Documentation de l'extension ARM pour la langage Deca

Baptiste Le Duc Mathéo Dupiat Malo Nicolas
Ryan El Aroud Théo Giozovanni

21 janvier 2025

Table des matières

Introduction **1**

Spécification de l'extension ARM **2**
Définition de la cible 2
Pourquoi le Cortex-A53 ? 3
Objectifs : 4
Défis techniques 5
Plan d'action 5
Méthode de validation 6
Mise en place de l'environnement 6

Introduction

Depuis une vingtaine d'années, l'informatique est omniprésente dans tous les aspects de notre société, que ce soit dans le domaine médical, l'industrie, l'enseignement ou encore les services publics. Cette révolution numérique, qualifiée d'innovation de rupture, a été rendue possible grâce à des investissements conséquents et a conduit à une augmentation exponentielle de la puissance de calcul des ordinateurs et de la complexité du matériel informatique.

Au cœur de cette évolution, la loi de Moore, formulée par Gordon Moore en 1965, a joué un rôle clé en prévoyant un doublement des transistors dans les processeurs tous les deux ans, à coût constant. Bien que ses effets s'atténuent aujourd'hui, elle a façonné les progrès des architectures processeur, notamment ARM.

Les processeurs ARM (Advanced RISC Machines) se sont imposés comme une référence, notamment dans les systèmes embarqués et les appareils mobiles, mais aussi, plus récemment, dans les ordinateurs personnels comme ceux

d'Apple. Leur architecture RISC (Reduced Instruction Set Computing) est conçue pour maximiser l'efficacité énergétique tout en offrant des performances adaptées à une large gamme d'applications. Leur finesse de gravure illustre les progrès des semi-conducteurs et la Loi de Morre : de 180 nm dans les années 2000 à 3 nm aujourd'hui pour les Cortex-X925 ou Cortex-A520.

Désireux d'approfondir nos connaissances sur l'architecture ARM, omniprésente dans les téléphones mobiles et récemment adoptée par Apple pour ses ordinateurs, nous avons entrepris d'étendre notre compilateur pour le langage Deca afin de le rendre compatible avec cette architecture. Dès lors, nous pouvons nous demander : **comment adapter efficacement notre compilateur pour exploiter les spécificités de l'architecture ARM tout en respectant des contraintes de temps limité ?**

Ce document se propose d'exposer notre démarche en quatres étapes principales : une spécification détaillée de l'extensio avec une analyse bibliographique des architectures ARM, la présentation de nos choix de conception, d'architecture et d'algorithmes, une description de la méthode de validation mise en œuvre, et enfin, les résultats obtenus lors de la validation de l'extension.

Spécification de l'extension ARM

Définition de la cible

Le choix du processeur cible pour notre compilateur a été un difficile, car il devait à la fois s'intégrer facilement dans notre environnement de développement, notamment via des outils comme QEMU, tout en répondant à des critères techniques précis. Ce choix s'est avéré complexe en raison de la diversité des processeurs disponibles, chacun ayant ses propres spécificités. Les contraintes que nous nous avions fixé était alors les suivantes :

- Possibilité de simuler la cible facilement via Qemu
- Accessibilité de la documentation
- Popularité du processeur

En explorant les différentes options, nous avons approfondi nos connaissances sur QEMU, un émulateur de processeur offrant un large éventail d'architectures. Parmi les processeurs ARM, deux grandes catégories se distinguent :

- **Les processeurs A-profile** : équipés d'une MMU *(Memory Management Unit)*, ils peuvent exécuter des systèmes d'exploitation complets comme Linux et permettent l'utilisation de syscalls.
- **Les processeurs M-profile** (ex. Cortex-M0, Cortex-M4) : dits *"bare-metal"*, ils ne disposent pas de MMU et sont conçus pour des environnements sans système d'exploitation, rendant leur utilisation plus complexe dans notre contexte.

Compte tenu de notre besoin de faciliter l'émulation et d'exécuter des

systèmes complets, nous avons opté pour les processeurs A-profile. De plus, nous souhaitions travailler avec une architecture moderne afin d'avoir un aperçu concret des standards actuels de l'industrie. Cela nous a conduits à choisir une architecture 64 bits.

Ainsi, de part ses contraintes, nous avions le choix entre 3 processeur finaux tous utilisant l'architecture ARMv8-A qui est la première architecture ARM en 64-bits :

- Cortex-A53
- Cortex-A57
- Cortex-A72

Notre choix final s'est porté sur le **Cortex-A53**.

Pourquoi le Cortex-A53 ?

Le **Cortex-A53** est un microprocesseur 64 bits, gravé en 13 nm et lancé en 2014 par ARM. Initialement conçu pour les smartphones milieu de gamme, il a été adopté plus tard pour les plateformes IoT, grâce à ses performances et son efficience énergétique. Nous l'avons choisi comme cible matérielle pour plusieurs raisons :

1. **Retrocompatibilité** avec l'architecture ARMv7-A 32-bits
2. **Efficience énergétique** : Lors de son lancement, le Cortex-A53 était l'un des processeurs ARM les plus économes en énergie, ce qui en fait un choix idéal pour des applications nécessitant une faible consommation.
3. **Fonctionnement en big.LITTLE** : Le Cortex-A53 peut être couplé à un processeur plus puissant et énergivore (Cortex-A57) grâce à la technologie big.LITTLE. Ce mode de fonctionnement optimise la consommation énergétique en répartissant les tâches entre un processeur économe (LITTLE) et un processeur performant (big). Les deux processeurs étant émulables via QEMU, cela permet de tester des configurations proches des standards industriels, tout en intégrant l'importance des économies d'énergie, un enjeu majeur dans notre projet.

Pour mieux comprendre le fonctionnement de la technologie **big.LITTLE**, il est important d'examiner comment les tâches sont réparties entre les processeurs. Cela dépend largement de la manière dont le scheduler est implémenté dans le noyau du système d'exploitation. Dans ce cadre, il nous semblait intéressant de présenter un exemple de mécanisme de répartition des tâches.

La méthode la plus simple pour gérer le changement entre les cœurs dans une architecture big.LITTLE est appelée changement de cluster (clustered switching). Cette approche regroupe les cœurs en deux clusters distincts de taille identique : un cluster "big" pour les cœurs puissants et un cluster "LITTLE" pour les cœurs économes en énergie.
Dans cette configuration, le scheduler ne voit qu'un seul cluster actif à la fois.

Lorsque la charge de travail (load), qui mesure le nombre d'opérations computationnelles demandées par le système d'exploitation, dépasse un certain seuil (haut ou bas), le scheduler bascule entre les deux clusters.

Pour assurer la continuité des données et éviter des pertes, celles-ci transitent via un cache de niveau 2 (L2 cache) partagé entre les clusters. Une fois le transfert terminé, le cluster actif est désactivé pour économiser de l'énergie, tandis que l'autre cluster est activé pour prendre en charge les nouvelles tâches.

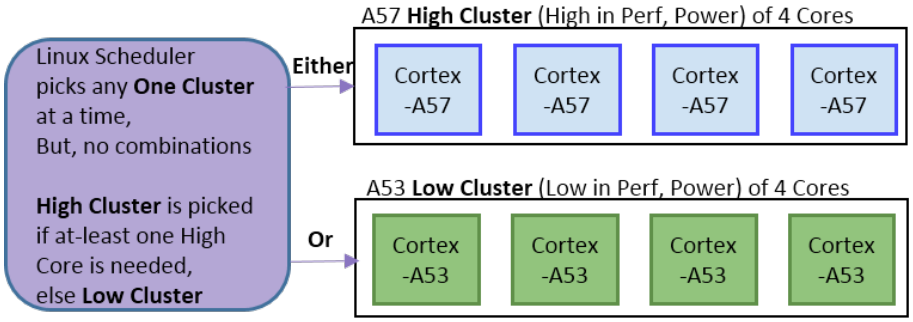

FIG. 1 : Changement de Cluster

Objectifs :

Au début du projet GL, nous nous sommes fixé un objectif SMART (Spécifique, Mesurable, Atteignable, Réaliste et Temporel) : réaliser l'extension ARM du compilateur pour la partie sans objet du langage Deca. Ce choix, justifié lors des réunions de suivi, reflétait notre priorité de créer un compilateur Deca respectant au mieux la spécification initiale du langage. Conscients que cette tâche nécessiterait un investissement de temps considérable, nous avons choisi de concentrer nos efforts sur cet objectif précis afin de garantir une base solide et conforme avant d'envisager l'extension.

Cet objectif sera alors considéré comme atteints si il remplit cette spécification :

1. Gestion des opération arithmétique (addition, soustraction, division, multiplication et modulo) sur les **entiers** et les **flottants** (via un cast implicit).
2. Gestion des opérations booléene :
   - Noeuds If-then-else
   - Opérateur de comparaison `(=, !=, >=, >, <=, <)`
   - And `(&&)`
   - Or `(||)`
3. Support du print :
   - printf
   - println

- printx
- printlnx

Défis techniques

1. **Double back-end** :
   - Nécessiter d'adapter le compilateur pour qu'il puisse produire du code pour deux cibles distinctes : la Machine Abstraite (IMA) et l'ARM Cortex-A53.
   - Concevoir l'architecture de manière à maximiser la réutilisation du code entre les deux cibles tout en permettant les optimisations spécifiques à ARM, afin de garantir un développement maintenable.
2. **Environnement d'exécution limité** :
   - Absence de librairies standard ARM pour les entrées/sorties et le débogage.
   - Conception et intégration d'un environnement minimaliste pour exécuter les programmes Deca sur ARM (rendu possible via Qemu)

Plan d'action

Afin de parvenir à l'objectif fixé, nous nous sommes organisés dès le début du projet afin d'organiser notre temps et nos développement. Nous avons alors établis ces différents étapes :

1. **Spécification et Analyse préliminaire** :
   - Choix de la cible
   - Étude de la la sémantique des instructions ARM et leur correspondance avec les instructions de Deca.
2. **Mise en place de l'environnement** :
   - Configurer un simulateur ou un émulateur ARM, comme **QEMU**, pour permettre l'exécution et le débogage des programmes Deca générés.
   - Finir la chaine de compilation via une étape d'assemblage et de linkage.
   - Automatiser les tests ARM à l'aide d'un pipeline CI/CD :
     - Configurer des scripts pour assembler, lier et exécuter automatiquement les programmes générés.
     - Intégrer la validation des programmes via le simulateur.
3. **Conception du back-end** :
   - Implémentation des classes nécessaires dans le package `fr.ensimag.deca.codegen` pour ARM.
   - Facteur commun avec le back-end IMA pour maximiser la réutilisation via des méthodes `codeGenInstARM()` semblable aux méthodes `codeGenInst()` de l'architecture initiale
4. **Optimisation** (si temps disponible) :
   - Ajout d'optimisations spécifiques à ARM (ex. utilisation efficace des registres, instruction "multiply-accumulate").

Méthode de validation

1. **Tests unitaires** :
   - Génération de code ARM pour une batterie de programmes Deca créés tout au long du projet pour IMA.
   - Vérification de la cohérence entre les résultats obtenus sur ARM et sur IMA.
2. **Évaluation de la performance** :
   - Mesure de la consommation en cycles ou via `power-top` pour des programmes réels exécutés sur ARM.
3. **Comparaison avec des compilateurs existants** :
   - Comparer les performances du code généré par Deca à celles obtenues par GCC sur des tâches similaires.

Mise en place de l'environnement

Pour développer et tester l'extension ARM, nous utiliserons les outils suivants :

1. **Cross-compilation avec aarch64-linux-gnu-gcc** :
   - Le compilateur `arm-linux-gnueabihf-gcc` permet, grâce à son outil d'assemblage et de linkage, de générer du code binaire pour l'architecture ARMv8-A (64 bits).
   - Installation sur une distribution Linux (ex. Ubuntu) :
     `sudo apt install gcc-aarch64-linux-gnu`
   - Assemblage et linkage pour produire un exécutable à partir d'un fichier assembleur (.s) en entrée :
     `aarch64-linux-gnu-gcc -o output_file_name input_file_name.s`
2. **Installation de l'environnement d'execution QEMU** :
   - QEMU est un émulateur et virtualiseur qui permet d'exécuter des programmes compilés pour ARM sur une machine 64 bits munie de notre processeur cible : **cortex-a53**.
   - Installation de QEMU :
     `sudo apt-get install qemu-user`
   - Exécution d'un programme ARM avec QEMU :
     `qemu-aarch64 -L aarch64-linux-gnu -cpu cortex-a53 ./output_file_name`
     L'option `-L` spécifie le chemin vers les bibliothèques dynamiques nécessaires à l'exécution du programme. Dans cet exemple, `aarch64-linux-gnu` est le répertoire contenant les bibliothèques pour l'architecture ARM.