# INF1600 — TP4
# Programmation en assembleur et C++

Giovanni Beltrame giovanni.beltrame@polymtl.ca
Luca G. Gianoli luca-giovanni.gianoli@polymtl.ca

**Polytechnique Montréal**
**Année académique 2016/2017**

# Remise

Voici les détails concernant la remise de ce travail pratique :

— **Méthode :** sur Moodle (une seule remise par groupe).

— **Échéance :** avant 23h55 ; le 20 novembre 2016 pour le groupe 1, le 27 novembre 2016 pour le groupe 2 ;

— **Format :** un seul fichier zip, dont le nom sera `<matricule1>-<matricule2>.zip`. Exemple : `0123456-9876543.zip`. L'archive doit contenir les fichiers `update_status.s`, `average_speed.s`, `total_distance.s` et `equivalent_acc.s`.

— **Langue écrite :** français.

— **Distribution :** les deux membres de l'équipe recevront la même note.

# Barème

| **Contenu** | **Points du cours** |
|---|---|
| `update_status.s` | 2 |
| `average_speed.s` | 1 |
| `total_distance.s` | 1 |
| `equivalent_acc.s` | 2 |
| Illisibilité du code (peu de commentaires, mauvaise structure...) | jusqu'à -1 |
| Format de remise erroné (irrespect des noms de fichiers demandés, fichiers superflus, etc.) | jusqu'à -1 |
| Retard | -0,025 par heure |

# Travail demandé

Vous êtes en charge de coder en assembleur quatre méthodes :

— `CCar::UpdateStatusAsm()`

— `CCar::AverageSpeedAsm()`

— `CCar::TotalDistAsm()`

— `CCar::EquivalentAccAsm()`

La réalization des ces méthodes doit suivre les exemples donnés dans les méthodes corrispondantes en C++ :

— `CCar::UpdateStatusCpp()`

— `CCar::AverageSpeedCpp()`

— `CCar::TotalDistCpp()`

— `CCar::EquivalentAccCpp()`

**Les opérations du code c++ doivent être exactement répliquées dans le code en langage assembleur**.

## Fichiers fournis

Les fichiers nécessaires à la réalisation du TP sont dans l'archive `inf1600_tp4.zip`, disponible sur Moodle.

Voici la description des fichiers :

— `Makefile` : le *makefile* utilisé pour compiler et nettoyer le projet ;

— `tp4.c` : programme de test qui utilise le fonctions de référence et celles en assembleur ;

— `car.h` : la définition de la classe `CCar` ;

— `car.cpp` : l'implementation C++ de la classe `CCar` ;

— `car.vtable` : la structure de la *virtual table* de la classe `CCar` ;

Vous devez compléter les fichiers `*.s` et les remettre dans un archive zip. Votre code doit passer chaque tests dans `tp4.c`.

## Compilation et testing

Pour compiler le programme de test (`tp4`), il est suffisant de taper :

```
$ make
```

Pour l'éxécuter, vous devez passer 4 arguments. Le premier est la position initiale, la deuxième est la vitesse initiale et la troisième est l'accélération initiale. Enfin, le quatrième est l'intervalle d'actualisation. Par exemple, dans la commande :

```
$ ./tp4 0 0 1 1
```

la voiture va commencer la simulation en position 0, avec un vitesse de 0 $m/s$ e une accélération de 1 $m/s^2$. Chaque fois que le les méthodes `CCar::UpdateStatusCpp()` et `CCar::UpdateStatusAsm()` sont appelées, position et vitesse sont mises à jour en fonction de l'accélération (1 $m/s^2$ constante pendant toute la simulation) et de l'intervalle de d'actualisation (1 $s$).

Accélération et intervalle de d'actualisation doivent être supérieurs ou égaux à 1. Si les valeurs ne le respectent pas, le programme sorte un erreur.

# État de la pile au début d'une méthode

Lorsqu'une méthode `M` d'une classe `C` est appelée, le premier argument qui se trouve dans la pile (`8(%ebp)`) est toujours l'adresse de l'objet de la classe `C`.

Par exemple, avec cette définition :

```
class C {
  public:
    void M(int x);
  private:
    int i;
};
```

quand la méthode `M` est appelée, la pile est :

| | |
|---|---|
| `12(%ebp)` | valeur de `x` |
| `8(%ebp)` | addresse de l'objet de type `C` |
| `4(%ebp)` | ancienne valeur de `%eip` |
| `(%ebp)` | ancienne valeur de `%ebp` |

## Opérations avec valeurs `float`

Pour cet exercice, on se sert de la partie FPU (unité à virgule flottante) du processeur Intel. Il s'agit d'une pile dédiée au calcul flottant (différente de la pile d'appel), de grandeur 8 (elle peut contenir jusqu'à 8 entrées de type `float`), mais il est rarement nécessaire de dépasser 2 ou 3 entrées. Les quelques instructions agissent toujours sur le premier et le deuxième éléments de la pile (`st[0]` et `st[1]`). Voici ces instructions :

| **Instruction** | **Rôle** |
|---|---|
| `fld x` | Ajoute au dessus de la pile la valeur à l'adresse mémoire `x` ; `st[1]` prend la valeur de `st[0]` et `st[0]` devient cette nouvelle valeur chargée de la mémoire. |
| `fldpi x` | Ajoute au dessus de la pile la valeur de $\pi$ (3,1415...). |
| `fstp x` | Retire l'élément `st[0]` pour le mettre en mémoire principale à l'adresse `x`. `st[1]` devient `st[0]`. |
| `faddp` | `st[0]` est additioné à `st[1]` et le resultat remplace ces deux éléments. |
| `fsubp` | `st[1]` est soustrait de `st[0]` et le resultat remplace ces deux éléments. |
| `fsubrp` | `st[0]` est soustrait de `st[1]` et le resultat remplace ces deux éléments. |
| `fmulp` | `st[0]` est multiplié avec `st[1]` et le resultat remplace ces deux éléments. |
| `fdivp` | `st[0]` est divisé par `st[1]` et le resultat remplace ces deux éléments. |
| `fdivrp` | `st[1]` est divisé par `st[0]` et le resultat remplace ces deux éléments. |

Pour retourner un `float` en sortant d'une fonction, laissez la valeur sur `st[0]`.