Programmation assembleur (ASM)

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

Développement croisé et *Makefile*

lab02 (08.03.2023)

Objectifs de laboratoire

L'objectif de ce laboratoire est de se familiariser avec la notion de *Makefile,* les outils de développement croisé ainsi que l'utilisation du moniteur *U-Boot*[1].

Le développement croisé se fera depuis une machine hôte (la VM avec une architecture de type x86_64) et consiste en la génération d'exécutables binaires pour des machines avec une architecture différente (de type **x86_32** et **ARM**). Les machines cibles seront émulées dans la VM grâce à l'environnement virtuel *QEMU*[2].

Echéance

- Le laboratoire sera rendu au plus tard la veille 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_lab02$ ./rendu.sh
```

Cette commande va générer un fichier *asm22_lab02_rendu.tar.gz* 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***

La première étape à effectuer permet de récupérer l'environnement complet nécessaire aux laboratoires ASM.

a) La récupération du dépôt s'effectue à l'aide de la commande suivante :

```
reds@reds2022:~$ retrieve_lab asm lab02
```

[1] https://www.denx.de/wiki/U-Boot
[2] https://www.qemu.org

Étape 2 - Premiers pas avec *U-Boot* et déploiement d'applications sur *x86*

Cette étape est dédiée à la prise en main du moniteur *U-Boot.* Le but consistera à déployer une application depuis *U-Boot* dans l'environnement émulé.

a) Dans le répertoire *x86*, compiler les sources avec la commande *make*.

Quelle est la *toolchain* utilisée ?

b) Le démarrage de l'émulateur s'effectue en lançant un des scripts disponibles à la racine du dossier. Par exemple, le lancement de *QEMU* pour l'émulation d'un CPU x86 s'effectue de la manière suivante :

```
reds@reds2022:~/asm/asm_lab02$ ./st86
```

c) Dans *U-Boot* (le *bootloader* ou moniteur), des commandes sont disponibles. La commande *help* permet d'obtenir des informations sur ces commandes. La commande *help* suivi d'une autre commande permet d'obtenir de l'aide pour la commande en question. Lire l'aide pour les commandes suivantes (P. ex. « help go ») :

- go
- md
- mm
- mw
- printenv
- setenv
- tftp

La commande *printenv* sera souvent utilisée par la suite.

⇒ Le transfert d'un fichier binaire (*image binaire*) depuis la machine hôte vers la machine cible (système émulé) s'effectue via *tftp (trivial file transfer protocol)*. A titre d'exemple, la variable d'environnement « *hello* » montre la commande *tftp* à exécuter pour charger l'image en mémoire.

⇒ La définition d'une telle variable est réalisée à l'aide de la commande « ***setenv*** » comme ci-dessous :

```
=> setenv hello "tftp x86/hello_world.bin"
```

d) Il est possible d'exécuter une commande enregistrée dans une variable avec « ***run*** ».

```
=> run hello
```

⇒ On remarque que l'image est chargée à l'adresse ***0x4'0000***. Cette adresse est définie dans la variable d'environnement « ***loadaddr*** ».

e) L'exécution de l'application ainsi chargée se fait avec la commande « ***go <addr>*** », comme suit :

```
=> go 0x40000
```

⇒ Pour quitter *QEMU*, il faut taper « **ctrl+a** » puis « **x** » (en relâchant « ctrl+a » avant « x »).

f) A présent, éditer le code de « *hello_world* » (version *x86*) en remplaçant le texte « Hello World » par « **Hello ASM x86** ».

g) Tester la nouvelle application.

⇒ Pour utiliser gdb avec QEMU, il suffit de lancer *QEMU* puis de démarrer la configuration « **(x86) hello_world – Debug** » dans Eclipse. Ajouter un breakpoint au début de la fonction « *main()* » et exécuter l'application dans QEMU avec la commande ***go <addr>***

Étape 3 - Déploiement d'applications sur *ARM*

a) Dans le répertoire *arm*, compiler les sources avec la commande *make*.

Quelle est la *toolchain* utilisée ?

b) Le lancement de *QEMU* pour *ARM* s'effectue à l'aide du script « *st* » comme suit :

```
reds@reds2022:~/asm/asm_lab02$ ./st
```

c) De la même manière que dans l'étape précédente, transférer l'image binaire correspondante au « *hello_world* » compilée pour ARM dans *U-Boot*.

d) Démarrer l'application et tester le fonctionnement.

Attention l'adresse de chargement n'est pas la même sur la cible ARM que sur la cible x86.

e) Comme dans l'étape précédente, éditer le code de « *hello_world* » en remplaçant le message affiché par « **Hello ASM ARM** » et tester le fonctionnement.

Vous avez maintenant un environnement de cross-développement complet (compilation, exécution en émulation et débug) pour x86_32 et ARM sur une machine hôte x86_64.

Étape 4 - Construction d'un *Makefile*

Le répertoire « *arm/makefiles/raspdrone* » contient une série de codes sources C (.c, .h) et assembleur (.S) utilisés dans un environnement de drone sur une plate-forme de type *Raspberry Pi*. Il s'agira ici d'écrire un fichier *Makefile* permettant la génération du fichier exécutable *raspdrone*. L'exécution sera testée sur la machine ARM émulée avec QEMU.

a) Ouvrir les fichiers sources (.c, .h) et identifier les dépendances entre les fichiers (notamment avec les directives #*include*).
b) Créer un fichier *Makefile* élémentaire permettant la génération de l'exécutable. Il est conseillé de lancer la commande *make* et de compléter le *Makefile* en analysant les messages d'erreur.

⇒ L'option « **--just-print** » aka « **--dry-run** » aka « **-n** » force *make* à (uniquement) afficher les commandes qu'il aurait exécuté. Ceci peut être utile si l'on veut s'assurer que les commandes sont les bonnes avant de les effectuer réellement (p. ex. pour les commandes « **rm** »). Ceci permet aussi de voir ce que ferait un Makefile qui pourrait provenir d'une source douteuse.

⇒ L'option « **--debug** » aka « **-d** » permet à *make* d'afficher des informations supplémentaires pour le *debug* ; cela va afficher toutes les évaluations que fait *make* en interne pour choisir quelles règles seront appliquées. Étant donné que *make* évalue tout un nombre de règles implicites, l'affichage peut être indigeste. L'option « **--debug** » (et non « -d ») permet de définir un niveau de verbosité (par ex. avec « **--debug=b** » pour « basic » l'affichage sera plus compact.

c) Constituer une librairie (*bibliothèque*, *archive*) ***libdriver.a*** rassemblant les fonctions implémentées dans les fichiers suivants : **motor.c**, **gyroscope.c** et **camera.c**.
d) Créer une cible logique (.PHONY) ***checklib*** qui permet de tester si la librairie doit être recompilée ou non et utiliser l'option « **-l** » au moment de l'édition des liens de l'exécutable *raspdrone pour linker avec la librairie*.

⇒ Utiliser la *toolchain* **arm-linux-gnueabihf** pour cross-compiler le programme.
⇒ L'utilitaire pour créer une archive est l'application ***ar*** de la *toolchain*. L'option "r" est suffisante.
⇒ Le compilateur et le *linker* utilisent des options spécifiques pour la localisation des librairies (-l, -L). Les pages *man* seront utiles pour leur compréhension.
⇒ Ne pas oublier d'utiliser « **$<** » (première dépendance) dans les règles d'inférence à la place de « **$^** » (toutes les dépendances) si nécessaire.

e) Editer le *Makefile* et rajouter l'option « **-T asm.lds** » au *linker* afin de passer les commandes supplémentaires du linker script **asm.lds** au linker.

⇒ Des informations détaillées sur l'option « -T » peuvent être trouvées dans le manuel du linker.
⇒ Lors de la consultation d'une page *man*, l'utilisation du caractère « / » permet de rechercher rapidement une chaînes de caractères. Par exemple, taper « /-T » « Enter » conduira à la première occurrence de « -T », la touche « N » permet d'aller à la suivante.

f) Rajouter l'option « **-marm** » à la règle de compilation des fichiers C, afin de produire de l'assembleur ARM 32-bit.
g) Rajouter une règle pour la génération du **.bin** nécessaire au déploiement sur cible. La génération du binaire se fait grâce à l'outil *objcopy (*de la bonne toolchain*)* après la phase de « *linkage* ». L'option à passer à *objcopy* est « **-O binary** » (cf. *man objcopy*).
h) Ajouter le flag de compilation **-g** aux bonnes commandes pour générer les symboles de débug, nécessaires pour débuguer l'application.
i) Compiler, déployer sur la cible ARM et tester l'application à l'aide du *debugger* mettre un *breakpoint* à l'entrée de la fonction « *main()* » (fichier *engine.c*) par exemple, et faire du pas-à-pas sur quelques instructions.

Étape 5 - Debugger une application *ARM ou x86* dans l'environnement émulé

Cette étape permet d'exercer des manipulations avec le *debugger* dans *Eclipse*. Une application simple appelée « ***crackme*** » est fournie et permettra d'analyser les instructions en assembleur durant son exécution. Cette application demande un mot de passe afin de poursuivre son exécution. Cette étape consiste à identifier les instructions assembleur du test du mot de passe, et à changer son résultat. **Le code source ne doit pas être modifié**. Il est possible d'effectuer cette étape pour *ARM*, *x86*, ou les deux.

a) Dans le répertoire « *arm/* ou *x86/* », compiler les sources avec la commande « *make* ».
b) Démarrer l'émulateur pour la cible de votre choix.
c) Connecter le client *gdb* depuis *Eclipse* en utilisant la bonne configuration de *debug* « **(arm) crackme – Debug** » ou « **(x86) crackme – Debug** »
d) Transférer le binaire de l'application *crackme (crackme.bin)* et démarrer son exécution.
e) Lorsque l'application attend une entrée de l'utilisateur, interrompre son exécution (bouton jaune *pause*) et mettre un *breakpoint* sur la fonction « **check_password()** ». Continuer l'exécution afin de pouvoir entrer un mot de passe. Une fois le *breakpoint* atteint, effectuer une exécution pas-à-pas avec *F5* avec le focus sur la fenêtre de désassemblage.
f) Trouver l'instruction qui effectue le test correspondant à « (**hash == MY_HASH**) » et identifier le registre de destination du résultat. Utiliser le *debugger d'Eclipse* afin de changer ce résultat et passer le test.
g) Une fois votre approche validée, la décrire dans le fichier « **crackme_reg.txt »** à la racine du dossier « *asm_lab02* ». Par exemple,

```
x86 : J'ai mis le registre %eax après l'instruction mov à l'adresse 0x4002b à 0.
ARM : Indiquer si les deux approches ont été effectuées ; cependant l'approche sur une seule
cible est suffisante.
```

⇒ Quelles seraient d'autres approches possibles pour contourner « *check_password()* » ?
⇒ Serait-ce une bonne idée lorsqu'on est dans « *check_password()* » de modifier le « *Program Counter (PC)* » du processeur afin de continuer l'exécution directement aux instructions qu'on aimerait atteindre (ici le « *printf("Good job !\n");* ») ? Si non, pourquoi ?
(NB : sur *x86,* le registre PC s'appelle IP (*Instruction Pointer*, registre %eip).

Étape 6 – Modification d'un binaire ARM ou x86

Comme l'étape précédente, on s'intéresse ici à passer outre la fonction « *check_password()* » de l'application « *crackme* ». Cette fois-ci la modification se fera directement sur le binaire chargé dans *U-Boot*. Il s'agira d'identifier l'instruction de saut « ***jump*** » qui aiguille la suite du déroulement du programme après l'appel de « *if (check_password(password))* » afin de la remplacer pour arriver à contourner le branchement normal du « *if ( )* ». Cette étape peut être réalisée pour x86, ARM, ou les deux. **Le code source ne doit pas être modifié**.

Si l'exercice précédent n'a été réalisé que pour une seule cible (ARM, x86), utiliser ici une cible différente.

a) Dans le répertoire « *arm/* » ou « *x86/* », compiler les sources avec la commande *make*.
b) Inspecter les instructions assembleur de l'exécutable (ELF) « *crackme* » avec l'utilitaire *objdump* de la *toolchain* correspondante) afin de trouver les instructions relatives au saut conditionnel « *branch* » dans le jargon ARM et « *jump* » pour x86.

⇒ L'option « --source » aka « -S » permet d'entremêler les instructions assembleur et le code C (si disponible) ceci permet de plus facilement identifier les parties du code assembleur correspondant au code C.

c) Prendre note de l'adresse de l'instruction de saut ainsi que de sa longueur (nombre d'octets), la première colonne de la sortie de « *objdump* » représente l'adresse et la seconde sont les octets de l'instruction. La dernière colonne étant l'instruction elle-même.
d) Démarrer l'émulateur pour la cible de votre choix.
e) Charger le binaire *crackme.bin* via *tftp* et utiliser les instructions **md** et **mw** de U-Boot étudiées à l'étape 2 afin de remplacer l'instruction de saut identifiée à l'étape c) avec des opérations « *NOP* » (*No operation*).

⇒ L'opération « *NOP* » en **x86** est encodée sur un seul octet : 0x90
⇒ L'opération « *NOP* » en **ARM** peut être encodée sur 32-bit avec la valeur 0x00000000

f) Exécuter l'application avec « *go <adresse>* » et vérifier que les modifications ont l'effet attendu.
h) Une fois l'approche validée, la décrire dans le fichier « **crackme_mod.txt** » à la racine du dépôt (dossier « *asm_lab02* »).

Dans l'exemple ci-après, un *xor* est remplacé, mais dans l'exercice c'est bien une instruction de saut qui sera remplacée :

```
x86 : J'ai remplacé l'instruction xor (0x31 0xC8) à l'adresse 0x40039 :
        40039:      31 c8               xor    %ecx,%eax

Par deux instructions « nop » (0x90)
De cette manière … et le message attendu est affiché !
```

⇒ La commande « **md.l** <adresse> <#bytes à afficher> » permet d'afficher le contenu de la mémoire dans *U-Boot."*
⇒ La commande « **mw.b** <adresse> <valeur sur 1 byte> » écrira 1 octet.
⇒ La commande « **mw.l** <adresse> <valeur sur 4 bytes> » écrira 4 octets.