INF1600 : TP3

Programmation en assembleur

Imane Hafnaoui <imane.hafnaoui@polymtl.ca>
Giovanni Beltrame <giovanni.beltrame@polymtl.ca>

École Polytechnique de Montréal

Automne 2014

Introduction

Deux travaux pratiques de simulation architecturale complétés, vous êtes fin prêt à attaquer l'écriture d'assembleur IA-32 pour compléter un programme. Dans ce TP, vous allez écrire le code nécessaire à l'optimisation d'un algorithme de calcul numérique à haute performance.

Remise

Voici les détails concernant la remise de ce travail pratique :

- **méthode** : sur Moodle (une seule remise par équipe) ;
- **échéance** : avant 23h50, le 09 novembre 2014 pour les groupes du Lundi B1, le 16 novembre 2014 pour les groupes du Lundi B2.
- **format** : une archive zip contenant les fichiers dot.s, mac.s, mul.s ;
- **langue écrite** : français, English ;
- **distribution** : les deux membres de l'équipe recevront la même note.

Barème

Contenu	***Points du cours***
Fonction dot	1
Fonction mac	1
Fonction mul	3
Lisibilité du code remis (assez de commentaires, pas trop de commentaires)	1
Langage écrit erroné (dans les commentaires)	jusqu'à -0,6
Format de remise erroné (irrespect du noms de fichiers demandé, fichiers superflus, etc.)	jusqu'à -1
Retard	-0,6 par jour

Travail demandé

Vous êtes dans l'équipe d'ingénierie informatique d'une compagnie, PolySim, spécialisée dans les simulations numériques à haute performance. Un algorithme numérique a été développé par votre compagnie mais ses performances ne sont pas suffisantes. Cet algorithme effectue un grand nombre de calculs avec des nombres à virgule flottante simple précision (float). Pour accélérer cet algoritme, on vous demande de ré-écrire trois routines critiques (dot, mac et mul) directement en assembleur de façon à faire les calculs non pas en virgule flottante, mais en virgule fixe et sans perdre en précision!

Virgule fixe

Une analyse fine du problème a permis de montrer que les valeurs numériques manipulées par l'algorithme restent toujours comprises entre -1000 et 1000. Les valeurs numériques peuvent donc être encodées à virgule fixe sur 32 bits avec le format suivant (appelé format 12.20) :

- les 12 bits de poids fort stockent la partie entière
- les 20 bits de poids faible stockent la partie fractionnaire

Voici un court exemple illustrant comment faire la multiplication de deux nombres a=8.5 et b=2.75 avec le format 12.20 proposé:

- a=0x0088 0000 (encodage de a avec le format 12.20)
- b=0x002C 0000 (encodage de b avec le format 12.20)
- a * b = 0x0000 1760 0000 0000 (résultat de la multiplication sur 64 bits)
- a * b = 0x0176 0000 (résultat remis sur 32 bits avec le format 12.20)

Le résultat encodé avec le format 12.20 a pour partie entière 0x17 et pour partie fractionnaire 0x6 0000, il s'agit donc de la valeur 23.375 (en base 10). On retrouve bien le résultat attendu de la multiplication : 8.5 x 2.75 = 23.375. Remarquez que lorsque le résultat est remis sur 32 bits, les 20 bits de poids faible du vrai résultat (sur 64 bits) sont perdus, ce qui introduit une erreur < 1e-6. Dans cet exemple, il n'y a pas d'erreur car les bits de poids faible étaient tous nul, mais ce n'est pas toujours le cas.

En vous inspirant de cet exemple, vous devrez vous même choisir les bonnes instructions assembleur (multiplication, décalage, etc..) pour écrire les trois routines: dot, mac et mul qui vous sont demandées, de façon à faire les calculs en virgule fixe. Ces routines prennent en argument les valeurs numériques déjà encodées dans le format à virgule fixe 12.20. Vous ne devez donc pas faire la conversion float → point fixe vous-même.

Attention le code de référence (en C) qui vous est fourni pour ces routines, utilise des float, mais votre code assembleur ne doit absolument pas utiliser la pile du coprocesseur (vue au TP2) puisqu'on cherche à faire les calculs en virgule fixe!

Fichiers fournis

Les fichiers nécessaires à la réalisation du TP sont dans l'archive `inf1600_tp3.zip` , disponible sur Moodle.

Il y a six fichiers, dont voici les descriptions :

- `Makefile` : le Makefile utilisé pour compilé et nettoyer le projet;
- `tp3.c` : programme de test sans vos fonctions en assembleur;
- `dot.s` : votre fonction dot en assembleur;
- `mac.s` : votre fonction mac en assembleur;
- `mul.s` : votre fonction mul en assembleur;
- `problem.txt` : contient les coefficients d'un problème modèle;

Votre travail devra donc être effectué seulement dans les fichier *.`s`. Il s'agit d'ailleurs des seuls fichiers à remettre dans une archive au format zip.

Vous pouvez compiler votre programme de test en tapant `make`. Vous pouvez ensuite l'exécuter comme ceci :

```
$ ./tp3
```

Le programme de test compare les routines dot, mac, mul de référence (écrite en C) à vos implémentations en assembleur. Pour cela, le programme de test charge en mémoire les coefficients d'un problème modèle, effectue leur conversion en virgule fixe, puis appelle vos routines et les routines de référence. La différence entre vos routines assembleur et l'implémentation de référence est affichée à l'écran. Elle ne devrait jamais dépasser 1e-6.

Astuces pour la réalisation du TP

Cette section rappelle quelques principes de base de l'assembleur x86 32 bits pour vous aider à réaliser ce TP. Soyez attentif à chacun de ces rappeles qui pourrait vous faire sauver beaucoup de temps.

Assembleur: modes d'adressage

Nous rappelons ici les différents modes d'adressage de l'architecture IA-32.

On appelle adresse effective l'adresse calculée par le processeur avant d'accéder à la mémoire. Vous avez vous-même calculé des adresses effectives aux TP1/TP2 lorsque vous calculiez, à partir de registres et de constantes, l'adresse à donner à

la mémoire pour obtenir une donnée. La famille x86 ne fait pas exception.

Le processeur a plusieurs façons de calculer une adresse effective, toutes au bénéfice du programmeur (vous). L'adresse effective la plus simple est une valeur immédiate :

```
mov %eax, 803182
```

Notez l'absence du symbole `$` avant la valeur littérale 803182 ici : ce nombre représente alors une adresse à déréférencer.

Évidemment, ce n'est pas un bon exemple, parce qu'on ne peut pas vraiment savoir (pas facilement, en tout cas) de façon absolue où écrire/lire en mémoire. Pour ce faire, nous utilisons toujours un symbole d'une façon ou d'une autre. Sachez que les symboles des variables globales ou les étiquettes ne sont que des emplacements en mémoire, alors il est tout à fait possible d'écrire

```
mov %eax, g_output_buf
```

À l'inverse,

```
mov g_output_buf, %eax
```

lira un entier de 32 bits (en little-endian) à l'adresse `g_output_buf` et le mettra dans le registre EAX. Par contre,

```
mov $ g_output_buf, %eax
```

ne touchera même pas à la mémoire : il mettra l'adresse `g_output_buf` dans EAX (et non ce qui se trouve en mémoire à cette adresse). Essayez de devenir familier avec cette subtilité.

Un autre mode d'adressage pour utiliser une adresse effective est :

```
mov %eax, (%ecx)
```

qui signifie, en RTN :

```
M[ECX] <- EAX
```

L'adresse est donc dans le registre ECX. Il s'agit ici de la forme la plus simple de ce mode d'adressage. Dans sa forme la plus complexe, on aurait :

```
mov %eax, g_output_buf(%ecx, %edx, 4)
```

qui signifie :

```
M[g_output_buf + ECX + (4 * EDX)] <- EAX
```

Ce mode est très sympathique pour aller chercher le n-ième entier (ici de 32 bits grâce à l'opérande `4`) à une certaine distance d'une adresse immédiate additionnée à un registre. Vous aurez certainement besoin de cette forme pour ce TP. La forme générale peut s'écrire :

```
imm(r1, r2, k)
```

où `imm` est une valeur immédiate, `r1` et `r2` n'importe quels registres et `k` une valeur parmi 1, 2, 4 ou 8.

Dernière note : l'adresse effective peut être calculée sans être utilisée pour accéder à la mémoire. Pour ce faire, utiliser l'instruction `lea` (load effective address) plutôt que `mov` :

```
$ lea g_output_buf(%ecx, %edx, 4), %eax
```

fera

```
EAX <- g_output_buf + ECX + (4 * EDX)
```
(aucun accès mémoire). `lea` peut être pratique pour optimiser certains passages lorsqu'une addition entre une constante et deux registres est nécessaire (plutôt que d'écrire deux fois `add`).

Assembleur : registres intouchables

Dans la convention d'appel du langage C pour IA-32, il est formellement interdit de modifier les valeurs des registres sauf pour EAX, ECX et EDX. Si vous voulez modifier EDI ou EBX, par exemple, vous devez vous assurer qu'ils auront leurs valeurs initiales au retour de la fonction.

Assembleur : grandeurs des copies

Lorsque vous copiez avec l'instruction `mov` sans qu'il y ait un registre impliqué (par exemple, une copie d'une valeur immédiate vers la mémoire), l'assembleur n'a aucune façon de déterminer la grandeur de votre copie. Les copies supportées par le processeur sont, à tout le moins, 1 octet, 2 octets ou 4 octets. Si vous écrivez cette instruction :

```
mov $ 0x25, (%eax)
```
et supposons que EAX contienne la valeur 40, trois situations sont possibles :

- placer l'octet 0x25 à l'adresse 40;
- placer les octets 0x25, 0x00 (little-endian) aux adresses 40 et 41 ou
- placer les octets 0x25, 0x00, 0x00, 0x00 (little-endian) aux adresses 40, 41, 42 et 43.

Dans le doute, l'assembleur émet une erreur et refuse de produire le code machine. C'est pourquoi, dans cette situation ambigüe, vous devez toujours spécifier la taille de la destination avec un des suffixes `b`, `w` ou `l`, respectivement pour byte (1 octet), word (2 octets) et long (4 octets). Les trois instructions suivantes sont donc valides :

```
movb $ 0x25, (%eax)
movw $ 0x25, (%eax)
movl $ 0x25, (%eax)
```
Lorsqu'il y a un registre impliqué (pas dans le calcul de l'adresse, mais en tant que donnée), le suffixe n'est pas nécessaire parce que la taille du transfert est connue par la taille du registre. Par exemple :

```
mov %ecx, (%eax)    // copie de 32 bits
mov %cx, (%eax)     // copie de 16 bits
mov %cl, (%eax)     // copie de 8 bits
mov %ch, (%eax)     // copie de 8 bits
```

Assembleur : convention d'entrée/sortie de fonction

Vous demandez-vous à quoi servent les lignes

```
push %ebp
mov %esp, %ebp
```

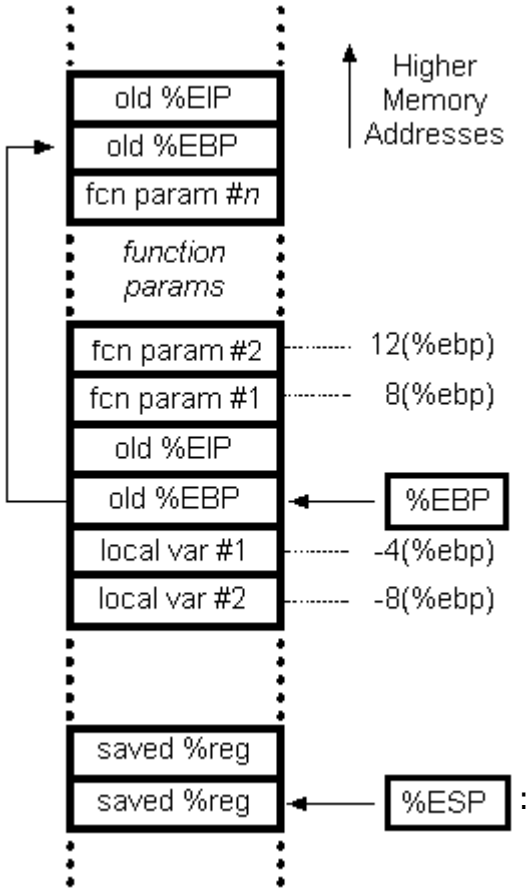
et

```
leave
```

dans le code fourni ? Il s'agit de conventions d'entrée et de sortie de fonction. En principe, toutes les fonctions contiennent ces instructions pour assurer un bon maintien de la pile et faciliter la tâche au programmeur.

Observez attentivement les deux premières lignes. La valeur actuelle du registre EBP est conservée sur la pile, puis on la remplace par le pointeur de pile actuel. Par la suite, dans notre fonction, EBP n'est pas supposé être modifié. Ceci vous assure alors que EBP pointe toujours où ESP pointait en début de fonction.

Cette assurance vous permet de vous baser sur EBP de façon absolue pour aller chercher des paramètres de fonction et des variables locales. Pour reprendre le très bon [schéma de Steve Friedl]



On voit ici l'état de la pile après la convention d'entrée. Ici, on a aussi effectué des réserves de variables locales et des sauvegardes de registres, d'où l'espacement entre ESP et EBP, mais notez malgré tout que EBP pointe toujours sur son ancienne valeur. Le premier paramètre de fonction est donc toujours accessible avec `8(%ebp)`, le second avec `12(%ebp)`, etc., puisque ceux-ci ont été poussés à l'envers.

La fin de routine est particulièrement satisfaisante. On veut évidemment que ESP reprenne sa valeur initiale, sans quoi le retour (avec l'instruction `ret`) produirait une exception parce qu'il y aurait un saut fait à une adresse probablement invalide. Au lieu de retenir le nombre de `push` effectué et de le synchroniser avec un nombre égal de `pop`, on peut profiter du fait que l'ancienne valeur de ESP est la valeur en cours de EBP. Donc on fait l'opération inverse :

```
mov %ebp, %esp
pop %ebp
```

Ces deux instructions sont tellement souvent exécutées qu'une instruction spécifique été créée qui produit exactement le même comportement : `leave`.

Assembleur : printf

Il vous est tout à fait possible d'afficher du texte (vers la console) pendant l'exécution de votre code si vous voulez. Vous devez alors appeler `printf` manuellement. Si vous ne connaissez pas cette fonction et ce qu'elle offre, consultez cette ressource.

Qu'avons-nous besoin pour appeler `printf` ? Son premier paramètre est une chaine C qui mentionne le format d'impression. Puis, les paramètres suivants sont les variables qu'on veut visualiser. Voici un exemple qui affiche une adresse contenue dans EAX, puis l'entier signé par cette adresse :

```
.data
chaine0:
.asciz “EAX=%p, M[EAX]=%d\n”
.text
pusha
pushl (%eax)
push %eax
pushl $ chaine0
call printf
add $ 12, %esp
popa
```

Consultez vos notes de cours pour savoir pourquoi les arguments sont poussés à l'envers sur la pile. La barrière `pusha`/`popa` sert à éviter de perdre les valeurs des registres modifiables par d'autres fonctions (EAX, ECX et EDX).

Débogage

Vous pouvez déboguer votre programme avec gdb. Vous pouvez aller voir votre source avec gdb et insérer des points d'arrêt dans le code assembleur.

Si vous avez une erreur de segmentation (ce qui vous arrivera fort probablement), gdb vous indiquera à quelle ligne se produit celle-ci et vous pourrez alors observer le contexte (valeurs des registres, des variables, de la mémoire, de la pile) et déterminer plus facilement la cause de cette erreur. Une erreur de segmentation arrive toujours lorsque le programme tente d'accéder à un emplacement mémoire invalide.