INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Enseignant : *Vincent Migliore*
vincent.migliore@insa-toulouse.fr

Architecture Matérielle 3^e année
Introduction aux architectures ARM, x86 et x64

Objectifs

L'objectif est d'explorer les aspects principaux des basses couches des logiciels, fortement contraintes par l'architecture matérielle. On s'intéressera particulièrement à la manière dont le compilateur C organise les appels de fonctions et le stockage des données[a], et à certains aspects de la sécurité qui en dépendent.

Cette exploration se fera principalement au moyen du désassemblage, ce qui suppose bien entendu une bonne compréhension du langage d'assemblage. Certains compilateurs peuvent produire un listing d'assemblage (par exemple gcc avec l'option -S), il existe aussi des services de listing d'assemblage en ligne, cependant il peut être très laborieux d'analyser l'exécution d'un programme sur un listing statique.

Il est beaucoup plus productif d'observer le fonctionnement du programme "en live" au moyen d'un debugger. C'est aussi la seule solution si on ne dispose pas du code source du programme. Ce sera notre stratégie pour cette série de TP.

[a]Note : Pourquoi le langage C, qui n'est pas forcément le préféré des développeurs ? C'est le langage de référence, qui sert à créer les noyaux et les bibliothèques de base des systèmes d'exploitation dominants (Unix, Linux, Windows, Android). Les compilateurs des autres langages sont contraints d'adopter les usages du C notamment pour pouvoir accéder aux bibliothèques système.

Introduction

1 Architectures ciblées

On s'intéressera aux 3 architectures dominantes du monde informatique, à savoir ARM, x86 et son extension 64 bits nommé x64. Elles sont aussi représentatives de 2 tendances historiques ; RISC et CISC.

1.1 Familles : RISC / CISC

Après que les chercheurs des universités de Stanford et Berkeley aient formalisés le concept RISC (à partir de 1981), l'acronyme CISC a été appliqué rétroactivement à toutes les architectures non-RISC (développées à partir de 1950), mais il n'y a pas de spécification formelle pour le concept CISC. Ces deux architectures possèdent néanmoins des différences caractéristiques :

- CISC = Complex Instruction Set Computer
  - → code d'instructions de longueur variable à partir de 1 byte (pas d'alignement) ;
  - → développement incrémental d'une famille de CPUs de plus en plus perfectionnés ;
  - → nombreux registres spécialisés ;

→ registres de tailles diverses éventuellement imbriqués ;

→ extensions du jeu d'instructions facilité par le microcodage ;

→ durée d'exécution d'une instruction très variable.

– Exemples :

* IBM 360 (mainframe)
* Motorola 6800 (µP 8 bits), 68000 (µP 32 bits)
* Intel 4004 (µP 4 bits), 8080 (µP 8 bits), 8051 (µC 8 bits), X86 (µP 16 bits, 32 bits, 64 bits)

- RISC = Reduced Instruction Set Computer

→ code d'instructions de longueur fixe = 1 mot de mémoire programme, pour faciliter la mise en œuvre d'un pipe-line ;

→ durée d'exécution d'une instruction = 1 cycle d'horloge, aussi pour minimiser la latence de traitement des interruptions ;

→ registres banalisés ;

→ éventuellement architecture load-store (la complexité des modes d'adressage est concentrée sur 2 instructions).

– Exemples :

* ARM (µP et µC 32 bits, 64 bits)
* Sun SPARC (µP 32 bits)
* PowerPC (µP 32 bits)
* Microchip/Atmel PIC (µC 8bits, 16bits), ATmega(µC 8 bits)

Après avoir été en compétition, les deux familles ont fini par converger :
les processeurs CISC se sont dotés de pipelines efficaces (Pentium, au prix d'un effort de conception énorme) le jeu d'instruction des RISC est devenu de plus en plus complexe, les règles ont été assouplies (ARM Cortex M : instructions Thumb2 de 16 et 32 bits, légère spécialisation des registres, instructions de longue durée mais interruptibles).

1.2 Les exemples choisis

- ARM est l'architecture la plus répandue dans le monde (en 2020) en termes d'unités fabriquées. Arm est le nom d'une entreprise qui a développé de nombreuses variantes d'une architecture RISC, mais laisse à d'autres le soins de fabriquer et commercialiser les circuits intégrés. Ici nous nous intéressons à la famille Cortex-M qui implémente un sous-ensemble du jeu d'instructions ARMv7-M ("M" pour "Microcontrôleur") avec un codage d'instructions sur 2 longueurs dit Thumb2.

- x86 est l'architecture la plus répandue (en comptant son extension 64 bits) pour les ordinateurs portables, les ordinateurs de bureau et les serveurs (web, cloud). On désigne par x86 une famille de processeurs 16 et 32 bits de la marque Intel, qui a comporté notamment les puces 8086, 80186, 80286, 80386, 80486, 80586, 80686 et Pentium. C'est l'exemple le plus caractéristique de progression incrémentale d'architecture CISC. La dénomination officielle de cette famille dans la doc Intel est IA-32 (depuis le 80386). Elle est aussi désignée par i386 et i686 dans la doc des outils GNU.

- x64 (historiquement AMD64) est une extension à 64 bits appliquée à l'architecture x86. Proposée par AMD, entreprise rivale mais aussi partenaire d'Intel, cette extension a été adoptée par Intel, sous la dénomination Intel 64. Elle est aussi désignée par x86-64 dans la documentation des outils GNU.

Rétrocompatibilité : La spécification impose à un processeur X86-64 de pouvoir fonctionner en mode 32 bits, donc exécuter un système d'exploitation et des applications compilés pour X86 32 bits. Il est possible de faire un système d'exploitation 64 bits supportant des applications fonctionnant en mode 32 bits (ex. Windows 10). Bien que la fabrication en masse de processeurs X86 32 bits soit arrêtée, les logiciels 32 bits ne sont donc pas obsolète.

Vu le temps disponible on se limitera au coeur du modèle d'exécution, on n'abordera pas les extensions pour le calcul en virgule flottante (floating point) ni les opération parallèles (SIMD = Single Instruction Multiple Data). De plus la gestion de mémoire virtuelle (mémoire paginée) sera abordée dans un autre module.

Plus d'info : Survol des jeux d'instructions ARM, x86, x64.

2 Outils

Pour étudier le comportement bas niveau des différentes architectures, nous utiliserons les chaînes d'outils offertes par le projet GNU, une chaîne distincte par architecture. Cette chaîne comprend outre le compilateur gcc, l'utilitaire make, le debugger gdb et le shell bash.

Votre processeur étant d'architecture x64, il peut faire tourner nativement du code compilé pour cette architecture, de même que le x86 par rétrocompatibilité. En particulier, nous utiliserons directement la chaîne d'outils gcc de votre machine.

Pour l'architecture armv7-m de ARM, la situation est plus délicate, votre processeur ne peut pas faire tourner nativement du code pour cette architecture, il est donc nécessaire de, soit: 1) utiliser une plateforme matérielle embarquant un tel processeur, ou 2) simuler une plateforme. Nous allons opter pour la deuxième possibilité pour ce TP, et pour se faire nous allons utiliser le simulateur open-source qemu. qemu est un outil de simulation et de virtualisation de machine particulièrement puissant, prenant en charge un grand nombre d'architectures différentes et est capable de émuler des systèmes exploitation. Dans le cadre de ce TP, nous nous restreindrons à l'utilisation d'un système minimal de type Bare Metal. Pour la compilation, nous utiliserons ce qui s'appelle la cross-compilation, c'est-à-dire utiliser un compilateur compilé pour s'exécuter sur une machine donnée (ici votre machine), permettant de compiler du code pour une autre machine (ici le cortex-M3 qui sera simulé par qemu).

3 Organisation

3.1 Expériences

Le TP sera constitué d'une séquence d'expériences. Le temps nécessaire à l'accomplissement d'une expérience est très variable, aussi le nombre d'expériences à terminer à chaque séance de TP n'est pas prédéterminé. Les expériences 1, 2 et 3 doivent être traitées entièrement. Les expériences suivantes sont optionnelles et permettent de traiter des points à la limite du hors programme mais néanmoins très enrichissantes.

Chaque expérience portera sur un petit programme qui, dans le cas général, sera compilé et testé sous debugger sur les 3 architectures ciblées, en vue de comparer les solutions spécifiques à chacune d'elles. Dans certains cas, le programme devra être expérimenté avec différentes options d'optimisation.

L'objectif concret de chaque expérience sera de trouver les réponses à une série de questions.

3.2 Impact des optimisations

Sur les 3 architectures ciblées, le compilateur C permet de choisir un niveau d'optimisation désigné par un numéro de 0 à 3.

- Au niveau 0, chaque opération est traitée indépendamment des autres, ce qui entraîne des répétitions d'actions telles que les accès mémoires. Cependant les expressions ne contenant que des constantes sont déjà pré-calculées par le compilateur. *Le niveau 0 est recommandé pour l'utilisation du debugger dans la recherche de bugs.*

- A partir du niveau 1, les accès mémoire sont mutualisés et l'ordre des actions peut être modifié.
- A partir du niveau 2, des simplifications très agressives sont appliquées, telles que déroulage des boucles for, la mise à plat de l'appel de certaines fonctions, et la suppression de tout code qui ne produit pas d'effet "perceptible" de l'extérieur, etc...

Note : Il existe également une optimisation visant à réduire au maximum la taille du binaire final (-s), particulièrement utile dans le domaine de l'embarqué vu les contraintes fortes sur les ressources matérielles. Elle ne sera pas traitée durant ces expérimentations.

Dans ces TP nous nous intéresserons à l'observation des mécanismes d'optimisation, aussi nous affronterons les difficultés du debug sur code optimisé.

Les petits programmes de nos expériences ne font pas grand-chose d'utile, aussi pour éviter que l'optimiseur supprime complètement leur contenu, les précautions suivantes sont prises :

- Une partie des données constantes seront traitées par la fonction seed(), qui masquera leur caractère constant ;
- Au moins une donnée de sortie sera traitée par la fonction expose(), qui la transmettra vers un périphérique (on pourrait aussi protéger cette données de l'optimisation avec la directive volatile) ;
- Les fonctions seed() et expose() seront compilées séparément (fichier io.c) pour que l'optimiseur ne puisse pas les mettre "à plat".

3.3 Les sujets d'expériences

- Expérience 1 : variables locales
- Expérience 2 : arguments d'une fonction
- Expérience 3 : débordement de pile

4 Documentation

4.1 Documents communs

- Survol des jeux d'instructions ARM, X86, AMD64 ;
- Fonctionnement de la pile.

4.2 Documents de référence ARM

- Le manuel de référence ARMv7-M ;
- Le document Quick Reference Card ;
- L'aide-mémoire de l'INSA.

4.3 Documents de référence X86

- Le manuel de référence Intel 64 and IA-32 Architectures Software Developer's Manual ;
- Le site x86asm.net de Karel Lejska ;
- le résumé de codage de Daniel Plohmann

5 Conventions

- le nom des fichiers exécutables est le même pour chaque expérience ;
- **arm.elf** fait référence au fichier exécutable relogable du cortex-M ;
- **arm.bin** fait référence au fichier exécutable du cortex-M ;
- **x86.bin** fait référence au fichier exécutable relogable du x86 ;
- **x64.bin** fait référence au fichier exécutable relogable du x64 ;
- **arm-gdb** fait référence au debugger gdb pour cible ARM ;
- **gdb** fait référence au debugger gdb pour cible x86 et x64 ;
- **qemu-system-arm** fait référence au simulateur qemu pour cible ARM ;
- **CFLAGS** fait référence à une variable du makefile permettant de passer des options à gcc ;
- **ARM_CFLAGS** fait référence à une variable du makefile permettant de passer des options à gcc pour ARM uniquement ;
- **x86_CFLAGS** fait référence à une variable du makefile permettant de passer des options à gcc pour x86 uniquement ;
- **x64_CFLAGS** fait référence à une variable du makefile permettant de passer des options à gcc pour x64 uniquement.

Préparation et Verification de l'environnement de travail

Préparation

Exécutez le scription setup.sh à la racine des ressources du TP. Ce script va télécharger une archive contenant la chaîne de compilation gnu pour le cortex-m depuis les serveurs d'Arm. L'extraction peut prendre un certain temps, aussi pour éviter tout problème par la suite, n'interrompez pas le processus d'extraction de l'archive.

Verification

Ouvrez un terminal dans le dossier HELLOWORLD puis executez les commandes suivantes (dans l'ordre) :

- **make**
- **make test**

Si vous n'obtenez pas de messages d'erreurs et que la chaîne de caractères *hello world!* s'affiche 3 fois, vous êtes bon pour la suite.

EXP01: variables locales

Préparation

Ouvrez un terminal dans le dossier EXP01.

ARM

Optimisation 0 (-O0)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 0 avec **make CFLAGS=-O0** ('O' zéro)
3. exécutez le script **gdb-arm.sh** pour lancer l'environnement de debug pour ARM.

Question Q1.1.1

Où les variables locales aa, bb, cc, dd, ee de la fonction sub01 sont-elles stockées ?

Question Q1.1.2

Comment l'espace de stockage pour ces variables est-il réservé ? Est-il libéré après usage ?

Optimisation 1 (-O1)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 1 avec **make CFLAGS=-O1**
3. exécutez le script **gdb-arm.sh**.

Question Q1.1.3

En quoi le stockage des variables locales a-t-il été amélioré ?

---

x86

Optimisation 0 (-O0)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 0 avec **make CFLAGS=-O0**
3. exécutez :
   - **gdb x86.bin** pour lancer l'environnement de debug pour x86.
   - **layout split** (pour afficher les fenêtres commandes, assembleur et source).
   - **b main** (pour ajouter un point d'arrêt au niveau du main.
   - **run** (pour lancer l'exécution du programme, qui s'arrêtera au premier point d'arrêt trouvé).

Question Q1.2.1

Où les variables locales aa, bb, cc, dd, ee de la fonction sub01 sont-elles stockées ?

Question Q1.2.2

Comment l'espace de stockage pour ces variables est-il alloué ? Est-il libéré après usage ?

Optimisation 1 (-O1)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 1 avec **make CFLAGS=-O1**
3. exécutez **gdb x86.bin**.

Question Q1.2.3

Qu'est-ce qui a changé dans le stockage des variables locales aa, bb, cc, dd, ee de la fonction sub01 ?

Question Q1.2.4

Comment les opérations arithmétiques portant sur des constantes on-t-elles été optimisées ?

x64

Optimisation 0 (-O0)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 1 avec **make CFLAGS=-O0**
3. exécutez **gdb x64.bin** pour lancer l'environnement de debug pour x64.

Question Q1.3.1

Précisez les différences qu'il y a avec le x86 sur le stockage des variables aa, bb, cc, dd et ee de la fonction sub01.

Optimisation 1 (-O1)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 1 avec **make CFLAGS=-O1**
3. exécutez **gdb x64.bin**.

Question Q1.3.2

Précisez les différences qu'il y a avec le x86 dans l'optimisation des opérations arithmétiques portant sur des constantes.

EXP02: arguments d'une fonction

Préparation

Ouvrez un terminal dans le dossier EXP02.

ARM

Optimisation 0 (-O0)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 0 avec **make CFLAGS=-O0**
3. exécutez le script **gdb-arm.sh**.

Question Q2.1.1

Comment la fonction seed() retourne-t-elle sa valeur ?

Question Q2.1.2

Comment les arguments aa, bb, cc, dd, ee sont-ils passés à la fonction sub01() ?

Optimisation 2 (-O2)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 2 avec **make CFLAGS=-O2**
3. exécutez le script **gdb-arm.sh**.

Question Q2.1.3

En quoi l'appel à la fonction sub01() a-t-il été amélioré ?

---

x86

Optimisation 0 (-O0)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 0 avec **make CFLAGS=-O0**
3. exécutez **gdb x86.bin**.

Question Q2.2.1

Comment la fonction seed() retourne-t-elle sa valeur ?

Question Q2.2.2

Comment les arguments aa, bb, cc, dd, ee sont-ils passés à la fonction sub01() ?

Remarque

A la fin de la fonction main(), l'instruction leave copie le registre ebp (base pointer) dans esp, puis dépile ebp.

Ceci suppose qu'au début de la fonction, ebp avait été empilé puis esp avait été copié dans ebp, afin de mémoriser dans ebp la base de l'espace de pile de cette fonction.

(Il semble que l'option -fomit-frame-pointer est ignorée pour la fonction main())

Optimisation 1 (-O1)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 1 avec **make CFLAGS=-O1**
3. exécutez **gdb x86.bin**.

Question Q2.2.3

Comment le traitement des arguments aa, bb, cc, dd, ee à l'intérieur de la fonction sub01() a-t-il été optimisé ?

Optimisation 2 (-O2)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 2 avec **make CFLAGS=-O2**
3. exécutez **gdb x86.bin**.

Question Q2.2.4

Comment l'appel à sub01() a-t-il été optimisé ?

Question Q2.2.5

Quel est l'impact sur les opérations arithmétiques ?

---

x64

Optimisation 1 (-O1)

Préparation

1. Nettoyez le projet avec **make clean**
2. Compilez avec l'optimisation à 1 avec **make CFLAGS=-O1**
3. exécutez **gdb x64.bin**.

Question Q2.3.1

Comment les arguments aa, bb, cc, dd, ee sont-ils passés à la fonction sub01() ?

EXP03: débordement de pile

Ce programme remplit un tableau avec des nombres de la forme **aa + i × i** , pour **i** allant de 0 à 8. Ensuite, le programme additionne à chaque élément la somme des précédents, dans le même tableau. Avec **aa = 0x1000**, le résultat attendu pour le dernier élément est : **37068 = 0x90cc**.

Préparation

Ouvrez un terminal dans le dossier EXP03.

Remarque

Lorsque vous codez un programme en langage C, vous pouvez définir ce qui s'appelle des macros : Ce sont des portions de code qui sont défini à l'aide de la commande **#define**. Avant de procéder à la compilation du code, le préprocesseur vient substituer dans le code toute occurrence du nom de macro par sa définition. Par exemple, **#define filesize 512** remplacera la chaine de caractère **filesize** dans votre code par la chaine de caractère **512** (qui sera interprété comme un entier). Il est possible de passer à gcc des macros directement depuis le terminal en mettant un prefix **-D** au nom de la macro. En gardant le même exemple, il suffit de passer à gcc la commande **-Dfilesize=512**.

Dans cette expérience, nous allons regarder l'impact de l'accès à des éléments qui se trouvent au delà de l'espace initialement prévu pour un tableau. En particulier, vous avez accès à la macro **BUFOVERFLOW** qui contrôle combien d'éléments au delà du tableau sont accédés (regardez le code c pour bien comprendre son fonctionnement).

Remarque

Dans cette expérience, vous allez devoir passer plus d'un argument à la variable **CFLAGS**. Pour cela, vous devez encapsuler vos arguments entre des paretheses **"**.

ARM

Optimisation 0 (-O0) + débordement de 2 (-DBUFOVERFLOW=2)

Préparation

1. Nettoyez le projet et compilez avec l'optimisation à 0 et un débordement de 2 éléments avec : **make clean && make CFLAGS="-O0 -DBUFOVERFLOW=2"** ;
2. Préparez l'environnement de travail qemu/gdb pour ARM.

Verification

Vérifiez que la valeur soumise à **expose()** est bien la valeur prévue pour le dernier élément.

Optimisation 1 (-O1)
+ débordement de 2 (-DBUFOVERFLOW=2)

Préparation

1. Nettoyez le projet et compilez avec l'optimisation à 0 et un débordement de 2 éléments avec : **make clean && make CFLAGS="-O1 -DBUFOVERFLOW=2"** ;
2. Préparez l'environnement de travail qemu/gdb pour ARM.

Question Q3.1.1

Le déroulement du programme est-il correct ? Sinon, à partir de quelle instruction y a-t-il une déviation ?

Question Q3.1.2

Quelle est la cause de l'incident ?

Optimisation 0 (-O0)

Préparation

1. Nettoyez le projet et compilez avec l'optimisation à 0 et un débordement de 2 éléments avec : **make clean && make CFLAGS="-O0 -DBUFOVERFLOW=2"** ;
2. Préparez l'environnement de travail qemu/gdb pour ARM.

Question Q3.1.3

Quel est le mécanisme qui permet à ce programme de montrer l'apparence d'un fonctionnement normal ?

---

x86

Optimisation 0 (-O0)
+ débordement de 1 (-DBUFOVERFLOW=1)

Préparation

1. Nettoyez le projet et compilez avec l'optimisation à 0 avec : **make clean && make CFLAGS="-O0 -DBUFOVERFLOW=1"** ;
2. Préparez l'environnement de travail gdb pour x86.

Question Q3.2.1

Comment la détection du débordement de pile est-il effectué ?

Question Q3.2.2

Le mécanisme varie-t-il d'une exécution à une autre ?

Optimisation 0 (-O0) + débordement de 2 (-DBUFOVERFLOW=2) + suppression de la protection de la pile (-fno-stack-protector)

Préparation

1. Compilez le binaire avec l'optimisation 0 et en supprimant la protection de la pile avec la commande

   make clean && make CFLAGS="-O0 -DBUFOVERFLOW=2 -fno-stack-protector" ;

2. Préparez l'environnement de travail gdb pour x86.

Question Q3.2.3

Le déroulement du programme est-il correct ? Sinon, à partir de quelle instruction y a-t-il une déviation ?

Question Q3.2.4

Quelle est la cause de l'incident ?

Optimisation 1 (-O1) + débordement de 2 (-DBUFOVERFLOW=2) + suppression de la protection de la pile (-fno-stack-protector)

Préparation

1. Compilez le binaire avec l'optimisation 0 et en supprimant la protection de la pile avec la commande

   make clean && make CFLAGS="-O1 -DBUFOVERFLOW=2 -fno-stack-protector" ;

2. Préparez l'environnement de travail gdb pour x86.

Question Q3.2.5

Quelle est la cause de l'incident ?