Programmation assembleur (ASM)

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

Inline Assembly, *Compiler Explorer* et instructions ARM

lab04 (05.04.2023)

Objectifs de laboratoire

L'objectif de ce laboratoire est d'apprendre à utiliser l'assembleur « *inline* » dans un programme C, de découvrir « *Compiler Explorer* » et de se familiariser avec les instructions ARM grâce à l'écriture de fonctions en assembleur ARM.

Echéance

- 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*

```
reds@reds2022:~$ retrieve_lab asm lab04
```

Étape 2a – Assembleur « *inline* » en C avec GCC

Il est parfois nécessaire d'utiliser des instructions assembleur spécifiques pour certaines tâches. Une solution consiste à écrire une fonction en assembleur et d'appeler cette fonction depuis un code C comme dans les laboratoires précédents. Une autre possibilité est d'intégrer directement le code assembleur dans le code C. C'est ce qui sera exploré dans cette première étape.

Afin d'avoir un exemple concret, un code C qui essaie de déterminer le type de processeur utilisé est étudié.

La documentation de GCC nous explique comment utiliser de l'assembleur « *inline* » dans un code C.
https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html

Sous sa forme la plus simple l'assembleur « *inline* » s'écrit

```
asm("instructions");
```

Le standard du langage C ne spécifie pas la sémantique pour le mot clé « ***asm*** », selon le compilateur utilisé la sémantique peut être différente, il faut donc se référer au manuel du compilateur utilisé.

D'ailleurs pour compiler avec « *-ansi* » ou « *-std* » il faut utiliser « *__asm__* » car « *asm* » est une extension spécifique GNU.

```
__asm__("instructions");
```

Par exemple :

```
__asm__("nop"); // Do nothing
__asm__("int $3"); // Software interrupt with value 3
```

⇒ De manière générale utiliser de l'assembleur dans du code C n'est pas quelque chose de portable, car on dépend de l'architecture du processeur utilisé, ainsi que du compilateur. Une bonne utilisation du préprocesseur avec des « *#if* » permet de rendre le code plus portable si nécessaire.

Si l'on utilise plusieurs instructions, il faut les séparer avec « ***\n\t*** »

```
__asm__("nop\n\t"
        "int $3");
```

Sous sa forme étendue l'assembleur « *inline* » peut prendre des entrées et sorties (opérandes) depuis le code C. Alors que sous la forme basique on peut seulement appeler des instructions assembleur. La forme étendue est donc plus pratique car permet de faire plus de choses. Elle se présente ainsi :

```
__asm__ [ asm-qualifiers ] ( AssemblerTemplate
                           : OutputOperands
                         [ : InputOperands
                         [ : Clobbers ] ])
```

Les crochets ici représentent des arguments optionnels.

La documentation complète est disponible ici :
https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm

Dans le jeu d'instruction x86, l'instruction « ***cpuid***[1] », qui prend comme argument la valeur dans « eax » et retourne un résultat dans les registres « eax » à « edx », permet d'identifier le processeur. Il n'y a pas d'instruction C qui permet directement d'appeler l'instruction « *cpuid* », il faut donc passer par de l'assembleur. On peut trouver un exemple de ceci dans le noyau Linux[2].

[1] https://en.wikipedia.org/wiki/CPUID
[2] https://elixir.bootlin.com/linux/v5.10-rc1/source/arch/x86/include/asm/processor.h#L211

Dans le jeu d'instructions ARM, il n'y a pas directement d'instruction qui donne l'identifiant du processeur. Par contre le coprocesseur P15 « *system control coprocessor* » a un registre qui contient cette information. Dans le langage C il n'y a pas d'instruction pour aller lire le contenu d'un registre d'un coprocesseur. Il faut donc passer par de l'assembleur.

Dans l'assembleur ARM il y a l'instruction « **MRC** » « ***M**ove to ARM **R**egister from **C**oprocessor* » qui permet de copier le contenu d'un registre d'un coprocesseur vers un registre du processeur principal. Étant donné que l'identifiant du CPU ARM se trouve dans le registre 0 du coprocesseur P15 il est possible de récupérer cet identifiant. On peut trouver un exemple dans le code du noyau Linux[3]. V

a) Examiner le code source du noyau Linux pour les deux exemples ci-dessus (suivre les liens en bas de page 2 et 3).
b) Compiler l'application C qui se trouve dans le dossier « ***arm/cpuid*** » et « ***x86/cpuid*** » avec la commande « *make* ». Ne pas hésiter à regarder son code source.
c) Exécuter « **arm/cpuid/main.bin** » dans l'émulateur QEMU ARM et « **x86/cpuid/main.bin** » dans l'émulateur QEMU x86.

⇒ Quelles sont les différences ? Le CPU a-t-il bien été identifié ?

Étape 2b – Assembleur « *inline* » en C avec GCC

En assembleur il y a parfois des instructions particulières qui n'ont pas d'équivalent direct en C. Ces instructions peuvent nous servir. Un exemple de ceci est l'instruction x86 « *lzcnt* » « Leading zeros count ». Cette instruction permet de retourner le nombre de bits à zéro dans un mot 32-bits à gauche du premier bit à un rencontré.

d) Implémentez la fonction « ***leading_zero_count()*** » qui compte le nombre de bits à 0 dans un mot de 32-bits à gauche du premier bit à 1. Premièrement en C puis grâce à de l'assembleur « *inline* » dans le fichier « ***x86/lzcnt/main.c*** ». Utilisez l'instruction x86 « ***lzcnt src, dst*** ».

⇒ Utiliser la forme étendue de l'assembleur inline « __asm__(…); ».

e) Vérifier le fonctionnement des deux versions.

⇒ Certains compilateurs proposent directement des « *intrinsic* » afin de faciliter l'utilisation de ces instructions depuis un code C. Par exemple pour « *lzcnt* » et GCC :

```
unsigned int __builtin_ia32_lzcnt_u32 (unsigned int);
```

- Plus d'informations : https://gcc.gnu.org/onlinedocs/gcc/x86-Built-in-Functions.html
- Pour les processeurs Intel, Intel propose un guide des « *intrinsics* » ici : https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Cela permet d'utiliser les instructions étendues des processeurs Intel comme des fonctions C grâce à un « *include* », par exemple :

[3] https://elixir.bootlin.com/linux/latest/source/arch/arm/include/asm/cputype.h#L132

```
#include <immintrin.h>
// Pour unsigned int _lzcnt_u32(unsigned int a); // Qui fait pareil que ci-dessus
```

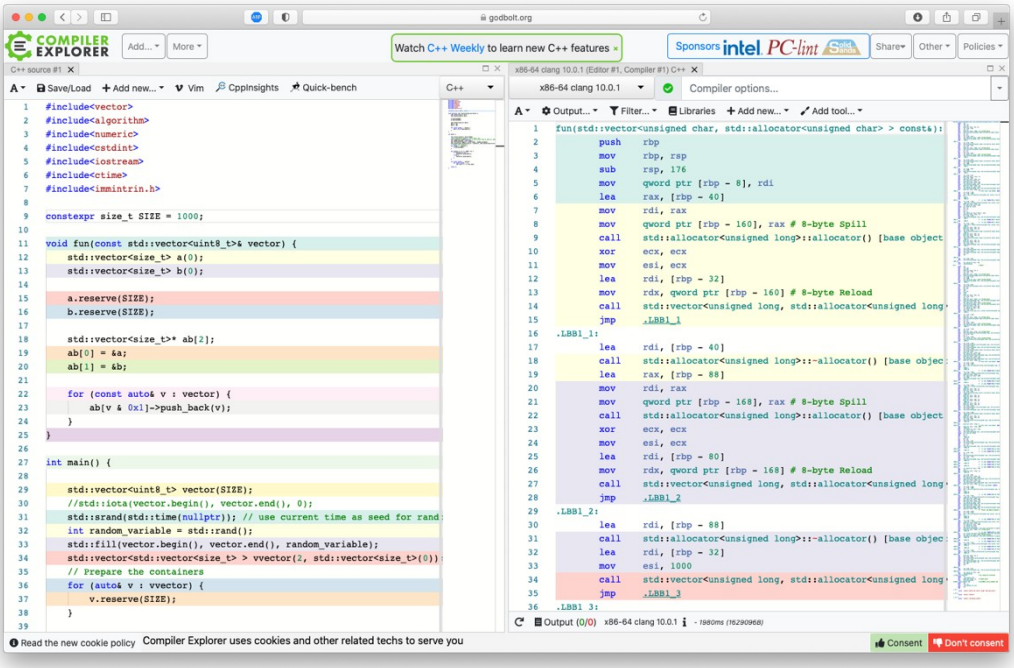
f) Chercher l'instruction « leading zeros count » sur le site d'Intel et lisez sa description, en particulier la partie « Operation » qui décrit le pseudo-code de l'instruction. Ceci peut s'avérer utile pour d'autres instructions, chercher aussi l'instruction « popcnt » et lisez la description.

Étape 3 – Compiler Explorer

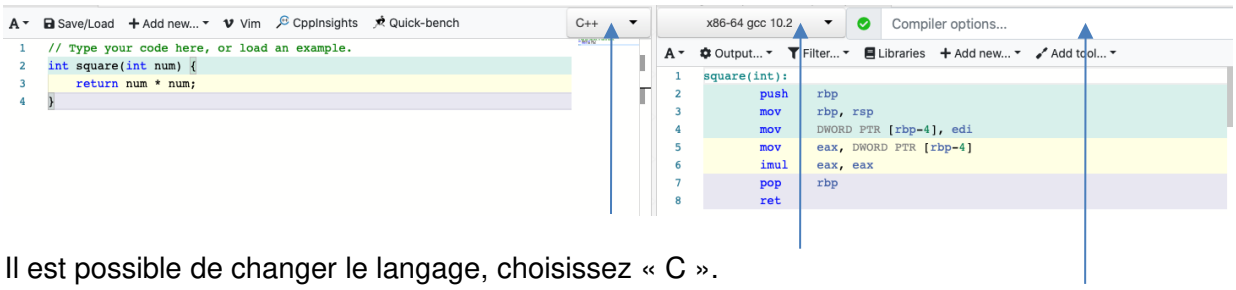
Dans les laboratoires précédents nous avons exploré l'assembleur résultant de compilation de code C. Ceci a été réalisé à l'aide de l'outil « *objdump* » de la *toolchain* correspondante à la cible d'intérêt.

Il existe un formidable outil on-line appelé « *Compiler Explorer* » créé par Matt Godbolt[4] qui permet d'écrire du code C/C++ et d'avoir directement le code assembleur résultant dans la même page web.

L'outil est disponible à l'adresse suivante : https://godbolt.org



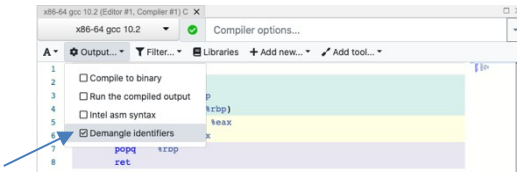
Le code peut être rentré dans la fenêtre de gauche, ici un exemple de code C++ et dans la fenêtre de droite on peut voir l'assembleur généré, celui-ci étant coloré afin d'avoir une correspondance visuelle avec le code.
Par défaut le site propose un exemple tout simple, une fonction qui met une valeur au carré. Le code assembleur résultant comporte trois sections (trois couleurs) l'entrée dans la fonction, le code effectif et la sortie de la fonction. On peut donc voir en jaune la mise au carré avec l'opération « *imul* ».



⇒ Il est possible de changer le langage, choisissez « C ».
⇒ Il est aussi possible de changer le compilateur, par exemple « ARM gcc 6.4 ».
⇒ Il est possible de passer des options supplémentaires au compilateur, par exemple « -03 ».

[4] https://xania.org/MattGodbolt

Avant de continuer désactiver la syntaxe « Intel » afin d'avoir la même syntaxe que vue en cours « AT&T ».



a) Copier le code de « ***compiler/gpio_c.c*** » dans le « *Compiler Explorer* » et observer le résultat.

⇒ Utiliser l'option « ***-m32*** » afin d'avoir du x86 32-bits et non du x86 64-bits.

b) Retirer le mot clé « *volatile* » de l'argument et observer le changement.

c) Changer le compilateur de x86 à ARM (retirer l'option « *-m32* »). Passer les différentes options d'optimisation de « *-O0* » (aucune optimisation) à « *-O3* » et observer les changements.

d) Copier le code de « ***compiler/dot_prod.c*** » dans le « *Compiler Explorer* » et observer l'assembleur résultant en x86 et ARM (sans options particulières sauf « *-m32* » pour x86).

e) Copier le code de « ***compiler/asm.c*** » dans le « Compiler Explorer » et observer l'assembleur résultant.

⇒ On peut voir ici comment le compilateur interprète l'instruction « __asm__(); » il s'agit d'un remplacement textuel des instructions, on remarque aussi que le compilateur pourrait donc écrire des instructions x86 dans de l'ARM et vice-versa. Ce sera l'étape d'assemblage qui va retourner une erreur dans ces cas. On voit aussi pourquoi il faut ajouter le « *\n\t* » après les instructions.

f) Copier le code de « ***compiler/cpuid.c*** » (x86) dans le « *Compiler Explorer* » et changer les paramètres des entrées/sorties (*inputs/outputs)*, par exemple en changeant le "*+m*" en "*+r*" et observer les différences. Retirer la liste des « *Clobbered registers* » et observer la différence.

⇒ Se référer au besoin au manuel de GCC :

- https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm
- https://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html#Simple-Constraints

Étape 4 – ARM Path Finding – Fonctions de distance

Dans le cadre du développement d'un algorithme de « path finding » pour un jeu vidéo nous avons besoin d'une fonction de distance afin d'évaluer la distance d'une entité (par exemple le joueur) d'un but (par exemple un trésor).

Dans cette étape il s'agira d'implémenter des fonctions de distances différentes afin d'évaluer leurs performances pour le « path finding ».

Une fonction de distance est une fonction qui prend en entrée deux valeurs d'un ensemble E et y assigne une valeur réelle positive.

$$d : E \times E \rightarrow R^{+¿¿}$$

Qui vérifie les propriétés suivantes :

- Symétrie $\forall (a,b) \in E^2, d(a,b) = d(b,a)$
  « La distance de a à b doit être égale à la distance de b à a »

- Séparation $\forall (a,b) \in E^2, d(a,b) = 0 \Longleftrightarrow a = b$
  « La distance entre a et b est 0 si et seulement si a est égal à b »

- Inégalité triangulaire $\forall (a,b,c) \in E^2, d(a,c) \leq d(a,b) + d(b,c)$
  « La distance entre a et c est plus petite ou égale que la distance entre a et b plus la distance entre b et c »

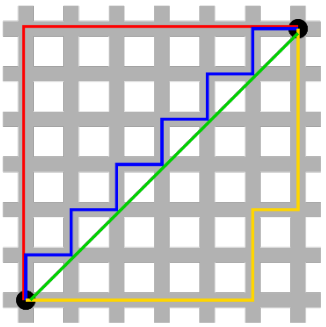
Si une fonction $d : E \times E \rightarrow R^{+¿¿}$ répond aux critères ci-dessus. C'est une fonction de distance. Nous allons en voir plusieurs.

Nous allons utiliser une grille de coordonnées $0,1,\ldots,N \in N$ sur deux dimensions pour représenter les positions. Notre ensemble sera $E = N^2$ car nos positions sont sur deux dimensions $\{x, y\} \in E$.

1. **Distance discrète** : Sur n'importe quel ensemble, la distance discrète d est définie par :
   Si $a = b$ alors $d(a,b) = 0$ et sinon, $d(a,b) = 1$
   C'est la distance la plus simple possible.

2. **Distance de Hamming** : La distance de Hamming compte le nombre de symboles qui diffèrent entre deux points, ici nous allons simplement évaluer les symboles x et y de la position. Donc la valeur de la distance entre deux points sera 1 si les deux points diffèrent que sur un seul axe, 2 si les deux points diffèrent sur les deux axes. (0 si les deux points sont égaux).

3. **Distance de Manhattan** : La distance de Manhattan dans un espace à deux dimensions est obtenue ainsi $d(a,b) = |x_a - x_b| + |y_a - y_b|$. Il s'agit de la somme des différences sur chacun des axes. Représenté en rouge, bleu et jaune sur l'image ci-contre (la distance usuelle (Euclidienne) est marquée en vert). La distance de Manhattan s'appelle aussi taxi-distance, c'est la distance entre deux points parcourus par un taxi lorsqu'il se déplace dans une ville où les rues sont agencées en réseau ou quadrillage.



4. **Distance Euclidienne** : La distance Euclidienne ou « à vol d'oiseau » est la distance que l'on utilise habituellement, elle est définie comme suit $d(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$.

5. **Somme des différences au carré** : Si l'on prend la distance Euclidienne au carré (ou simplement la formule de la distance Euclidienne sans la racine) on se retrouve avec la somme des différences au carré. Ceci n'est pas une fonction de distance car elle ne respecte pas l'inégalité triangulaire. Par exemple :

$$d(\{0,0\},\{0,2\}) \not\leq d(\{0,0\},\{0,1\}) + d(\{0,1\},\{0,2\}).$$
$$(0-0)^2 + (0-2)^2 \not\leq ((0-0)\|2 + (0-1)^2) + ((0-0)\|2 + (0-1)^2)$$
$$4 \not\leq 1 + 1$$

Une telle fonction est ce que l'on appelle une « semi-distance ». Elle permet de donner une valeur à l'éloignement entre deux points mais ne permet pas de former un espace métrique (espace où l'inégalité triangulaire est respectée). En gros elle permet de juger de l'éloignement entre deux points, mais parfois, selon cette « semi-distance », passer par un troisième point rend le trajet plus court (comme démontré dans l'exemple ci-dessus).

Nous allons implémenter certaines de ces distances pour évaluer leurs performances dans un algorithme de « path finding » donné. Dans cet algorithme on aimerait déterminer l'éloignement du but et si l'on est arrivé au but (distance nulle), ceci fonctionne aussi avec une « semi-distance ». Nous allons évaluer des positions par rapport à leur éloignement du but afin de pouvoir explorer les positions plus proches en priorité.

L'algorithme utilisé est A* « A star » ou « A étoile » en français[5]. C'est un algorithme de recherche de chemin dans un graphe (par exemple positions accessibles sur une carte) entre un nœud initial (par exemple un joueur) et un nœud final « but » (par exemple un trésor). Afin que l'algorithme puisse évaluer une position il a besoin d'une notion de distance (d'éloignement) entre la position et le but.

a) Les fonction sont à implémenter dans le fichier « ***arm/path/student_functions_asm.S*** », la gestion des paramètres et la valeur de retour vous est donnée. Nous n'allons pas implémenter la fonction de distance Euclidienne car elle requiert une racine carrée, il est possible de la calculer avec le coprocesseur à virgule flottante mais cela demande un peu de travail.

La signature des fonctions est la suivante :

```
uint32_t xxx_distance_asm(const uint32_t a, const uint32_t b);
```

Où les positions sont définies ainsi et converties en entiers 32-bits de la manière suivante (pour faciliter le passage des arguments en assembleur) :

```
typedef struct Position {
	uint16_t x;
	uint16_t y;
} Position;

Position Pos_a = {3,4};
uint32_t a = (Pos_a.x << 16) | Pos_a.y; // x se trouve au poids fort, y au poids
faible
```

b) Testez le bon fonctionnement de vos fonctions de distance. Pour ce faire un programme « *print* » est compilé, il affiche les distances par rapport à une cible '@' en position {2,2} sur une grille.

[5] https://fr.wikipedia.org/wiki/Algorithme_A*

Étape 5 – ARM Path Finding – Gestion du terrain

Dans cette étape nous allons nous intéresser à la gestion du terrain pour notre algorithme de « Path Finding ». Il s'agira de détecter si le terrain donné (dans une carte) autour de la position du joueur est traversable. Par exemple s'il y a un mur le joueur ne pourra pas passer.

D'abord profitons de tester nos fonctions de distance dans l'algorithme A* grâce au programme « *path1* » qui va utiliser les fonctions pour trouver un chemin entre un player 'p' et une cible '@'.

a) Lancez le programme « *path1* » et observez le résultat. Si vos fonctions de distance sont correctes un chemin entre le player et la cible devrait être trouvé. Cependant il ne tient pas encore compte du terrain de la map.

Il faudra implémenter la fonction qui génère les directions possibles depuis une position et une map donnée dont la signature de la fonction est la suivante :

```
uint32_t get_directions_asm(const char *map, const uint32_t position);
```

La position est encodée comme à l'étape précédente, les x,y 16-bits chacun dans un mot 32-bits avec x au poids fort. La carte est un pointeur sur caractère et les dimensions sont fixées par les constantes MAP_SIZE_X et MAP_SIZE_Y.

```
+-----------> Y                La représentation en mémoire est ligne par ligne :
| ..........                    .......... .......... ..W....... ..W....... etc.
| ..........                    Le pointeur char *map pointe sur le premier
caractère
| ..W.......                    si l'on incrémente le pointeur on pointe sur le
prochain
| ..W.......                    caractère à droite ou celui de la prochaine ligne
si l'on
| ..W.......                    est en bout de ligne.
| ..W....@..
| p.W.......                    Légende :
| ..W.......                    p : Player (position = {6,0})
V                               @ : Target (position = {5,7})
X                               W : Wall
```

Il s'agira d'aller voir si on peut se déplacer dans les directions N,S,E,W (Nord, Sud, Est, Ouest). S'il y a un mur (encodé par le caractère 'W' comme « wall ») on ne peut pas passer, si on a atteint le bord de map on ne peut pas passer non plus.

La valeur de retour est encodée comme suit :

Bit n° 31 …	3	2	1	0
28-bits de poids fort	N	S	E	W

Si la direction donnée est passable le bit correspondant est mis à 1. Donc si toutes les directions sont passables on retourne 0xF = 0b1111. Si le joueur se trouve dans une impasse et qu'il ne peut seulement aller vers le nord on retourne 0x8 = 0b1000. Si aucune direction n'est possible on retourne 0.

Dans l'exemple ci-dessus le player est en position {6,0} pourrait se déplacer au nord, sud, et à l'est mais pas à l'ouest car il est en bord de map. Donc le résultat de la fonction devra être 0xE = 0b1110 pour la map donnée et la position {6,0}.

b) Implémentez la fonction « *get_directions_asm()* » dans le fichier « ***student_functions_asm.S*** ».

c) Testez grâce au programme « *path2* » qui utilise vos fonctions de distance ainsi que votre fonction de génération des voisins. Le chemin trouvé en solution ne devrait pas traverser les murs. Donc dans la map « *prison* » le player ne devrait plus pouvoir s'échapper.

⇒ Rendez un code compréhensible et commenté.
⇒ Il est possible de changer les maps testées dans le fichier *« **path.h** »*.
⇒ Le fait que l'algorithme trouve la solution ne veut pas forcément dire que vos fonctions de distances sont « justes » mais qu'elles suffisent à trouver une solution.