INF1600

Travail pratique 2

Architecture IA-32 et ASM

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal

TABLE DES MATIERES

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 chaînes 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, de même qu'un rapport en format `.PDF`. Incluez une **page titre** où figurent les noms et matricules des deux membres de l'équipe, votre groupe de laboratoire, le nom et le sigle du cours, la date de remise et le nom de l'École. Dans une seconde page, incluez le **barème** de la section 1.2. Finalement, diverses captures d'écran **pertinentes** doivent figurer au sein de votre rapport. Celles-ci ne peuvent expliquer votre travail d'elles-mêmes ; les captures d'écran servent de support complémentaire. Une justification écrite est de mise.
- **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

Les travaux pratiques 1 à 5 sont notés sur 7 points chacun, pour un total de 35. Le TP2 est noté selon le barème suivant.

2 MANIPULATION DES CHAINES DE CARACTERES

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

2.1 FAMILLE D'INSTRUCTIONS IA-32 POUR LES CHAINES DE CARACTERES

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 utilise les registres `%ESI` et `%EDI` comme pointeurs pour effectuer les opérations. Cela veut dire que ces registres doivent pointer vers les emplacements mémoires des chaînes de caractères à vouloir traiter. **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, soit 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 chaîne 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 chaîne de caractère, par la taille des données spécifiées.

Le tableau 1 ci-bas explique les cinq 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).

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 chaîne en mémoire d'adresse `%ESI`, vers l'emplacement mémoire d'adresse `%EDI`.	`MOVSB`
		`MOVSW`
		`MOVSL`

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

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

Tableau 1. Instructions en assembleur (IA-32) pour le traitement des chaînes de caractères

La mise à zéro ou à un du drapeau DF se fait par le 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`)

Tableau 2. Mise à zéro ou à un du drapeau DF

Une autre instruction souvent utilisée pour manipuler les chaînes 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 chaîne 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 :

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

Tableau 3. Autres instructions de type REP

La manipulation des chaînes 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.

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`

Tableau 4. Codes ASCII utiles pour la partie 2.2 du TP

2.2 ORDRE LEXICOGRAPHIQUE

Vous devez trouver si deux mots sont identiques et ceux mêmes si les lettres sont écrites en majuscule.

Exemple :

1) Arbre et aRBRE = True
2) Niche et Chien = False
3) Banane et Banane = True

Q2.2.1/ 1 pt

Vous devez compléter le programme *comparaison.s* dans le répertoire *Ordre_lexicographique*.

2.3 CHIFFRE DE VERNAM

Le chiffre de Vernam est un algorithme de cryptographie. Cet algorithme se compose d'un message clair et d'une clé qui répond aux trois critères suivants :

- La clé est aussi longue que le texte à chiffrer
- Elle peut être aléatoire
- Elle doit ne chiffrer qu'un seul message

Pour chiffrer le code, on attribue un chiffre allant de 0 à 25 à chaque lettre de l'alphabet (a = 0, z = 25). En agençant les lettres du message à chiffrer aux lettres de la clé on effectue :

- (Message + Masque) % 26
- Si le mot à chiffrer serait HO et que la clé est WL
- Le message chiffrer serait : DZ

Par exemple :

H (7) O (14) message
W (22) L (11) masque

? (29) Z (25) masque + message
D (3) Z (25) (masque + message) % 26

Q2.4.1/ 0,50 pt

Étant donné que nous travaillons avec le code ASCII, il existe un opérande nous permettant d'appliquer cet algorithme (ou une partie). De quel opérande s'agit-il ?

Expliquez deux limitations au fait d'utiliser le chiffre de Vernam comme algorithme de cryptage avec le code ASCII.
Vous pouvez répondre à la question en créant un fichier .txt

3 APPROXIMATION DE LA VALEUR DE PI PAR LA FORMULE DE LEIBNIZ

Cette partie traite de 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 LEIBNIZ

Avec la formule de Leibniz, on peut obtenir le quart de la valeur de pi en effectuant la sommation suivante :

$$\sum_{n=0}^{\infty}(-1)^n a_n \rightarrow a_n \equiv \frac{1}{2n+1}$$

On pourrait choisir $n = 10$ pour 3.2 et $n = 50$ pour 3.3. Ce qui nous permettra d'obtenir une approximation du quart de pi.

3.2 APPROXIMATION ENTIÈRE DE LA VALEUR DE PI

Q3.2.1/ 0.50 pt

Complétez le programme *pi_approx_entiere.s* qui approxime le calcul de la valeur de **pi**. Votre programme doit impérativement utiliser l'instruction `DIV`.

3.3 APPROXIMATION FLOTTANTE DE LA VALEUR DE PI

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.

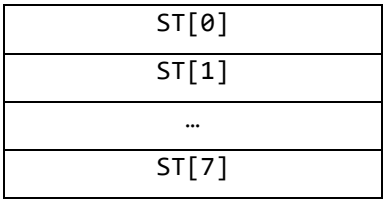

Figure 2. Pile de la FPU

`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 ci-haut.

La FPU étend le jeu d'instructions 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) Cela est équivalent à l'instruction `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]`. Cela est équivalent à l'instruction `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.

Tableau 5. Quelques instructions de la FPU

Q3.3.1/ 0.75 pt

Complétez le programme *pi_approx_flottante.s* qui approxime le calcul de la valeur de **pi** par la formule de Leibniz en utilisation la FPU.

4 APPROXIMATION DE LA RACINE CARRÉE ENTIÈRE

Nous allons utiliser la méthode d'héron

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2},$$

Les variables sont les suivantes :

n : le nombre d'itération (ou à quel itération on est rendu)

$x_0 = a$: La valeur initiale qu'on veut trouver la racine

On veut répéter ce calcul n fois. Les paramètres à chercher sont dans le fichier .c

Q4.2.1/ 0.75 pt

Complétez le programme *racine_carree_entiere.s* qui approxime le calcul de la racine carrée d'un chiffre a positif.

Q4.2.1/ 0.50 pt

Complétez le programme *racine_carree_flottante.s* qui approxime le calcul de la racine carrée d'un chiffre a positif.