Extension ARM

Auteurs :
Mehdi El Idrissi El Fatmi
Khadija El Adnani
Alaa Jennine
Badr El Khoundafi
Yassine El Kouri

Contents

Extension-ARM

GL02

January 2025

1 Introduction

Dans le cadre du projet d'implémentation du compilateur pour le langage Deca, nous avons choisi de développer une extension visant à générer un code assembleur optimisé pour l'architecture ARMv7 32 bits. Cette architecture, largement adoptée dans le domaine des systèmes embarqués, offre un équilibre entre puissance de calcul et efficacité énergétique.

Notre objectif principal est de permettre au compilateur Deca de produire un code assembleur exécutable sur des processeurs conformes à l'architecture ARMv7, tout en respectant les contraintes et les spécificités de cette architecture. En développant cette extension, nous visons également à explorer les mécanismes d'optimisation des performances et de gestion des ressources, en alignant nos choix de conception sur les meilleures pratiques de développement pour les systèmes embarqués.

Cette documentation détaillée présente les aspects fondamentaux de cette extension, en abordant notamment la spécification de l'extension, une analyse bibliographique des travaux précédents, nos choix de conception, ainsi que les méthodes et les résultats de validation.

2 Spécification de l'extension ARMv7 pour le compilateur Deca

Extension ARMv7 32 bits pour le compilateur Deca

L'extension ARMv7 32 bits pour le compilateur Deca vise à produire un code assembleur optimisé pour les processeurs basés sur l'architecture ARMv7. Cette architecture, largement utilisée dans les systèmes embarqués, se distingue par son jeu d'instructions 32 bits, conçu pour offrir un compromis idéal entre performances élevées et efficacité énergétique.

L'objectif principal de cette extension est d'exploiter les spécificités de l'ARMv7, telles que la gestion efficace des registres, le traitement des opérations en virgule flottante, et l'utilisation de la pile, afin de générer un code exécutable répondant aux contraintes de cette architecture. En intégrant ces capacités au

compilateur Deca, cette extension garantit une production de code performante et adaptée aux besoins des systèmes embarqués, tout en respectant les exigences d'optimisation en matière de ressources et d'énergie.

2.1 Compatibilité avec ARMv7 et support des instructions

Le compilateur Deca, enrichi de l'extension ARMv7 32 bits, est conçu pour générer du code pleinement compatible avec les processeurs respectant la spécification ARMv7. Cette compatibilité repose sur le support des instructions fondamentales du jeu ARM 32 bits, incluant principalement les instructions de type `R` et `I`, qui jouent un rôle central dans la gestion des opérations sur les registres et des calculs immédiats.

Les instructions de type `R` permettent d'effectuer des opérations directement entre registres, comme les calculs arithmétiques (addition, soustraction, multiplication) ou les manipulations logiques (ET, OU, XOR). Elles exploitent les registres internes du processeur pour maximiser la vitesse d'exécution et minimiser les accès à la mémoire, souvent plus coûteux en termes de cycles d'horloge.

Les instructions de type `I`, quant à elles, sont essentielles pour effectuer des opérations immédiates en manipulant directement des constantes intégrées dans l'instruction. Cela permet de réduire la complexité du code généré en évitant des chargements intermédiaires de données depuis la mémoire.

L'extension ARMv7 pour le compilateur Deca veille à optimiser ces instructions en fonction des capacités du processeur cible. Par exemple, une même opération complexe peut être traduite en un minimum d'instructions tout en exploitant les ressources matérielles spécifiques, telles que les unités de calcul en virgule flottante ou les registres dédiés. Cela garantit une exécution rapide et efficace des programmes, tout en réduisant les besoins en énergie et en minimisant les cycles d'instructions nécessaires.

En résumé, le support complet des instructions ARMv7 et leur optimisation permettent de tirer parti des performances et de l'efficacité de cette architecture, rendant le code généré par le compilateur Deca adapté aux contraintes et aux exigences des systèmes embarqués modernes.

2.2 Gestion des registres et conventions d'appel

L'architecture ARMv7 met à disposition 16 registres généraux, nommés de `R0` à `R15`, dont l'utilisation doit être soigneusement optimisée pour garantir une exécution efficace des programmes. Parmi eux, les registres `R0` à `R12` sont utilisés pour des opérations générales, tels que le stockage temporaire de données et les calculs, tandis que les registres `R13` (Stack Pointer - SP), `R14` (Link Register - LR) et `R15` (Program Counter - PC) remplissent des fonctions spécifiques liées à la gestion de la pile, des appels de fonctions et du contrôle du flux d'exécution.

Le registre `R15` (Program Counter, ou PC) est particulièrement crucial, car il contient l'adresse de la prochaine instruction à exécuter. Sa gestion exige une attention minutieuse, notamment lors des opérations de branchement (`BL`, `B`)

et des retours de fonction (`BX LR`). Une mauvaise manipulation du `PC` pourrait entraîner des comportements imprévisibles ou des plantages du programme. Lors d'un appel de fonction, le registre `R14` (Link Register, ou LR) joue un rôle clé en stockant l'adresse de retour, permettant ainsi au programme de reprendre son exécution normale une fois la fonction terminée. Quant au `R13` (Stack Pointer, ou SP), il est utilisé pour gérer la pile d'exécution, où sont stockées des données temporaires telles que les variables locales et les adresses de retour.

En complément, l'ARMv7 dispose également d'un registre de statut appelé `xPSR` (Program Status Register étendu). Ce registre regroupe plusieurs informations essentielles pour le fonctionnement du processeur :

- **N (Negative)** : Indique si le résultat de la dernière opération est négatif. - **Z (Zero)** : Indique si le résultat de la dernière opération est égal à zéro. - **C (Carry)** : Représente le dépassement (carry-out) ou l'emprunt (borrow) lors d'opérations arithmétiques. - **V (Overflow)** : Signale un dépassement arithmétique dans des calculs signés. - **T (Thumb state)** : Définit si le processeur est en mode Thumb (16 bits) ou ARM (32 bits). - **Bits exceptionnels** : Certains bits du `xPSR` sont utilisés pour identifier et gérer des exceptions, telles que les interruptions matérielles et les appels système (`SVC`).

La gestion correcte du `xPSR` est essentielle pour prendre des décisions conditionnelles (par exemple, lors de l'exécution d'instructions conditionnelles telles que `BEQ` ou `BNE`) et pour assurer un comportement correct lors des interruptions.

Les conventions d'appel d'ARMv7, qui régissent l'utilisation des registres dans le cadre des fonctions, sont également fondamentales. Ces conventions dictent :

- **Le passage des paramètres** : Les quatre premiers paramètres d'une fonction sont placés dans les registres `R0` à `R3`. Si la fonction nécessite plus de quatre paramètres, les valeurs supplémentaires sont stockées sur la pile. - **Le retour des valeurs** : Les résultats des fonctions sont renvoyés dans `R0` (ou `R1` si nécessaire). - **La sauvegarde des registres** : Les registres `R4` à `R11` sont considérés comme des registres "sauvés" (callee-saved), ce qui signifie que toute fonction les modifiant doit restaurer leurs valeurs initiales avant de revenir au programme appelant.

Ainsi, l'extension ARMv7 du compilateur Deca doit non seulement garantir une utilisation efficace de ces registres, mais également respecter ces conventions pour assurer une compatibilité et une robustesse maximales du code généré.

2.3 Opérations de base et structures de contrôle

Le compilateur Deca doit générer du code pour un large éventail d'opérations de base, en tenant compte des spécificités de l'architecture ARMv7 afin d'assurer à la fois performance et fiabilité. Ces opérations incluent les opérations arithmétiques telles que l'addition (`ADD`) et la soustraction (`SUB`), ainsi que les opérations logiques comme le ET logique (`AND`), le OU logique (`ORR`) et l'inversion (`MVN`). Ces instructions forment le cœur de nombreux programmes et doivent être traduites de manière optimale, en utilisant un nombre minimal d'instructions tout en préservant la précision et la cohérence des calculs. Par exemple, une addition

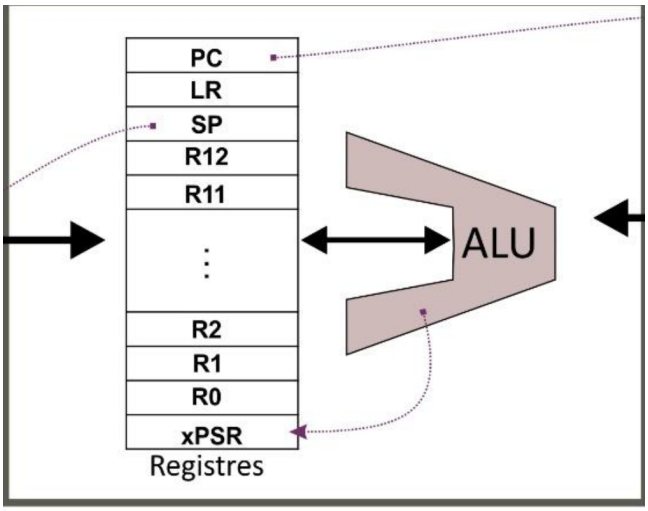

Figure 1: Gestion des registres et conventions d'appel.

complexe impliquant plusieurs registres peut être optimisée pour minimiser les redondances, en évitant les recalculs inutiles ou les accès fréquents à la mémoire.

Pour les opérations de chargement et de stockage (`LDR`, `STR`), le compilateur doit générer du code efficace pour manipuler la mémoire. Ces instructions sont essentielles pour transférer des données entre les registres et la mémoire, un aspect crucial pour les programmes nécessitant de fréquents accès à des variables ou des tableaux. Cependant, comme les accès mémoire sont plus coûteux en cycles processeur que les opérations sur registres, il est impératif de minimiser ces accès. Par exemple, dans une boucle répétitive, les valeurs fréquemment utilisées devraient être chargées dans des registres au lieu d'être relues depuis la mémoire à chaque itération. Cette stratégie d'optimisation réduit considérablement la latence et améliore les performances globales, en particulier dans les systèmes embarqués où les ressources sont limitées.

Les structures de contrôle, telles que les boucles (`while`) et les conditions (`if-else`), sont également fondamentales dans tout programme. Le compilateur doit générer les instructions de branchement nécessaires pour modéliser ces structures de manière efficace. Par exemple :

- **Conditions** : Les blocs conditionnels (`if-else`) doivent être traduits en tests conditionnels et branchements conditionnels (`BGE`, `BLT`). Ces branchements, basés sur les drapeaux du registre de statut (`xPSR`), permettent de sauter à des parties spécifiques du code selon le résultat des comparaisons.

Les instructions de branchement, telles que `B`, `BL` (appel de sous-routine) et `BX` (retour d'une fonction), sont également utilisées pour gérer le flux du programme. Le compilateur doit optimiser ces branchements en réduisant la distance des sauts et en minimisant les cycles inutiles causés par des instructions de branchement redondantes ou mal alignées. Par exemple, dans les boucles

imbriquées, il peut être avantageux de reconfigurer les sauts pour diminuer la latence et améliorer la lisibilité du code généré.

En résumé, le compilateur Deca doit non seulement gérer la génération d'instructions pour les opérations de base, mais également s'assurer que les structures de contrôle sont traduites en un code ARMv7 compact et efficace. Cette optimisation est cruciale pour obtenir des performances optimales, en particulier dans des contextes contraints comme les systèmes embarqués, où la puissance de calcul et la consommation d'énergie sont limitées. La capacité à minimiser les accès mémoire, à réduire les cycles processeur nécessaires pour les branchements, et à optimiser les boucles et les conditions garantit un code final à la fois rapide, fiable et adapté aux exigences des processeurs ARMv7.

2.4 Optimisation des performances

Une des priorités majeures de cette extension est l'optimisation des performances du code généré. Le compilateur doit minimiser le nombre d'instructions générées tout en maintenant la correction fonctionnelle. Cela peut être réalisé en utilisant des techniques d'optimisation telles que l'élimination de code mort, qui consiste à supprimer les instructions qui n'affectent pas le résultat final du programme, et la fusion d'instructions similaires pour réduire le nombre total d'instructions.

La réduction de l'utilisation de la mémoire et des cycles processeur est également un objectif clé. En optimisant l'allocation de mémoire et en gérant les registres de manière plus efficace, il est possible de réduire la pression sur la mémoire et de garantir une exécution plus rapide du programme. Cela est particulièrement important dans les systèmes embarqués, où les ressources sont souvent limitées.

2.5 Intégration avec le compilateur Deca

Enfin, cette extension doit s'intégrer de manière fluide avec le reste du compilateur Deca, garantissant une cohérence entre toutes les phases du processus de compilation. L'une des étapes fondamentales dans la chaîne de compilation est la génération d'un arbre abstrait décoré. Cet arbre constitue une représentation intermédiaire enrichie des informations nécessaires pour produire un code machine correct et optimisé. Avec l'intégration de l'extension ARMv7, cet arbre doit être adapté pour inclure des détails spécifiques à l'architecture cible, comme les registres à utiliser, les conventions d'appel, et les optimisations applicables aux instructions ARM.

Le compilateur doit ensuite transformer cet arbre abstrait décoré en code assembleur conforme aux standards ARMv7. Cette étape nécessite une traduction précise des opérations définies dans l'arbre vers des instructions du jeu d'instructions ARMv7. Par exemple, une opération arithmétique comme une addition ou une soustraction devra être traduite en une séquence minimale d'instructions ARM (`ADD`, `SUB`), en tenant compte des contraintes de taille et de performance. De plus, pour des structures complexes telles que les boucles

ou les branchements conditionnels, le code assembleur généré devra inclure des instructions adaptées (`B`, `BL`, `CMP`) qui respectent les mécanismes de contrôle de flux de l'architecture ARM.

L'extension ARMv7 doit également veiller à générer des fichiers assembleurs compatibles avec les assembleurs standard pour cette architecture, tels que GNU Assembler (GAS). Cela implique de respecter les conventions de syntaxe et les directives spécifiques à ARM, comme l'utilisation correcte des sections de texte (`.text`), de données (`.data`) et des directives de marquage (`.global`, `.align`, etc.). Cette compatibilité garantit que le code produit peut être directement assemblé en un exécutable fonctionnel pour les processeurs ARMv7.

Une attention particulière doit être accordée à l'optimisation du code généré pour les systèmes embarqués. Ces systèmes, souvent limités en termes de ressources, exigent un code compact et performant. L'extension ARMv7 doit tirer parti des capacités spécifiques de l'architecture pour réduire la consommation énergétique et maximiser l'efficacité. Par exemple, l'utilisation judicieuse des instructions de type `MOV` pour minimiser les transferts de données inutiles, ou encore l'exploitation des instructions conditionnelles (`IT`, `CMP`) pour éviter des branchements coûteux, est essentielle. De plus, les accès à la mémoire doivent être réduits au strict nécessaire, et les registres disponibles (`R0` à `R12`) doivent être exploités au maximum pour limiter les cycles processeur.

Une autre dimension de cette intégration concerne la fluidité entre l'extension ARMv7 et les autres phases du compilateur Deca. Par exemple, les optimisations globales effectuées au niveau de l'arbre abstrait décoré (comme l'élimination de code mort ou la simplification des expressions) doivent être compatibles avec les contraintes de génération de code ARM. Cela garantit que les optimisations restent efficaces sans compromettre la validité du code final.

Enfin, l'extension doit également inclure des mécanismes robustes de vérification et de gestion des erreurs. Lors de la traduction en assembleur, le compilateur doit s'assurer que les instructions générées respectent les limites de l'architecture ARMv7, comme le nombre de registres disponibles ou les contraintes des instructions immédiates (valeurs encodables directement dans une instruction). En cas d'incompatibilité, le compilateur doit fournir des messages d'erreur clairs pour aider les développeurs à identifier et corriger les problèmes.

En somme, l'extension ARMv7 pour le compilateur Deca vise à offrir une solution complète et optimisée pour la génération de code sur les systèmes embarqués basés sur l'architecture ARMv7. En combinant une gestion efficace des registres, un support complet des instructions de base et une prise en charge approfondie des optimisations spécifiques, cette extension garantit une génération de code performante, compacte et adaptée aux contraintes des systèmes embarqués. Grâce à une intégration fluide avec le reste du compilateur Deca, elle permet de produire un code conforme aux standards de l'architecture ARM tout en assurant une exécution fiable et rapide sur les processeurs compatibles ARMv7.

3 Analyse Bibliographique sur l'Architecture ARM

L'architecture ARM, développée par ARM Holdings (aujourd'hui une filiale de SoftBank), est une architecture de processeur largement utilisée dans les systèmes embarqués, les appareils mobiles et d'autres domaines nécessitant une faible consommation d'énergie tout en maintenant des performances élevées. L'architecture ARM se distingue par son faible coût, sa faible consommation d'énergie et sa flexibilité, ce qui la rend particulièrement adaptée aux applications mobiles et aux systèmes embarqués.

Les processeurs ARM sont utilisés dans une variété d'appareils allant des smartphones aux objets connectés, en passant par les tablettes et les systèmes embarqués. ARMv7, en particulier, est une version 32 bits de l'architecture qui a été largement utilisée dans ces applications jusqu'à la transition vers ARMv8 (64 bits).

3.1 Caractéristiques Techniques de l'Architecture ARMv7

L'architecture ARMv7 32 bits se caractérise par un jeu d'instructions RISC (Reduced Instruction Set Computing), où les instructions sont simples et de longueur fixe, permettant une exécution rapide et une gestion efficace des ressources.

- **Jeu d'Instructions** : L'ARMv7 supporte un large éventail d'instructions, notamment des instructions arithmétiques et logiques de base (addition, soustraction, multiplication), des instructions de contrôle de flux (sauts conditionnels et inconditionnels), et des instructions de manipulation de la mémoire (chargement et stockage).

- **Registres** : ARMv7 dispose de 16 registres généraux de 32 bits (R0 à R15), avec des rôles spécifiques attribués à certains registres, comme le pointeur de pile (R13), le registre de lien pour les appels de fonction (R14), et le compteur de programme (R15). Ces registres doivent être utilisés de manière efficace pour maximiser les performances et minimiser les accès à la mémoire.

- **Modes de Fonctionnement** : ARMv7 supporte différents modes de fonctionnement, comme le mode utilisateur (privileges limités) et le mode superviseur (accès complet). Ces modes permettent une gestion des exceptions et des interruptions efficace, ce qui est crucial pour le bon fonctionnement des systèmes embarqués.

- **Optimisations pour l'Embarqué** : ARMv7 est conçu pour minimiser la consommation d'énergie, ce qui est essentiel dans les systèmes embarqués où les ressources sont limitées. La gestion fine des registres et la capacité à exécuter plusieurs instructions par cycle d'horloge permettent d'obtenir de bonnes performances tout en optimisant la consommation d'énergie.

Optimisation des Compilateurs pour ARMv7

L'optimisation du code pour ARMv7 est une tâche essentielle dans le développement de logiciels pour cette architecture, notamment en raison des contraintes de ressources souvent présentes dans les systèmes embarqués. Plusieurs techniques sont couramment utilisées pour optimiser le code produit par les compilateurs ARM :

- **Optimisation du Jeu d'Instructions** : Les compilateurs doivent générer des séquences d'instructions qui tirent parti des instructions ARMv7 les plus efficaces, comme les instructions de type `R` pour les opérations sur les registres et les instructions `I` pour les opérations immédiates. Les optimisations peuvent réduire la taille du code généré et minimiser le nombre d'instructions nécessaires pour exécuter une tâche donnée.
- **Allocation des Registres** : L'allocation de registres est un aspect clé de l'optimisation pour ARMv7. Les compilateurs doivent s'assurer que les registres sont utilisés de manière optimale, en réduisant au minimum les accès mémoire, qui sont plus coûteux en termes de performances. Le processus d'allocation doit respecter les conventions d'appel de l'architecture ARMv7, comme le passage des paramètres dans les registres et la gestion des retours de fonction.
- **Elimination de Code Mort** : Il s'agit d'une technique d'optimisation qui consiste à supprimer les instructions qui n'ont pas d'impact sur le résultat final du programme. Cela permet de réduire la taille du code et d'améliorer l'efficacité du programme.
- **Fusion d'Instructions** : Certains compilateurs optimisent le code en fusionnant des instructions similaires pour réduire le nombre total d'instructions. Par exemple, les compilateurs peuvent fusionner plusieurs instructions de calcul en une seule instruction qui effectue les mêmes opérations de manière plus efficace.

Travaux Précédents et Outils de Génération de Code pour ARM

Plusieurs travaux de recherche et outils ont été développés pour améliorer la génération de code pour ARM. Ces travaux portent principalement sur l'optimisation des compilateurs et la gestion des ressources dans les systèmes embarqués.

- **Compilateurs et Outils de Génération de Code** : GCC (GNU Compiler Collection) est l'un des compilateurs les plus utilisés pour ARM. Il supporte l'architecture ARMv7 et propose des options spécifiques pour optimiser le code pour ARM, telles que `-O2` pour l'optimisation du code, ou `-mcpu=cortex-a8` pour spécifier l'optimisation pour un processeur ARM particulier. D'autres outils, comme Clang, sont également utilisés pour générer du code optimisé pour ARM.

- **Optimisation du Code pour ARMv7 dans les Systèmes Embarqués** : Des études telles que celles de Zhang et al. (2013) ont examiné l'optimisation du code pour ARMv7 dans les systèmes embarqués, en mettant l'accent sur les techniques permettant de réduire la taille du code et d'améliorer les performances d'exécution. Ces travaux ont montré que l'optimisation des accès mémoire et l'utilisation de techniques de prévision de branchement peuvent améliorer de manière significative les performances.

- **Compilateurs Spécifiques à ARM** : Certaines entreprises développent des compilateurs spécifiquement adaptés à l'architecture ARM, tels que le compilateur ARMCC développé par ARM Ltd. Ce compilateur propose des optimisations spécifiques à l'architecture ARM, notamment une meilleure gestion des registres et des optimisations pour les processeurs ARM modernes.

3.2 Comparaison avec d'autres Architectures

L'architecture ARMv7 se distingue d'autres architectures de processeurs, comme x86 et MIPS, par plusieurs aspects :

- **x86 vs ARM** : L'architecture x86, utilisée principalement dans les ordinateurs de bureau et les serveurs, est une architecture CISC (Complex Instruction Set Computing), contrairement à ARM qui est une architecture RISC. Cela signifie que les processeurs ARM ont tendance à avoir des instructions plus simples, ce qui les rend plus efficaces dans les systèmes embarqués. Cependant, x86 offre des performances supérieures pour des charges de travail plus lourdes, grâce à des instructions complexes et une gestion plus efficace des tâches multi-threadées.

- **MIPS vs ARM** : MIPS, également une architecture RISC, est une autre architecture couramment utilisée dans les systèmes embarqués. Cependant, ARM offre un meilleur rapport performance/consommation d'énergie et dispose d'un plus large écosystème de développement et de support matériel.

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

4.1 Choix de conception

Choix de conception et d'architecture

Dans le cadre de notre projet, nous avons porté une attention particulière aux choix de conception et d'architecture afin de garantir une extension efficace et adaptée à nos besoins. Ces choix ont été guidés par les contraintes du projet, les performances attendues, ainsi que par les spécificités de l'architecture ciblée.

4.2 Architecture cible

Nous avons choisi l'architecture ARMv7 en 32 bits en raison de sa large adoption dans les systèmes embarqués et de sa compatibilité avec une variété d'outils et de bibliothèques existants. Cette architecture offre un équilibre optimal entre complexité et performance, tout en étant bien documentée, ce qui a facilité son intégration dans notre projet.

L'intégration des instructions ARMv7 et VFP (Vector Floating Point) dans notre compilateur permet de tirer parti des capacités de calcul et d'optimisation propres à cette architecture.

Quelques instructions significatives que nous utilisons :

Instructions utilisées

Dans le cadre de ce projet, de nombreuses instructions ont été utilisées pour optimiser les performances et gérer efficacement les opérations au niveau bas. Voici une liste détaillée des instructions employées et leurs rôles :

- `BL` : Effectue un appel à une sous-routine, indispensable pour la gestion des fonctions.
- `SVC` : Génère un appel système, essentiel pour interagir avec le système d'exploitation.
- `VLDR` : Charge une donnée en mémoire dans un registre flottant.
- `VSTR` : Stocke une donnée d'un registre flottant vers la mémoire.
- `VMOV` : Permet des transferts entre registres ou entre un registre ARM et un registre VFP.
- `VADD` et `VSUB` : Effectuent respectivement des additions et des soustractions en virgule flottante.
- `VMUL` : Multiplie deux valeurs en virgule flottante.
- `VCMP` : Compare deux valeurs flottantes pour des opérations conditionnelles.
- `VCVT` : Convertit des données entre différents formats, comme entre simple et double précision, ou entre flottant et entier.
- `ADD` et `SUB` : Réalisent des opérations arithmétiques basiques, comme l'addition et la soustraction de registres.
- `MOV` et `MOV64` : Transfèrent des valeurs entre registres ou entre un registre et la mémoire.
- `MUL`, `VDIVF64`, et `VMULF64` : Effectuent des multiplications et des divisions, y compris sur des données en double précision.

- `NEG` : Inverse le signe d'une valeur.
- `LDR` et `STR` : Chargent et stockent des données entre les registres et la mémoire.
- `POP` et `PUSH` : Manipulent la pile, en sauvegardant et restaurant les registres lors des appels de fonction.
- `VADDRF64`, `VCVTF64S32`, `VNEGF64`, `VSTR`, et `VSUBF64` : Des instructions SIMD (Single Instruction, Multiple Data) utilisées pour le traitement parallèle des données flottantes en double précision.

Ces instructions ont été essentielles pour gérer les opérations de bas niveau, optimiser les performances et tester les fonctionnalités du compilateur dans différents contextes.

4.3 Choix de conception

Pour la gestion des instructions ARM, nous avons adopté une approche modulaire et orientée objet. Ce choix de conception repose sur la création de classes spécialisées, comme `ARMBinaryInstruction` et `ARMTernaryInstructionDValToReg`, qui permettent de structurer et de représenter efficacement les différentes instructions de l'architecture ARMv7.

- **Abstraction des instructions :** Les classes `ARMBinaryInstruction` et `ARMTernaryInstructionDValToReg` encapsulent les caractéristiques fondamentales des instructions binaires et ternaires respectivement. Cette abstraction facilite l'ajout ou la modification des instructions sans affecter la structure globale du code.
- **Affichage des instructions :** Les méthodes comme `displayOperands(PrintStream s)` permettent de gérer facilement la représentation des instructions, ce qui est essentiel pour générer un code assembleur lisible et conforme à l'architecture ARMv7.
- **Support des opérandes variés :** Les classes prennent en charge différents types d'opérandes, comme les registres (`ARMGPRegister`), les valeurs immédiates (`DVal`), et les chaînes (`ARMImmediateString`). Cela nous a permis d'étendre facilement les fonctionnalités pour inclure des instructions complexes nécessitant des opérandes mixtes.
- **Extensibilité :** La hiérarchie des classes, basée sur des classes abstraites comme `ARMInstruction`, est conçue pour supporter l'ajout futur d'instructions spécifiques ou d'optimisations ciblées. Par exemple, `ARMTernaryInstructionDValToReg` permet de manipuler trois opérandes tout en respectant les contraintes de l'architecture ARM.

- **Support des registres VFPv3 :** Pour tirer pleinement parti des capacités de calcul flottant, nous avons spécifié l'utilisation du compilateur avec l'option `-mfpu=vfpv3`. Cette option permet d'utiliser les registres flottants `S` et `D`, essentiels pour effectuer des opérations en virgule flottante avec des performances optimisées, conformément aux exigences de notre projet.

- **Gestion des registres ARM :** Afin de gérer efficacement les registres de l'architecture ARM, nous avons mis en place des méthodes spécifiques pour associer et vérifier la disponibilité des registres. Par exemple :

  - `associerRegARMD()` : Cette méthode assigne un registre flottant de type `D` (double précision) disponible parmi les registres ARM (index 2 à 15). Si tous les registres sont utilisés, elle appelle `associerRegOffsetARMD()` pour récupérer un registre à partir d'un offset spécifique.
  - `associerRegARMS()` : Cette méthode fonctionne de manière similaire pour les registres `S` (simple précision) de l'architecture VFPv3, en vérifiant la disponibilité des registres dans la plage de 2 à 31.
  - `associerRegARM()` : Associe un registre général de l'architecture ARM, dans la plage de 4 à `OverflowValARM+1`.
  - `hasAvailableregARM()` : Vérifie si un registre général ARM est disponible en parcourant la plage spécifiée de registres et retourne `true` si un registre est libre, sinon `false`.

  - **Association d'un registre pour le résultat :** Un registre de type `ARMGPRegister` ou `ARMDoubleRegister` est associé pour stocker le résultat de l'expression, en utilisant la méthode `compiler.associerRegARMD()`. Cela permet d'assurer une allocation efficace des registres pour les opérations en cours.
  - **Gestion des opérandes :** En fonction du type d'opérande généré par l'expression (`DAddr`, `ARMImmediateInteger`, `ARMImmediateFloat`, `ARMGPRegister`), différentes instructions sont ajoutées au compilateur. Les cas suivants sont pris en compte :
    * **Opérande de type `DAddr` :** L'adresse est chargée à l'aide d'une instruction `LDR`, puis la valeur à cette adresse est extraite. Ensuite :
      · Pour les entiers : aucune conversion n'est nécessaire, et la valeur est directement utilisée.
      · Pour les flottants : la valeur est déplacée vers le registre flottant `S0` avec `VMOV`, puis convertie en double précision dans le registre `D0` à l'aide de `VCVTF64S32`.
    * **Opérande immédiate entière (`ARMImmediateInteger`) :** Une instruction `VMOV` est utilisée pour déplacer la valeur immédiate dans `S0`, suivie d'une conversion en double précision dans `D0` avec `VCVTF64S32`.

    - **Opérande immédiate flottante (`ARMImmediateFloat`) :** La valeur immédiate est directement déplacée dans le registre flottant `D0` via `VMOV`.
    - **Opérande registre général (`ARMGPRegister`) :** La valeur du registre général est d'abord transférée dans `S0` à l'aide de `VMOV`, puis convertie en double précision dans `D0`.
- **Stockage du résultat :** Le résultat final est déplacé dans le registre associé (`reg`) en utilisant l'instruction `VMOVF64`. Cette étape garantit que le registre contient le résultat approprié pour les opérations suivantes.

4.4 Gestion des expressions et des opérations ARM

Une partie essentielle de la conception du compilateur concerne la génération de code pour les expressions ARM. La méthode `codeGenExprARM()` permet de gérer les différents types d'expressions et de générer les instructions appropriées pour l'architecture ARM. Cet exemple montre comment gérer une expression, mais l'algorithme utilisé est similaire pour d'autres types d'opérations et d'expressions.

4.5 Génération de code pour les opérations arithmétiques et la division

La classe `codeGenARM` joue un rôle clé dans la génération du code ARM pour les différentes opérations arithmétiques. Les opérations telles que l'addition, la multiplication et la division suivent des schémas spécifiques, chacun ayant des particularités pour gérer les types de données (entiers ou flottants) et leur traitement au niveau de l'architecture ARM.

- Pour les opérations arithmétiques simples comme l'addition et la multiplication, la méthode `codeGenExprARM` commence par vérifier les types des opérandes. - Si les opérandes sont des entiers, la méthode `codeGenARM` est utilisée. Elle charge les opérandes dans des registres généraux ARM (comme `R0`, `R1`, etc.) et génère les instructions appropriées, telles que `ADD` pour l'addition ou `MUL` pour la multiplication. - Une fois l'opération effectuée, les registres sont libérés, assurant ainsi que les ressources sont utilisées de manière optimale et que l'état des registres ne persiste pas après l'opération.

- Cependant, pour les opérations impliquant des nombres à virgule flottante, la méthode `codeGenARMFloat` est utilisée. - Cette méthode commence par gérer les conversions nécessaires, par exemple, en utilisant l'instruction `VCVTF64S32` pour convertir un entier en flottant. - Les valeurs des opérandes flottants sont ensuite chargées dans les registres `D0` et `D1`, qui sont spécifiquement conçus pour traiter les nombres à virgule flottante sur ARM. - L'instruction ARM appropriée est alors générée, comme

`VADD` pour l'addition ou `VMUL` pour la multiplication, à l'aide de l'objet `ARMconstructeur`.

- Pour la division, le processus devient plus complexe en raison des particularités de l'architecture ARM. - Si au moins un des opérandes est un flottant, la méthode `codeGenARMFloat` est à nouveau utilisée. Cette fois, l'instruction `VDIV` est générée, permettant de réaliser une division flottante entre les deux opérandes dans les registres `D0` et `D1`. - En revanche, si les deux opérandes sont des entiers, la méthode adopte une approche différente. Les opérandes sont chargés dans des registres généraux ARM comme `R0` et `R1`. Si les opérandes sont stockés en mémoire, ils sont récupérés avant d'être chargés dans les registres. - L'instruction `BL` (Branch with Link) est alors utilisée pour appeler la routine standard de division entière ARM, `__aeabi_idiv`, qui effectue la division entière en utilisant une procédure spécifique de l'ABI (Application Binary Interface). - Une fois la division effectuée, le résultat est retourné dans le registre `R0` par la routine, et il est ensuite transféré dans le registre alloué pour le résultat, qui pourra être utilisé pour des calculs ultérieurs.

Cette gestion spécifique des opérations arithmétiques et de la division dans le générateur de code ARM reflète les particularités de l'architecture ARMv7-A, qui nécessite une attention particulière pour les flottants et les divisions entières, afin de maximiser l'efficacité et la compatibilité avec les standards de l'ABI ARM.

4.6 Génération de code ARM pour l'impression des résultats

La méthode `codeGenPrintARM` a pour objectif de générer le code ARM nécessaire pour effectuer l'opération d'impression d'une expression, et ce, en fonction du type des opérandes (flottant ou entier). Le processus inclut plusieurs étapes distinctes qui varient selon le type de donnée à imprimer. Voici un décryptage détaillé des différentes étapes de ce processus :

- **Traitement des opérandes flottants :** - Lorsqu'un opérande est un flottant, la méthode utilise des instructions de manipulation de registres flottants ARM pour gérer les opérations sur ces types de données. - Les registres dédiés aux flottants, tels que `D0`, `D1`, et `S0`, sont utilisés pour stocker et manipuler les valeurs flottantes. Ces registres sont optimisés pour les calculs en virgule flottante et sont manipulés avec des instructions spécifiques comme `VADD` (addition flottante), `VDIV` (division flottante), ou `VMOV` (transfert entre registres flottants). - Si les opérandes sont des entiers mais doivent être imprimés sous forme flottante, la méthode utilise l'instruction `VCVTF64S32` pour convertir les entiers en flottants (en 64 bits). Cette conversion est nécessaire pour garantir que les résultats soient affichés correctement sous le format `%f`. - Une fois le calcul effectué, le résultat est formaté pour l'impression en flottant. Le format `%f` est en-

suite passé à la fonction `printf` afin que le résultat soit affiché de manière lisible à l'écran.

- **Traitement des opérandes entiers :** - Lorsque les opérandes sont des entiers, la méthode passe à un traitement différent. En effet, les entiers sont manipulés via les registres généraux ARM, tels que `R0`, `R1`, etc. - Les instructions de base pour manipuler des entiers incluent des instructions telles que `ADD` (addition), `MOV` (déplacement de valeur), etc. Ces instructions sont suffisantes pour traiter les entiers, car l'architecture ARM possède des instructions spécifiques et très efficaces pour la gestion des registres généraux. - Si une division entière est nécessaire, la méthode utilise la fonction externe `__aeabi_idiv`, qui est optimisée pour effectuer des divisions entières. Cette fonction est appelée via l'instruction `BL` (Branch with Link), et elle retourne le résultat de la division dans le registre `R0`. - Une fois le résultat calculé, il est formaté pour l'impression en entier en utilisant le format `%d`. Ce format permet d'afficher le résultat sous forme d'un nombre entier standard.

- **Appel à la fonction `printf` :** - Une fois que le résultat des calculs (qu'ils soient flottants ou entiers) est prêt, la méthode génère le code nécessaire pour appeler la fonction standard `printf`, qui est utilisée pour afficher les résultats à l'écran. - La fonction `printf` prend en entrée une chaîne de format (soit `%f` pour les flottants, soit `%d` pour les entiers), suivie des registres contenant les valeurs calculées. Ce mécanisme permet de garantir que les résultats sont affichés correctement et dans le bon format.

En résumé, la méthode `codeGenPrintARM` génère du code ARM adapté aux types des opérandes des expressions arithmétiques. Elle choisit dynamiquement les bonnes instructions en fonction du type de l'opérande, que ce soit flottant ou entier, et génère le code nécessaire pour afficher les résultats correctement via la fonction `printf`. Ce processus garantit une sortie correcte et lisible pour l'utilisateur, qu'il s'agisse de résultats flottants ou entiers, tout en exploitant au mieux les instructions et les registres spécifiques de l'architecture ARM pour chaque type de donnée.

4.7 Génération de code ARM pour l'affectation

La méthode `codeGenExprARM` est responsable de la génération du code ARM nécessaire pour réaliser l'affectation d'une expression dans un contexte donné. Cette tâche implique plusieurs étapes distinctes, qui dépendent du type de la variable à affecter (entier ou flottant) et de l'opérande droit impliqué dans l'affectation. Voici une explication détaillée du processus :

- **Marquage de l'affectation :** - La méthode commence par marquer l'opération comme une affectation en définissant deux attributs dans le compilateur : `isAssign` et `typeAssign`. - L'attribut `isAssign` indique que l'on est dans un contexte d'affectation, tandis que `typeAssign` spécifie

le type de la variable à gauche de l'affectation (soit entier, soit flottant). Cela permet au compilateur de déterminer la nature de l'affectation à réaliser.

- **Génération du code pour l'opérande droit :** - Le code ARM est généré pour l'opérande droit de l'affectation en appelant la méthode `codeGenExprARM` sur cet opérande. Cela permet de générer les instructions appropriées pour évaluer l'opérande droit, en tenant compte de son type. - Pour l'opérande gauche, la méthode `getLeftOperand().codeGenExprARM(compiler)` est utilisée pour obtenir l'adresse mémoire où la valeur doit être stockée. Cela génère le code nécessaire pour accéder à l'emplacement mémoire de la variable gauche.

- **Traitement de l'opérande droit (registre général ARM) :** - Si l'opérande droit est un registre général ARM (`ARMGPRegister`), le code ARM utilise les instructions suivantes : - `LDR` (Load Register) : Cette instruction est utilisée pour charger l'adresse de la variable gauche dans le registre `R1`. - `STR` (Store Register) : Cette instruction permet de stocker la valeur contenue dans le registre de l'opérande droit à l'adresse mémoire pointée par `R1`. - Le type de l'opérande droit (entier ou flottant) détermine si `STR` est utilisée pour un entier ou `VSTR` pour un flottant.

- **Traitement de l'opérande droit (non registre général) :** - Si l'opérande droit n'est pas un registre général ARM, un registre temporaire est alloué pour stocker sa valeur. Ce registre est alloué par la méthode `compiler.associerRegARM()`. - Une fois le registre temporaire alloué, la valeur de l'opérande droit est chargée dans ce registre. Une instruction `MOV` (Move) est utilisée pour copier la valeur de l'opérande droit dans le registre temporaire. - Ensuite, l'adresse de l'opérande gauche est chargée dans `R1`, et les instructions `STR` ou `VSTR` sont utilisées pour effectuer l'affectation de la valeur stockée dans le registre temporaire à l'adresse mémoire de l'opérande gauche.

- **Libération des registres temporaires :** - Une fois l'affectation effectuée, le registre temporaire utilisé pour l'opérande droit est libéré. Cette opération est effectuée par la méthode `compiler.libererReg(reg.getNumber())`, ce qui permet de récupérer le registre pour d'autres usages dans le programme et ainsi optimiser l'utilisation des registres du processeur.

- **Optimisation de l'utilisation des registres :** - Le processus d'allocation et de libération des registres temporaires permet de maximiser l'efficacité du code généré, en évitant l'utilisation prolongée de registres inutiles. - Cela garantit également que les ressources matérielles du processeur sont utilisées de manière optimale, ce qui est particulièrement important pour les architectures avec un nombre limité de registres.

En résumé, la méthode `codeGenExprARM` génère du code ARM pour effectuer l'affectation d'une expression. Le code utilise des instructions telles que `LDR`, `MOV`, `STR` et `VSTR` en fonction du type des opérandes (entier ou flottant) et du type de l'opérande droit (registre général ou non). Le processus comprend également l'allocation et la libération de registres tem-

poraires pour garantir une gestion optimale des ressources matérielles du processeur. Ce mécanisme permet de réaliser des affectations efficaces tout en minimisant l'impact sur les registres disponibles pour d'autres opérations dans le programme.

4.8 Compatibilité avec le langage Deca

Notre conception s'appuie sur une adaptation rigoureuse des spécificités du langage Deca pour produire un code assembleur conforme. Cela inclut notamment la gestion des types de données, des structures conditionnelles, et des boucles.

5 Validation du compilateur

Pour valider le compilateur développé, une approche rigoureuse et méthodique a été adoptée. Cette validation a été réalisée en plusieurs étapes afin de s'assurer que le compilateur fonctionne correctement et génère un code ARM valide pour l'architecture cible.

- **Définition des tests unitaires :** - Nous avons tout d'abord défini un ensemble de tests unitaires couvrant l'ensemble des fonctionnalités implémentées dans le compilateur. Ces tests ont été conçus pour inclure une gamme variée de cas d'utilisation, comprenant des programmes simples et complexes écrits dans le langage `Deca`. - Les tests incluent la vérification des opérations de base comme les assignations, les expressions arithmétiques, ainsi que des fonctionnalités plus avancées telles que les conversions de types entre entiers et flottants. - Des tests de bordure ont également été mis en place pour valider le comportement du compilateur dans des conditions limites, assurant ainsi une couverture exhaustive.

- **Test des étapes du processus de compilation :** - Chaque étape du processus de compilation (analyse lexicale, analyse syntaxique, génération de l'arbre abstrait, et génération du code) a été isolée et testée indépendamment. Cela a permis de valider chacune des phases du compilateur avant de passer à la suivante. - Des outils de débogage tels que **QEMU** ont été utilisés pour simuler l'exécution des instructions ARM sur une architecture cible, ce qui a permis de tester le code généré dans un environnement contrôlé.

- **Vérification des résultats :** - Après chaque test, les résultats obtenus ont été comparés aux résultats attendus, notamment pour des opérations complexes telles que les conversions de types et les calculs arithmétiques impliquant des entiers et des flottants. - Cette validation a permis de garantir que le compilateur génère correctement du code pour tous les cas d'utilisation du langage `Deca`.

- **Évaluation de la robustesse et des performances :** - Afin de tester la robustesse et la performance du compilateur, des programmes représentatifs de scénarios réels ont été utilisés. Ces tests incluent des programmes complexes, mais aussi des tests de stress impliquant des entrées volumineuses. - Cette approche a permis de mettre en évidence des éventuels problèmes de performance, de consommation de mémoire ou d'autres failles dans la gestion des ressources.

- **Conclusion :** - Les résultats des tests ont confirmé que le compilateur produit un code fonctionnel, fiable et conforme aux spécifications du langage `Deca`. Les erreurs identifiées ont été corrigées, et les optimisations ont permis d'améliorer les performances globales.

5.1 Plateformes d'exécution pour ARMv7-A

Pour tester et exécuter du code généré pour l'architecture ARMv7-A, plusieurs options sont disponibles, dont l'utilisation de scripts automatisés et de l'émulateur QEMU. Ces méthodes permettent de simuler l'exécution du code sans avoir besoin de matériel physique dédié.

5.2 Simulation en ligne de commande avec QEMU

L'utilisation de QEMU pour simuler l'exécution du code ARMv7-A repose sur plusieurs outils de la chaîne de compilation ARM. Les étapes suivantes décrivent le processus de simulation :

- **Assembler le code** : Le fichier source `programme.s` est assemblé en utilisant l'outil `arm-none-eabi-as`.
- **Lier le code** : Ensuite, le fichier objet généré est lié à l'aide de l'outil `arm-none-eabi-ld` pour produire un exécutable au format ELF.
- **Exécuter le programme** : Le programme lié est ensuite exécuté avec `qemu-arm`, qui émule l'architecture ARMv7-A et permet d'observer l'exécution du code.

Scripts d'automatisation

Pour simplifier le processus d'exécution et de validation, deux scripts principaux ont été développés :

- **`script_arm`** : Ce script automatise l'ensemble du processus de génération et d'exécution pour un fichier source `programme.s`. Il comprend les étapes suivantes :

```
arm-none-eabi-as -o programme.o programme.s
arm-none-eabi-ld -o programme.elf programme.o
qemu-arm programme.elf
```

Ce script permet d'assembler, de lier et d'exécuter facilement le code ARM en quelques commandes simples.

- **`test_codegen_arm`** : Ce script est utilisé pour tester l'exécution du code généré sur plusieurs fichiers source à la fois. Il automatise la procédure d'assemblage, de liaison et d'exécution pour chaque fichier, ce qui permet de tester le compilateur à grande échelle. Ce script est particulièrement utile pour valider la robustesse du code généré et pour s'assurer qu'il est cohérent dans différents scénarios.

Ces scripts facilitent la validation et le test du code généré pour ARMv7, offrant un moyen rapide et efficace d'exécuter et de déboguer le code sans nécessiter de matériel physique. L'utilisation de QEMU permet également de simuler un environnement ARMv7 sur des machines hôtes x86, ce qui rend les tests plus accessibles et répétables.

Ces outils ont joué un rôle clé dans l'optimisation du processus de développement et ont permis de détecter rapidement les erreurs dans le code généré.