

Document ARM

Groupe 41

Corentin HEUZÉ
Dorian VERNEY
Adrien AUBERT
Yuxuan LU
Yung Pheng THOR

Table de matière

1. Introduction	3
2. Spécification de l'extension	4
a. Comment utiliser l'extension ?	4
i. Installer l'environnement	4
ii. Lancer le compilateur ARM	4
b. Que fait l'extension ?	5
3. Analyse bibliographique	5
a. Émulateur : QEMU - ARM system emulator	5
b. Architecture ARM	7
c. Documentation ARM	8
4. Choix de conception, d'architecture et d'algorithmes	9
a. Choix de conception	9
b. Choix d'architecture	9
c. Exemple : Le juste prix	10
5. Méthode de validation	14
a. Tests	14
b. Intégration avec le compilateur	14
6. Résultats de la validation de l'extension	14

1.Introduction

L'architecture ARM est une architecture de processeur RISC (Reduced Instruction Set Computing) qui est largement utilisée dans les appareils électroniques portables et embarqués. Elle est notamment présente dans les smartphones, les tablettes, les ordinateurs portables, les automobiles, les robots et les objets connectés.

L'étude de l'architecture ARM est pertinente pour plusieurs raisons. Tout d'abord, elle permet de développer des connaissances fondamentales sur les processeurs RISC, qui sont de plus en plus utilisés dans les systèmes électroniques. De plus, l'architecture ARM est une technologie qui est largement utilisée dans l'industrie, ce qui en fait une compétence recherchée par les employeurs. Enfin, l'étude de l'architecture ARM permet de développer des compétences transférables à d'autres architectures de processeurs, comme l'architecture x86.

L'architecture ARM est utilisée dans une grande variété d'applications. Dans les appareils portables, elle permet de réduire la consommation d'énergie, ce qui est essentiel pour prolonger l'autonomie des batteries. Sur le plan de la consommation d'énergie, l'architecture ARM est jusqu'à 50 % plus efficace que l'architecture x86. Dans les automobiles, elle permet de réduire le coût et la consommation de carburant des véhicules. L'architecture ARM est également utilisée dans les automobiles de nouvelle génération. Elle permet de réduire la consommation de carburant et d'améliorer la sécurité des véhicules. Dans les robots, elle permet d'augmenter la puissance et l'efficacité des systèmes de contrôle. L'architecture ARM est une technologie prometteuse pour l'intelligence artificielle. Elle permet de réduire la consommation d'énergie des systèmes d'intelligence artificielle, ce qui est essentiel pour les applications mobiles. Et dans les objets connectés, elle permet de réduire la taille et le coût des appareils.

L'architecture ARM est en constante évolution et elle est de plus en plus performante. Elle est donc susceptible de devenir encore plus largement utilisée dans le futur. En effet, elle présente plusieurs avantages par rapport aux autres architectures de processeurs, notamment une consommation d'énergie réduite, une taille réduite et un coût plus faible.

2. Spécification de l'extension

a. Comment utiliser l'extension ?

i. Installer l'environnement

Tout d'abord avant de pouvoir lancer le compilateur ARM, il faut installer l'environnement.

Pour installer l'environnement sur Ubuntu (qemu-user, GNU As, libc compatible...) exécutez le script :

- `./arm-environment/scripts/ubuntu.sh <dest_dir>` (Deploy ARM environment)

Ensuite il faut sourcer le fichier `env.sh` situé à la racine de l'environnement

- `source <dest_dir>/env.sh` (Initialize ARM environment)

Vous avez désormais accès aux outils :

- `compile-deca-arm <file>.s` (Compiler un fichier ASM vers un exécutable ELF)
- `qemu-deca-arm <file>` (Lancer le fichier ELF en utilisant qemu user en mode émulation)

ii. Lancer le compilateur ARM

Pour lancer le compilateur, il faut :

- `deca -arm ./code.deca` (Generate a `./code.s` file)
- `compile-deca-arm ./code.s` (Build `./code.s` to `./code`)
- `qemu-deca-arm ./code` (Run newly built executable using qemu)

b. Que fait l'extension ?

L'extension ARM de notre projet permet de faire toute la partie sans objets, i.e. opération binaires, affichage, allocation de registres, etc.

Cependant, il resterait à implémenter la partie avec objets. Cela impliquerait de développer les fonctionnalités centrales de programmation orientée objets : La création d'objets, l'accès aux champs d'objets, les méthodes d'objets. L'implémentation de la partie avec objets nécessiterait donc de développer de nouvelles fonctionnalités dans l'extension ARM. Cela pourrait être fait en ajoutant de nouvelles instructions ou en modifiant les instructions existantes.

3. Analyse bibliographique

a. Émulateur : QEMU - ARM user emulation

Nous avons d'abord étudié les différentes architectures de processeurs ARM. Nous avons appris que l'architecture ARM est une architecture de processeur RISC (Reduced Instruction Set Computer) 32 bits. Elle est principalement utilisée dans les appareils mobiles, tels que les smartphones et les tablettes.

Nous avons ensuite recherché les différentes solutions permettant d'exécuter du code ARM sur une machine avec un processeur x86. Nous avons découvert qu'il existe deux principales solutions :

La compilation croisée : cette solution consiste à compiler le code ARM pour une architecture x86. Il s'agit de la solution la plus simple, mais elle peut entraîner une perte de performances.

L'émulation : cette solution consiste à utiliser un logiciel qui émule une machine ARM sur une machine x86. Cette solution est plus complexe, mais elle permet de conserver les performances du code ARM original.

Nous avons décidé d'utiliser le simulateur QEMU pouvant émuler du code ARM sur des machines.

Nous avons choisi QEMU car il est un simulateur open source, gratuit et multiplateforme. Il est également capable d'émuler une large gamme de machines ARM, ce qui nous permettait de tester notre code sur différents types d'appareils.

Pour utiliser QEMU, nous avons suivi les étapes suivantes :

1. Nous avons installé QEMU sur notre machine x86 (Ubuntu et Windows WSL).
2. Nous avons compilé le fichier ELF permettant d'exécuter de l'ARM 32 bits sous Linux.
3. Nous avons lancé QEMU user avec les bons paramètres, permettant de faire une édition dynamique des liens, notamment avec la librairie standard C.

Une fois QEMU lancé, nous avons pu exécuter notre code ARM sans problème. Nous avons effectué quelques tests pour nous assurer que notre code fonctionnait correctement sur QEMU. Nous avons également vérifié que les performances de notre code étaient satisfaisantes.

En conclusion, nous avons trouvé que QEMU était une solution idéale pour exécuter du code ARM sur une machine x86 (Ubuntu/WSL).

b. Architecture ARM

Par la suite, notre attention s'est tournée vers l'architecture ARM, une étape cruciale dans notre démarche, étant donné la nécessité de faire fonctionner ARM sur les machines Linux. Après avoir réussi à exécuter du code ARM sur une machine x86 avec QEMU, nous avons pu nous concentrer sur la prochaine étape de notre projet : l'intégration de l'architecture ARM dans le code source java..

Ainsi, une vigilance particulière a été accordée à l'Application Binary Interface (ABI) Linux. L'ABI Linux définit l'interface entre les applications et le système d'exploitation Linux. Elle est essentielle pour garantir la compatibilité entre les applications et les différentes versions de Linux. Pour ce faire, nous avons exploité les ressources disponibles sur le site azeria-labs.com, qui s'est avéré être une source précieuse d'informations et de conseils spécialisés. Le site Web d'Azeria Labs, ainsi que goodbolt.org fournit des informations détaillées sur l'ABI Linux, y compris les exigences spécifiques pour les applications ARM.

Cette démarche méticuleuse nous a permis d'assurer une intégration harmonieuse et efficace de l'architecture ARM dans notre environnement, renforçant ainsi la robustesse et la compatibilité de notre système.

En suivant attentivement les instructions d'Azeria Labs, nous avons pu garantir que notre code ARM fonctionnerait correctement sur Linux. Cela nous a permis de créer un système plus robuste et compatible, qui peut être utilisé sur une gamme plus large d'appareils.

L'ABI Linux définit l'interface entre les applications et le système d'exploitation Linux. Elle comprend des informations sur la façon dont les applications interagissent avec le système d'exploitation, telles que la façon dont elles appellent les fonctions du système d'exploitation, la façon dont elles manipulent la mémoire et la façon dont elles interagissent avec les périphériques.

Les applications ARM doivent respecter les exigences spécifiques de l'ABI Linux pour pouvoir fonctionner correctement sur Linux. Ces exigences comprennent des choses telles que la taille des registres, la disposition des structures et les types de données utilisés.

Il existe plusieurs façons d'exécuter du code ARM sur Linux. L'une des façons les plus courantes est d'utiliser un compilateur croisé. Un compilateur croisé est un compilateur qui est conçu pour compiler du code pour une architecture de processeur différente de celle de la machine sur laquelle il s'exécute.

c. Documentation ARM

Suite à cela nous devons "apprendre" et connaître le langage ARM.

Après avoir réussi à exécuter du code ARM sur une machine x86 avec QEMU et à l'intégrer dans un environnement Linux, nous avons pu nous concentrer sur la prochaine étape de notre projet : l'apprentissage du langage ARM. Nous avons donc exploité deux documentations.

La première documentation était la documentation officielle de ARM, qui est une ressource complète et détaillée sur l'architecture ARM. Cette documentation comprend des informations sur les instructions ARM, les registres ARM, les modes d'adressage ARM et les autres aspects de l'architecture ARM.

La deuxième documentation était une documentation plus simple et plus courte se trouvant sur iitd-plos.github.io. Cette documentation est une bonne introduction au langage ARM, mais elle ne fournit pas autant d'informations que la documentation officielle.

Ces documentations nous ont permis d'utiliser au mieux ARM. En lisant ces documentations, nous avons appris les bases du langage ARM. Nous avons également appris les instructions ARM les plus courantes et les modes d'adressage ARM les plus importants. Grâce à ces connaissances, nous avons été en mesure de commencer à écrire notre propre code ARM. Nous avons commencé par écrire des programmes simples, puis nous avons progressé vers des programmes plus complexes.

4.Choix de conception, d'architecture et d'algorithmes

a. Choix de conception

Comme dit précédemment on parcourt l'arbre décoré par la partie contextuelle et on génère notre code ARM. Cependant on utilise la libc pour pouvoir générer certaines fonctions.

b. Choix d'architecture

Pour les choix d'implémentation des structures de données, nous avons fait de la même manière que dans la partie codeGen classique

Fonctionnalités	Classes ajoutées
Gestion des registres disponibles ou non pour le calcul d'expressions	RegisterAllocator
Gestion de l'unicité des labels pour les expressions, les	On utilise les lignes et le colonnes de l'instructions pour avoir l'unicité des

ifThenElse et les while	labels
Gestion de la pile pour le placement des éléments dans la pile (Global base et Local base)	Stack

c. Exemple : Le juste prix

Des programmes comme le jeu du “Juste prix” fonctionne. On pourrait aussi penser à des fonctions mathématiques que l’on peut calculer de façon non réursive.

```

1  {
2      int secret_number = 1080;
3      int i = 1;
4      int input;
5
6      print("(", i, ") - " , "Entrez une valeur: ");
7      input = readInt();
8      i = i + 1;
9
10     while (input != secret_number) {
11         if (input > secret_number) {
12             print(input, " > SECRET_NUMBER");
13             println("");
14         } else if (input < secret_number) {
15             print(input, " < SECRET_NUMBER");
16             println("");
17         }
18
19         print("(", i, ") - " , "Entrez une valeur: ");
20         input = readInt();
21         i = i + 1;
22     }
23
24     println("Bravo vous avez trouve le nombre correct!");
25 }

```

On y retrouve la définition des variables :

```
@
@ =====
@ == Global variables ==
@ =====
@
.section .data
@ Variables declarations:
var_secret_number:
.word 0x00000438
var_i:
.word 0x00000001
var_input:
.word 0x00000000
```

Puis on retrouve le code principal :

```

@
@ =====
@ == Main ==
@ =====
@
.globl main
main:
@ Main instructions:
    LDR r0, =str_KA__
    BL write_str
    LDR r0, =var_i
    LDR r0, [r0]
    BL write_int
    LDR r0, =str_KSAAtIA__
    BL write_str
    LDR r0, =str_RW50cmV6IHVuZSB2YWxldXI6IA__
    BL write_str
    BL read_int
    LDR r1, =var_input
    STR r0, [r1]
    LDR r0, =0x00000001
    LDR r1, =var_i
    LDR r1, [r1]
    ADD r0, r1, r0
    LDR r1, =var_i
    STR r0, [r1]
while_start_10_1:
    LDR r0, =var_secret_number

```

5. Méthode de validation

a. Tests

De la même manière que dans la section principale du projet, nous avons élaboré des tests unitaires afin de détecter les erreurs de manière exhaustive. Ces tests ont été conçus pour simplifier au maximum la détection d'anomalies. En plus des tests unitaires, nous avons également mis en place des tests couplés, impliquant plusieurs opérations simultanées. Ces derniers ont permis d'évaluer le compilateur dans sa globalité, offrant ainsi une vision plus holistique de son fonctionnement. Il est important de souligner que la batterie de tests initiale, développée pour la partie principale, n'a pas subi de modifications substantielles lors de l'intégration de l'extension ARM. Cette approche a assuré une continuité dans l'évaluation de la robustesse et de la fiabilité du compilateur, tout en préservant la validité des tests déjà en place.

b. Intégration avec le compilateur


















L'intégration de cette extension ARM a suivi une approche similaire à celle employée pour le compilateur initial. Nous avons entrepris cette démarche en naviguant à travers l'arbre décoré par la partie contextuelle du compilateur, puis en procédant à la génération du code ARM correspondant. Cette méthodologie familière s'est avérée être un moyen efficace de garantir la cohérence entre les différentes phases du processus de compilation, tout en assurant une adéquation précise avec les spécifications requises pour l'architecture ARM. Ce processus rigoureux a contribué à l'interopérabilité fluide de l'extension ARM avec le compilateur existant, renforçant ainsi l'intégrité et la performance globale du système.

6. Résultats de la validation de l'extension

Les tests de couvertures sont avec l'extension :

 Deca Compiler

Decca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Mis
fr.ensimag.deca.syntax		74%		56%	4
fr.ensimag.deca.tree		85%		80%	1
fr.ensimag.deca		69%		67%	
fr.ensimag.arm.pseudocode		0%		0%	
fr.ensimag.deca.context		79%		66%	
fr.ensimag.ima.pseudocode		80%		83%	
fr.ensimag.ima.pseudocode.instructions		79%		n/a	
fr.ensimag.deca.codegen		92%		75%	
fr.ensimag.deca.tools		98%		87%	
Total	5,151 of 24,203	78%	417 of 1,262	66%	8

Avec l'extension on obtient ces résultats en termes de couverture de tests. Comme dit dans l'autre partie n'est pas optimale, cependant elle reste honorable.