Programmation assembleur (ASM)

Prof. Daniel Rossier
Assistants : Bruno Da Rocha Carvalho, Thomas Rieder

Macros et instructions de transfert x86

lab05 (26.04.2022)

Objectifs de laboratoire

L'objectif de ce laboratoire est de se familiariser avec les instructions de transfert x86 grâce à l'écriture de fonctions en assembleur x86 et à l'utilisation de macro-instructions. Ce laboratoire permettra aussi de se familiariser avec les pseudo-instructions ainsi que les différents modes d'adressage ARM et x86.

Échéance

- Le laboratoire doit être rendu avant le début du prochain laboratoire (**durée : 2 semaines**)

Validation du laboratoire

Un script de rendu vous est fourni afin de packager les fichiers sources modifiés sans inclure tous les fichiers générés (code objet, exécutables, etc.). Ce script s'appelle ***rendu.sh*** et sera fourni pour tous les laboratoires.

```
reds@reds2022:~/asm/asm_lab0X$ ./rendu.sh
```

Cette commande va générer un fichier *asm_lab0X_rendu.tar.gz* (où X est le numéro du labo) et c'est cette archive qui sera à remettre sur Cyberlearn. Il est demandé de toujours préparer l'archive du laboratoire grâce au script de rendu fourni.

Étape 1 - Récupération des sources & environnement *Eclipse*

```
$ retrieve_lab asm lab05
```

Étape 2 – Macros ARM

Dans le laboratoire précédent nous avons implémenté des fonctions de distance. Pour la distance de Manhattan par exemple il fallait calculer la valeur absolue de la différence entre deux coordonnées.

Afin d'éviter la duplication de code une option est d'utiliser des fonctions, une autre possibilité est d'utiliser une macro. De même, une macro était utilisée dans le code C.

En l'occurrence dans le fichier « *path_finding.c* » on retrouve :

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
#define MIN(X, Y) ((X) < (Y) ? (X) : (Y))
#define ABSDIFF(X, Y) (MAX(X,Y) - MIN(X,Y))
```

a) Ajouter une macro « absdiff » qui prend en paramètre trois registres et compléter les fonctions « manhattan_distance_asm() » et « ssd_semi_distance_asm() » en utilisant cette macro dans le fichier « arm/path/student_functions_asm.S ». Les trois registres passés en paramètres sont les deux registres pour les valeurs d'entrée, X, Y dans le code C ci-dessus, et le troisième registre sera pour le résultat. Pour rappel, les algorithmes de distances se trouvent dans la donnée du *lab04*.

a) Ajouter une macro « unpack » qui permet de « dépacker » les valeurs X, Y. Le premier registre passé en paramètre sera celui qui contient à la fois X et Y et les deux autres registres la destination pour X, Y respectivement.
b) Tester le bon fonctionnement de vos fonctions (programmes ARM « ***print*** » et « ***path*** »).

Étape 3 – Détection de contours (traitement d'image en x86)

Cette étape consiste à implémenter un algorithme de détection de contours sur une image. Il existe plusieurs algorithmes de détection de contours mais l'idée générale est la suivante ; S'il y a une variation de couleur brusque autour d'un point donné il y a un contour.

Si l'on imagine une image de couleur unie, pour chaque point il n'y a aucune variation par rapport à ses voisins, donc il n'y a pas de contours. A l'inverse, si l'on considère une image avec un cercle d'épaisseur d'un pixel noir sur fond blanc, pour chaque point du cercle les voisins seront majoritairement des points blancs, donc il y a une grande variation de couleurs entre les pixels du cercle et ses voisins : il s'agit bien d'un contour.

Un autre exemple consiste à considérer un rond de couleur sur fond uni ; à l'extérieur du rond, ainsi qu'à l'intérieur, il n'y a pas de variations, mais sur le bord la variation de couleurs signifie la présence d'un contour.

Pour simplifier l'algorithme on ne fera pas le calcul de la variation de la couleur mais celui de la luminosité. Car la variation de la couleur demande une analyse sur trois dimensions : rouge-vert-bleu. La variation de la luminosité permet de travailler dans une seule dimension, la luminosité, de clair (blanc) à foncé (noir), en niveau de gris.

Une méthode pour implémenter la détection de contours est d'effectuer une convolution (voir ci-dessous) avec une matrice appelée ***noyau de traitement d'image pour détection de contours*** qui permettra de quantifier la variation de luminosité autour d'un point. Le noyau de traitement d'image utilisé est le suivant :

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Convolution

La convolution entre deux matrices de mêmes dimensions, bien que souvent notée avec le symbole * n'est pas une multiplication traditionnelle entre deux matrices, mais utilise le procédé suivant :

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}\right)[2,2] = (i*1)+(h*2)+(g*3)+(f*4)+(e*5)+(d*6)+(c*7)+(b*8)+(a*9)$$

Le procédé est ici appliqué au point [2, 2], représenté par la valeur e au centre de la matrice les autres lettres étant les pixels voisins. La seconde matrice (1, 2 …, 9) représente le noyau de traitement d'image, ici avec des valeurs de 1 à 9 afin de bien différencier l'ordre des opérations.

Exemples

1) Si on utilise le noyau de détection de contours sur un fond noir (que des "0") on aura le résultat 0 pas de variation et donc pas de contours.

$$\left(\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}\right)$$

2) Le même noyau de détection de contours est maintenant utilisé sur un point qui se trouve sur le bord d'un rectangle gris (64) le résultat est -192 il y a donc une variation de luminosité quantifiée de -192.

$$\left(\begin{bmatrix} 0 & 64 & 64 \\ 0 & 64 & 64 \\ 0 & 64 & 64 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}\right)$$

On peut voir que le résultat est signé, comme **on ne s'intéresse pas au signe** mais seulement à l'amplitude de la variation on prendra donc la valeur 192.

3) Ici avec un contour en diagonale de noir (0) vers un gris moyen (128)

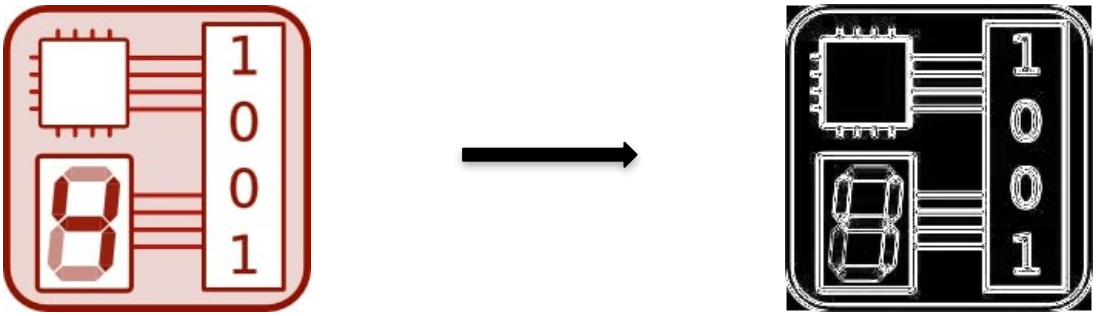
$$\left(\begin{bmatrix} 0 & 0 & 128 \\ 0 & 128 & 128 \\ 128 & 128 & 128 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}\right)$$

La valeur de la convolution est -384, et l'on prendra 384 **qu'on limitera à 255 (blanc)** pour quantifier le contour.

Avant toute chose, il est recommandé **de tester la méthode sur papier** avec différentes valeurs dans la matrice de gauche et faire la convolution avec le noyau de détection de contours afin de se familiariser avec la convolution et de voir les résultats ainsi obtenus dans les différents cas de figure.

Détection de contours

L'idée est d'appliquer cette opération pour chaque pixel de l'image et de créer une image avec les résultats de la convolution de chaque pixel de l'image de départ avec le noyau de détection de contours, comme le montre la figure ci-dessous



Voici un pseudo-code pour effectuer ce traitement :

```
POUR chaque ligne de l'image FAIRE
        POUR chaque pixel dans la ligne FAIRE
                Accumulateur := 0

                POUR chaque ligne du noyau FAIRE
                        POUR chaque élément de la ligne du noyau FAIRE
                                Multiplier la valeur de l'élément avec la valeur du pixel correspondant
                                Additionner le résultat à l'accumulateur
                        FIN POUR
                FIN POUR

                SI l'accumulateur est négatif FAIRE
                        Accumulateur = -Accumulateur (valeur absolue)
                FIN SI
                SI l'accumulateur est > luminosité max FAIRE
                        Accumulateur = luminosité max (255)
                FIN SI
                Affecter la valeur de l'accumulateur au pixel de l'image de destination
        FIN POUR
FIN POUR
```

Attention ! Sur les bords de l'image il n'est pas possible d'effectuer la convolution car les voisins du pixel hors de l'image ne sont pas définis, on vous demandera de simplement copier le pixel si ses voisins sont hors de l'image.

Les images de départ et de destination sont en niveaux de gris de 0 (noir) 255 (blanc) encodées sur un octet par pixel. L'adresse de départ de l'image pointe sur le pixel [0,0] et l'octet suivant est le prochain pixel sur la ligne et ainsi de suite. Après autant d'octets que la largeur de l'image le pixel suivant sera le premier pixel de la prochaine ligne donc [1,1].

Le kernel de convolution à utiliser se trouve dans le fichier kernels.c et s'appelle edge_detection_3x3

Le programme de traitement d'image fourni s'appelle « ***image_processing*** » dans le dossier « ***x86/image_processing/*** ». Il peut être généré à l'aide du *Makefile* fourni. Ce programme sera compilé et exécuté sur le machine-hôte (*x86_64*) et **non** dans QEMU. Cependant il sera compilé en *x86_32* afin de générer de l'assembleur 32-bits comme vu en cours.

a) L'application « *image_processing* » nécessite l'installation du paquet « *gcc-multilib* » (pour les librairies 32-bits) avec la commande suivante :

```
$ sudo apt install gcc-multilib
```

b) Produire l'exécutable en lançant la commande ***make*** dans le dossier « ***x86/image_processing/*** »
c) Lancer l'application comme ci-dessous :

```
$ ./image_processing
```

Le programme lit une image fournie en niveaux de gris (par défaut le fichier « ***cpu.png*** ») et effectue le traitement d'image, c'est-à-dire la réduction de bruit. Deux fichiers seront générés : « ***expected_result.png*** » et « ***student_result.png*** ».

Le premier fichier « ***expected_result.png*** » est le résultat attendu de l'algorithme, écrit en C dans le fichier « ***image_processing.c*** ».

Le second fichier « ***student_result.png*** » sera le résultat obtenu grâce au code assembleur. L'image sera toute noire au début car la partie assembleur ne fait encore rien pour le moment, le fichier image ne contenant que des "0".

Le programme affiche l'image de départ, le résultat de l'algorithme écrit en assembleur, et si le résultat diffère du résultat attendu, une image **avec des pixels rouges** pour chaque pixel différent ; cette image est stockée dans « ***diff.png*** ».

Il est possible de spécifier un fichier image différent avec l'option –f comme suit :

```
$ ./image_processing –f  /home/reds/mon_image.jpg
```

⇒ Cette option sera très utile afin de tester l'implémentation avec de petites images. Travailler avec une image de 5x5 pixels sera plus simple pour *debugger* et trouver des erreurs. Des images plus petites que « ***cpu.png*** » sont fournies.

Il est possible aussi de demander d'afficher les coordonnées des erreurs ainsi que leurs valeurs avec l'option –s, comme suit :

```
$ ./image_processing -s
```

Avec cette option le programme affichera une ligne sur la sortie standard (*stdout*) pour chaque pixel qui diffère de la solution attendue. Les deux options peuvent être utilisées simultanément. Ceci permet de poser des breakpoints conditionnels (x = … y = …).

d) Éditer et compléter le fichier assembleur « *asm_filter.S* ».

⇒ Les sources du programme « *image_processing* » sont fournies et peuvent aider à la compréhension. Ne pas hésiter à les modifier au besoin (pour ajouter des « *printf()* » par ex.). La solution dans le fichier « *asm_filter.S* » devra toutefois fonctionner avec le programme C d'origine.

⇒ Le fichier assembleur est commenté et pour chaque tâche à effectuer il y a un commentaire "TODO : … ". Une partie du pseudo code vert est déjà présente.

⇒ Dans le fichier assembleur, des commentaires proposent des suggestions.

⇒ La fonction assembleur est appelée par le programme C ; il ne faut en aucun cas changer sa signature ou l'ordre des arguments dans le code assembleur.

⇒ Le noyau de convolution défini dans « *kernels.c* » est accessible depuis le code assembleur