne pas ou il ne donne pas le résultat voulu.

Indice : Chaque erreur comporte une ou deux lignes à rajouter.

Q2.2.3/ Arranger les erreurs dans le fichier *main.s* dans le répertoire *GDB* et assurer vous que le programme fonctionne avec le résultat attendu.

2.3 Anagramme :

Une anagramme est un mot ou une expression obtenue en permutant les lettres d'un mot ou d'une expression de départ. Exemple : une anagramme de chien peut être niche.

Pour détecter qu'un deux mots peuvent être une anagramme on établit l'algorithme suivant :

- Si les deux mots ne sont pas de longueurs équivalentes, il n'y a pas d'anagramme possible.
- Si les deux mots sont de longueurs équivalentes, on calcule pour chaque mot :
    - La somme des lettres divisés par la longueur du mot
- Si la somme des deux mots sont égaux, alors il y a une anagramme possible, sinon il n'en a pas. Il faut afficher la réponse (voir le fichier *main.c* dans le répertoire *anagramme*).

Note : On applique les anagrammes seulement pour les lettres minuscules. Bien que niChE et CHIEN est une anagramme, le code ne doit pas le prendra pas en compte.

Q2.3.1/ Votre devrez compléter le programme *main.s* dans le répertoire *anagramme.*.

Q2.3.2/ Si maintenant on considère les lettres majuscules, donc niChE et CHIEN est une anagramme. Que faut-il ajouter au code pour que le programme fonctionne toujours ? Veuillez répondre dans le code assembleur

2.4 Monnaie :

Vous décidez de prendre un emploi étudiant en tant que caissier au dépanneur le plus proche de chez vous. Le propriétaire vous demande de venir un peu plus tôt afin de pratiquer *la monnaie à rendre* auprès d'un client. Dans votre caisse vous possédez les billets suivants : 25$, 10$, 5$ et 1$. Le principe est donc de remettre auprès d'un client le nombre minimal de billets.

Voici quelques cas à guise d'exemple :

- Un client vient avec 75$: 3 billets de 25$ sont remis
- Un client vient avec 106$: 4 billets de 25$, un billet de 5$ et de 1$ sont remis

Dans le code assembleur la caisse est initialiser de cette manière :

caisse :

.int : 25, 10, 5, 1

Vous ne devez pas modifier cette partie du code. Les billets doivent être récupérés de cette manière.

Vous devez gérer l'affichage des billets à remettre à l'aide de *printf tel* que :

- Le nombre de billets de x$ est : x $

Q2.4/ Votre devrez compléter le programme *main.s* dans le répertoire *monnaie.*

3 Approximation de la valeur de *e* par une série entière

Cette partie traite l'utilisation des instructions arithmétiques de l'assembleur de la famille IA-32 pour approximer le calcul de la valeur de pi

3.1 La formule de la série entière

La définition de e peut être définie par la série suivante :

$$\sum_{n=0}^{+\infty} \frac{1}{n!} = 1 + \frac{1}{1} + \frac{1}{1\ x\ 2\ x\ 3} + \frac{1}{1\ x\ 2\ x\ 3\ x\ 4} + \cdots$$

On pourrait choisir $n = 10$ pour 3.2 et $n = 3$ pour 3.3. Ce qui nous permettra d'obtenir une approximation de la valeur de *e*.

Note : Vous pouvez avoir une étiquette avec la valeur 1 pour le cas $\frac{1}{0!}$ **seulement.**

3.2 Approximation entière de la valeur de *e*

Q3.2.1/ Complétez le programme *e_euler_entiere.s* qui approxime le calcul de la valeur de ***e*** Votre programme doit impérativement utiliser l'instruction `DIV.`

Q3.2.2/ Expliquer ce que vous remarquez après un certain nombre **x** d'itérations. Veuillez répondre dans le code assembleur.

3.3 Approximation flottante de la valeur de *e*:

L'architecture IA-32 implémente un coprocesseur spécialisé pour le calcul scientifique, appelé la FPU (*Floating Point Unit*). La FPU se compose de 8 registres à virgule flottante, chacun de 80 bits, qui sont organisés en pile : c'est-à-dire chaque registre représente un élément de la pile.

La notation `st` réfère à un registre sur la pile, par exemple `st[0]` pointe vers le premier registre en haut de la pile, `st[1]` pointe vers le deuxième registre à partir du haut de la pile, jusqu'à `st[7]` qui pointe vers le dernier registre à partir du haut de la pile. La table 1 illustre ce concept.

Table 3: pile de la FPU

ST[0]
ST[1]
…
ST[7]

`ST[n]` signifie n positions en dessous du sommet de la pile. Faire un push dans la pile du FPU déplace `ST[0]` vers `ST[1]`, `ST[1]`vers `ST[2]`, `ST[2]` vers `ST[3]`, etc.

Toutes les opérations arithmétiques dans la FPU se font entre le haut de la pile, donc `ST[0]`et l'un des registres `ST[i]`, avec i allant de 1 à 7. Ou encore entre `ST[0]` et la mémoire.

Notez bien qu'il n'y a pas de registre à usage général dans la FPU comme `%eax, %ebx`, etc ou d'opérandes immédiats. La FPU consiste uniquement en un ensemble de registres gérés en pile, comme expliqué en dessus.

La FPU étend le jeu d'instruction x86 en ajouter des instructions supplémentaires. Le tableau ci-dessous liste les instructions de la FPU pertinentes pour cette partie.

Instruction	Description
`flds` *adr*	Ajoute au-dessus de la pile l'entier à l'adresse mémoire *addr*. (`st[1]` prend la valeur de `st[0]` et `st[0]` devient la nouvelle valeur chargée de la mémoire.) (c'est équivalent a `Push` utilisée pour la pile principale)
`fstps` *adr*	Retire l'élément `st[0]` pour le mettre en mémoire principale à l'adresse addr. `st[1]` devient `st[0]` (c'est équivalent a `Pop` utilisée pour la pile principale)
`faddp`	`st[0]` est additionné à `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fsubp`	`st[1]` est soustrait à `st[0]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fsubrp`	`st[0]` est soustrait à `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fmulp`	`st[0]` est multiplié avec `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fdivp`	`st[0]` est divisé par `st[1]` et le résultat remplace `st[0]`. `st[1]` est libéré.
`fdivrp`	`st[1]` est divisé par `st[0]` et le résultat remplace `st[0]`. `st[1]` est libéré.

Q3.3.1/ Complétez le programme *e_euler_flottante.s* qui approxime le calcul de la valeur de ***e*** par la série entière en utilisation la FPU.