ENSIMAG

Projet Génie Logiciel

Groupe 13

Documentation de Conception de l'extension ARM

Réalisé par :

Benjelloun Otman
Birée Thomas
Boulouz Amine
El Goumiri Rida
Loginova Aleksandra

Année Universitaire : 2024/2025

Contents

1 Introduction

Cette extension vise à ajouter une compilation pour l'architecture ARM dans le compilateur Deca, en ciblant initialement ARMv7-A (32 bits). Il était sujet d'ensuite transitionner vers AArch64, cela s'est cependant avéré impossible, par souci de manque de temps. La cible est plus précisément ARMv7-A muni du module VFP (Vector Floating Point), qui introduit non seulement des instructions de calculs de flottants, mais aussi les regitres s0-s31 et d0-d15.

2 Caractéristiques et Progrès Actuels

Fonctionnalités existantes

- **Registres ARM** :
  - Gestion de 16 registres pour ARMv7-A (R0-R15), y compris des registres spéciaux comme `SP`, `FP`, et `PC`.
  - Registres généraux gérés avec des classes spécifiques (ex. `ARMRegister`), qui est analogue à `Register`. Nous y reviendrons plus en détails ultérieurement.
  - Gestion des registres s0-s31 (Single Precision floating point registers) et des registres d0-d15 (Double Precision floating point registers). Des classes associées ont été introduites : `ARMSPRegister` et `ARMDPRegister`. Leur gestion étant assez analogue à celle des registres généraux, elle figure dans la classe `ARMRegister`.
- **Compilation basique** :
  - Mise en place des classes analogues à celles d'IMA dans le paquetage `fr.ensimag.arm.pseudocode`.
  - Ajout de l'option `--arm` dans le compilateur Deca pour activer la compilation ARM.
  - Le linking et la transformation en binaire de l'assembleur produits se font avec `arm-linux-gnueabihf-as` et `arm-linux-gnueabihf-gcc` (ou encore `arm-linux-gnueabi-as` et `arm-linux-gnueabi-gcc`). Des scripts "jouant le rôle" équivalent du binaire ima sont fournis sous le répertoire `src/test/script/launcher` sous les noms de `run_ubuntu` et `run_archlinux`. Ils requièrent l'installation des packages contenant les binaires menttionnés ci-dessus.
- **Tests préliminaires** :
  - Utilisation de simulateurs tels que `QEMU` pour vérifier l'exécution du code.
- **Avancement actuel**
  - Langage sans-objet partiellement implémenté : la comparaison, le unary minus et le test d'égalité ne sont pas fonctionnels ou ne le sont que partiellement, le modulo ne fonnctionne pas s'il figure dans une expression composée.

2.1 Fonctionnalités non encore implémentées

- **Code génération** :
  - Les méthodes spécifiques à ARM (`codeGen*ARM`) ne sont pas encore toutes implémentées.
  - Gestion des instructions avancées et des optimisations de la gestion des registres (il est question d'utiliser un algorithme plus performant et plus connu, plutôt que notre façon de faire actuelle, assez naïve).
  - Transitions vers AArch64. En soi, par rétrocompatibilité, le code ARMv7-A fonctionne parfaitement pour tout processeur AArch64. Une transition signifierait la gestion d'une troisème architecture, bien que celle-ci serait bien plus similaire à ARMv7-A que'ARMv7-A ne l'était par rapport à IMA.
- **Gestion avancée des registres** :
  - Adaptation des fonctionnalités pour les 31 registres d'AArch64 (64 bits).
  - Transition vers des instructions spécifiques à ARMv8-A et gestion des niveaux d'exception.
  - Contrairement à IMA, les registres ARM ne sont pas typés dynamiquement. Une Encore une fois, cela soulève la nécessité d'une nouvelle approche de gestion des registres, qui est le point le plus déliquat et le plus fréquemment problématique. Un changement de paradigme vers une approche plus fiable et plus robuste (surtout en considérant l'intoduction des registres s0-s31 et d0-d15) s'avère de plus en plus nécessaire. La difficulté présentée est réellement le respect et la compréhension de l'ABI ARM, surtout la spécification sur les registres.
- **Compatibilité complète avec les fonctionnalités Deca** :
  - Le langage sans-objet pourrait être finalisé en une après-midi, bien que cela soit sans tenir compte de la transition vers une meilleure gestion de registres, ce qui n'est pas recommandé.
  - L'élaboration d'une meilleur stratégie de gestion des registres s'effectuerait, elle, en moins de deux jours. C'est une estimation assez réfléchie, compte tenu de la familiarité acquise envers le projet, et surtout envers l'ABI ARM.
  - La partie Objet n'a pas encore été abordée. Nous restons optimistes quant à la durée que cela prendrait, surtout en considérant notre familiarité avec IMA, la sémantique de Deca, et le compilateur en général. Cela devra cependant être accompagné par un processus de résolution de bugs – processus qui se déroulerait en parallèle. Nous estimons 4 à 5 jours additionnels.
  - La gestion d'erreurs runtime n'a elle non plus pas été abordée. Elle se ferait cependant en parallèle avec ce qui a été évoqué ci-dessus.

3 ARMCodegen

3.1 Gestion des instructions `print` :

Les méthodes `ARMCodeGenPrint` sont conçues pour générer le code correspondant à l'instruction `print` sur l'architecture ARM. Voici une explication du fonctionnement :

- **Évaluation de l'expression à imprimer** :
  - Utilisation de `ARMCodeGenInst` pour évaluer l'expression et placer son résultat dans un registre ou sur la pile.
- **Sauvegarde des registres nécessaires** :
  - Les registres critiques (`R1`, `R0`, `R2`, `R3`) sont sauvegardés sur la pile si déjà utilisés.
- **Chargement des données et appel à `printf`** :
  - Les formats (par exemple `int_format`, `float_format`) sont chargés dans `R0`.
  - Conversion des nombres flottants en double précision (avec l'instruction `VCVT. .F64..F32`). Ils sont ensuite stockés dans les registres `R2` et `R3` (`R2` contenant les 32 bits de poids forts, et `R3` contenant ceux de poids faibles). Ce comportement, obscure à première vue, s'explique en deux temps :
    - l'ABI ARM dicte que les registres `R0 - R4` continnent les 5 premiers arguments éventuels d'une fonction appelée, le reste (si présent) étant ensuite présents sur le stack. L'alignement de ces derniers est lui aussi d'une grande importance : une valeur stockée sur 32 bits prend tout simplement le registre le plus "proche" (i.e `R0` à `R4` par priorité). Les valeurs stockées sur 64 bits doivent quant à elles être stockées sur deux registres, dont le premier doit être pair (`R2`, puis `R4` si ce dernier est déjà utilisé) qui contiendra les bits de poids fort. Les 32 bits de poids faibles seront quant à eux stockés dans le registre qui suit directement ce dernier, ou éventuellement le stack.
    - La fonction de la libc `printf` s'attend à ce que tout argument de type `float` soit en fait traîté comme un `double` (Une nécessité bien obscure, qui est, disons-le, embêtante à dénicher). Ainsi, le string de format est passé en `R0`, et le float est d'abord transformé en unn double, qui est ensuite stocké en `R2` et `R3` avant l'appel à `printf`
  - L'instruction `BL` est utilisée pour effectuer un appel à `printf` avec les arguments préparés.
- **Restauration des registres sauvegardés** :
  - Les registres utilisés précédemment sont restaurés depuis la pile.

Ce processus garantit que les instructions ARM respectent les conventions d'appel et fonctionnent correctement dans un environnement simulé ou réel.

3.2 Gestion des opérations arithmétiques

Le module `AbstractOpArith` gère les opérations arithmétiques binaires (`+`, `-`, `*`, `/`, `%`) pour l'extension ARM. Son implémentation repose sur les points suivants :

- **Évaluation des opérandes** :
  - Les opérandes gauche et droite sont évalués et stockés soit dans des registres généraux (`R0-R15`) soit sur la pile. La gestion des positions est assurée par `ARMRegister` et `ARMDVal`.
- **Calculs et instructions en fonction du type** :
  - Les opérations sur les `int` utilisent des instructions génériques (`ADD`, `MUL`, etc.).
  - Les opérations sur les `float` exploitent les registres flottants (`s0-s31`) et des instructions spécifiques (`VADD..F32`, `VMUL..F32`, etc.).
- **Gestion des registres critiques** :
  - Les registres critiques (`R0`, `R1`) sont sauvegardés/restaurés si nécessaire pour respecter les conventions d'appel.
  - Les registres flottants temporaires (`s`) sont alloués dynamiquement et libérés après usage.
- **Cas particuliers pour les divisions et modulo** :
  - Les divisions et modulos sur `int` utilisent la libc ARM, plus précisément, ces opératins font appel à (`__aeabi_idiv`), implémentant la division entière en logiciel.
  - Les calculs sur `float` nécessitent des conversions et des instructions adaptées (`VDIV..F32`).
- **Détection des débordements** :
  - Lorsqu'activée, l'instruction `BOV` est insérée pour vérifier les débordements en cas d'opérations sur des `float`. (Ceci n'est pas encore implémenté, mais est totalement analogue à ce qui a été fait pour IMA. Encore une fois, le temps a été notre ennemi principal. L'Homme ne pourra jamais se défaire de l'avancée impitoyable du temps.)

3.3 Gestion des Assignations

Le module `Assign` gère les assignations (`lvalue = expr`) en tenant compte des différentes configurations possibles de l'opérande gauche (`lvalue`) et de l'opérande droit (`expr`). Voici les points principaux de son implémentation :

- **Évaluation de l'opérande droit** :
  - L'expression à assigner (`expr`) est évaluée et stockée soit dans un registre (`GPRegister`) soit temporairement sur la pile (`SP`).
  - La méthode `ARMCodeGenInst` génère le code correspondant à cette évaluation.

- **Détection du contexte de l'opérande gauche** :
  - Si `lvalue` est un champ (`field`), une gestion spécifique est appliquée pour calculer son adresse effective.
  - Si `lvalue` est une variable simple, son adresse est récupérée directement.
  - Les paramètres sont gérés en fonction de leur position relative au pointeur de base (`LB`).
- **Assignation des données** :
  - Si `expr` est sur la pile, une instruction `POP` est utilisée pour la récupérer dans un registre temporaire (`R1`).
  - Si `expr` est dans un registre, une instruction `STR` est utilisée pour copier la valeur dans l'adresse de `lvalue`.
- **Gestion des champs (`field`)** :
  - Lorsqu'un champ est assigné, un registre inutilisé est temporairement alloué pour calculer l'adresse effective.
  - La valeur de `expr` est ensuite stockée à cette adresse via une instruction `STR`.

Cette gestion assure la flexibilité nécessaire pour traiter différents types d'assignations tout en respectant les conventions de l'architecture ARM.

3.4 Gestion des constantes booléennes

Le module `BooleanLiteral` gère la représentation et la génération de code des constantes booléennes (`true` ou `false`) dans l'extension ARM. Voici les points principaux de son implémentation :

- **Génération de code ARM** :
  - La constante booléenne est traduite en `#1` ou `#0` en utilisant un opérande immédiat (`ARMImmediateInteger`).
  - Si un registre général ARM (`R0-R8`) est disponible, l'instruction `MOV` est utilisée pour y charger la valeur.
  - Sinon, la valeur est placée temporairement sur la pile à l'aide de `PUSH`.
- **Optimisation et gestion des registres** :
  - Les registres inutilisés sont marqués comme utilisés temporairement et libérés après leur usage.
  - La pile est utilisée uniquement lorsque les registres disponibles sont insuffisants.

Cette implémentation garantit une gestion efficace et conforme des constantes booléennes dans le compilateur Deca, tout en respectant les contraintes spécifiques à l'architecture ARM. On notera que l'implémentation est analogue à celle d'IMA.

3.5 Gestion des déclarations de variables

Le module `DeclVar` gère les déclarations de variables, leur vérification contextuelle et la génération du code correspondant pour les architectures IMA et ARM. Une spécificité d'ARMv7-A est que seul certains flottants immédiats peuvent être acceptés. Il n'y a non plus pas d'immédiats string. Il s'avère qu'il est nécessaire de rajouter une section `.data` en début de code, dans laquelle sont spécifiés des littéraux contenant les valeurs des flottants et des strings à charger dans les registres. Similairement, il est impossible de charger directement dans un registre du VFP des littéraux flottants (ou du moins, ceci n'est possible que pour certains cas...) et donc toutes les valeurs sont chargées dans des registres ARM directement. Ils sont ensuite chargés dans les registres du VFP quand cela est nécessaire. Cette approche n'est aucunement optimale, elle décuple le nombre d'instructions de `LDR`, `VLDR`, `STR` et `VSTR`, surtout pour des programmes où beaucoup d'opérations entre flottants sont nécessaires. Voici alors les principaux points abordés dans l'implémentation :

- **Vérification contextuelle** :
  - Le type de la variable est validé pour s'assurer qu'il est compatible avec les déclarations de variables (par exemple, `void` n'est pas autorisé).
  - La variable est associée à une définition de type `VariableDefinition`, qui est déclarée dans l'environnement local.
  - L'initialisation de la variable est vérifiée, que ce soit une initialisation explicite (`Initialization`) ou implicite (`NoInitialization`).
- **Génération de code pour ARM** :
  - Une position mémoire est attribuée dans le registre de base ARM (`SB`) à l'aide d'un `ARMRegisterOffset`.
  - Si une initialisation est spécifiée, l'expression est évaluée et le résultat est stocké en mémoire en utilisant les instructions `STR` (pour tous les types de valeurs, étant donné le fait qu'`VSTR` ne fonctionne pas pour certaines valeurs littérales).
- **Gestion des registres et de la pile** :
  - Les registres inutilisés sont alloués pour le calcul de l'expression associée à l'initialisation, et libérés après leur utilisation.
  - La pile est utilisée si les registres disponibles sont insuffisants.

Cette gestion permet de garantir une déclaration de variables correcte et optimisée, tout en prenant en charge les spécificités de l'architecture ARM.

3.6 Gestion des littéraux flottants

Le module `FloatLiteral` gère les valeurs constantes en virgule flottante (simple précision) dans le langage Deca et leur traduction vers les architectures IMA et ARM. Voici les principales fonctionnalités :

- **Vérification contextuelle** :

- Lors de la vérification contextuelle, le littéral est validé pour s'assurer qu'il n'est ni `NaN` ni infini.
- Le type `FLOAT` est associé au littéral pour garantir la compatibilité avec les opérations de calcul.

- **Génération de code pour ARM** :
  - Une entrée spécifique est créée pour chaque littéral flottant dans la section des données (`ARMDataSection`).
  - La valeur est chargée dans un registre ARM à l'aide de l'instruction `LDR`. Si aucun registre n'est disponible, elle est placée sur la pile (`PUSH`).

- **Impression des littéraux** :
  - Pour IMA, les instructions `WFLOAT` ou `WFLOATX` sont utilisées pour imprimer respectivement en format décimal ou hexadécimal.
  - Pour ARM, le format `"float_format"` est utilisé et l'impression est réalisée via l'instruction `BL` pour appeler `printf`.
  - Les valeurs sont converties en double comme cela est nécessaire (`VCVT.F64.F32`) et transférées vers les registres (`R2`, `R3`) pour respecter les conventions d'appel.

- **Optimisation des registres** :
  - Les registres inutilisés sont marqués comme disponibles après usage.
  - La pile est utilisée uniquement en dernier recours pour réduire les conflits de registre.

Cette implémentation garantit une gestion efficace des constantes flottantes dans le compilateur Deca, tout en respectant les contraintes des architectures IMA et ARM.

3.7 Gestion des identificateurs

Le module `Identifier` permet de manipuler les identificateurs dans le langage Deca et de gérer leur traduction en code machine pour les architectures IMA et ARM. Voici une description détaillée des fonctionnalités :

Vérification contextuelle

- **Validation des définitions** :
  - Lorsqu'un identificateur est utilisé, son type et sa définition sont vérifiés dans l'environnement local.
  - Le module gère différents types d'identificateurs : variables, paramètres, champs, méthodes et expressions générales.
  - En cas d'incompatibilité entre le type attendu et celui de l'identificateur, une erreur contextuelle est levée.

- **Prise en charge des types spécifiques** :
  - Les identificateurs associés aux types `FieldDefinition`, `MethodDefinition`, `ParamDefinition`, ou `VariableDefinition` sont correctement traités grâce à des vérifications explicites et des casts sécurisés.

Génération de code IMA

- **Chargement des valeurs** :
  - Les valeurs associées aux identificateurs sont chargées dans un registre général disponible (`GPRegister`).
  - Si aucun registre n'est disponible, les valeurs sont temporairement stockées sur la pile (`SP`).
- **Impression des valeurs** :
  - Les identificateurs de type entier (`int`) sont imprimés avec l'instruction `WINT`.
  - Les identificateurs de type flottant (`float`) sont imprimés en format décimal ou hexadécimal à l'aide des instructions `WFLOAT` et `WFLOATX`.

Génération de code ARM

- **Chargement et impression des valeurs** :
  - Les valeurs des identificateurs sont récupérées depuis leur position en mémoire (`LDR`) et, si nécessaire, chargées dans des registres du VFP et/ou converties en double précision (`VCVT.F64.F32`).
  - Pour les identificateurs entiers, les formats (`int_format`) sont chargés dans `R0`, puis imprimés avec `BL`.
  - Les identificateurs flottants utilisent le format `float_format`, et leurs valeurs sont transférées dans `R2` et `R3` avant l'appel à `printf`.
- **Sauvegarde et restauration des registres** :
  - Les registres critiques (`R0`, `R1`, `R2`, `R3`) sont sauvegardés sur la pile pour éviter tout conflit lors de l'exécution des instructions.
  - Une fois l'impression terminée, les registres sont restaurés dans leur état initial.

Optimisation

- **Gestion des registres** :
  - Les registres inutilisés sont marqués comme disponibles après utilisation.
  - La pile est utilisée comme solution de secours uniquement lorsque tous les registres sont occupés.
- **Compatibilité avec les champs** :
  - Les identificateurs correspondant à des champs (`FieldDefinition`) nécessitent des étapes supplémentaires pour accéder à leur emplacement mémoire via des registres temporaires.

Cette implémentation garantit une gestion robuste et efficace des identificateurs, tout en respectant les spécificités des architectures IMA et ARM.

3.8 IfThenElse : Génération de Code

Le module `IfThenElse` est utilisé pour gérer la génération de code des structures conditionnelles complexes en ARM. Cette section décrit la logique de compilation des blocs `if`, `else if` et `else`.

Génération de Code pour ARM

- **Création des étiquettes :**
  - Les étiquettes spécifiques pour les blocs `if`, `else` et `end_if` sont générées dynamiquement à l'aide d'un compteur global `ifClauseCount`.
- **Évaluation de la condition :**
  - La condition est réalisée en utilisant `CMP`, comparant la valeur calculée avec zéro.
  - Si la condition échoue, un saut (`BEQ`) vers la branche `else` est effectué.
- **Compilation des branches :**
  - Les instructions du bloc `then` sont générées en premier.
  - Un saut inconditionnel (`B`) est ajouté pour diriger l'exécution vers `end_if`.
  - Les instructions du bloc `else` sont compilées ensuite.
- **Finalisation :**
  - Une étiquette de fin est placée après le bloc `else` pour marquer la fin de la structure conditionnelle.

Résumé du Processus

Le code généré pour une structure conditionnelle en ARM garantit une exécution correcte grâce à :

- Une gestion précise des étiquettes.
- Une séparation claire des blocs `if`, `else` et `end_if`.
- Une optimisation de l'utilisation des registres pour limiter l'encombrement mémoire.

3.9 Main : Génération de Code

Le module `Main` est au cœur de l'exécution du programme principal. Cette section détaille les mécanismes mis en œuvre pour la génération de code sur l'architecture ARM.

Génération de Code pour ARM

- **Déclarations de variables :**
  - Les variables globales sont allouées dans la pile ARM via la méthode `ARMCodeGenListDeclVar`.
  - Un commentaire est ajouté pour signaler le début des déclarations.
- **Instructions principales :**
  - Les instructions principales du programme sont compilées à l'aide de `ARMCodeGenListInst`.
  - Chaque instruction est traduite en son équivalent ARM en respectant les conventions de l'architecture.
- **Optimisation :**
  - L'optimisation de la gestion des registres est effectuée pour limiter les accès à la pile et maximiser l'utilisation des registres ARM.

Cette structure garantit une exécution fiable et conforme aux spécifications du compilateur Deca pour ARM.

3.10 Program : Génération de Code ARM

La classe `Program` est responsable de la gestion globale du programme Deca, incluant la génération de code pour les déclarations de classes, le programme principal, et la section des données. Voici une description détaillée des étapes de génération de code ARM.

Structure de la Génération de Code

La méthode `ARMCodeGenProgram` effectue les étapes suivantes :

1. **Génération du bloc principal** :
   - La méthode `ARMCodeGenMain` génère les instructions ARM associées au bloc principal du programme.
2. **Ajout de la terminaison du programme** :
   - Des instructions spécifiques pour arrêter l'exécution sont ajoutées via la méthode `ARMCodeGenHalt`.
3. **Génération de la section .text** :
   - La méthode `ARMCodeGenMainLabel` initialise la section `.text`, définit l'entrée principale avec une étiquette `main`, et configure les registres de base (`SP`, `SB`, et `FP`).
4. **Ajout des données nécessaires** :

- La section des données (`.data`) est générée via la méthode `ARMDataSection.codeGenDataSection`, incluant les chaînes de caractères, les formats, et les constantes flottantes nécessaires.

5. **Configuration de l'entête ARM** :
   - Les directives spécifiques à l'architecture ARM, telles que `.arm`, `.arch armv7-a`, et `.syntax unified`, sont définies par `ARMCodeGenHeader`.

Détails Techniques

- **Bloc Principal** :
  - Les instructions du programme principal sont générées et marquées avec l'étiquette `main`.
  - Les registres de base (`SP`, `SB`, `FP`) sont configurés en début de programme.
- **Gestion des Arrêts** :
  - Une séquence d'arrêt est générée en utilisant des instructions telles que `MOV` et `SVC`, avec les registres `R7` et `R0`.
- **Données Statiques** :
  - La section `.data` contient toutes les constantes nécessaires à l'exécution, telles que les formats pour les entiers et les flottants (`int_format`, `float_format`).

Résumé

La méthode `ARMCodeGenProgram` assure une génération de code efficace et conforme pour l'architecture ARMv7-A. Elle respecte les conventions d'appel ARM, prépare les registres nécessaires, et structure le programme pour une exécution correcte sur une plateforme ARM.

3.11 ARMCodeGen : Génération de Code pour les Instructions While en ARM

Le module `ARMCodeGenInst` associé à l'instruction `While` vise à générer du code ARM optimisé pour gérer les boucles en respectant les conventions ARM tout en garantissant une exécution correcte et efficace.

Structure Générale de la Boucle While

L'instruction `While` est composée de deux parties principales :

- **La Condition** : Elle est évaluée en début de boucle pour déterminer si le corps doit être exécuté.
- **Le Corps de la Boucle** : Une liste d'instructions exécutées tant que la condition est satisfaite.

Processus de Génération du Code ARM

Voici les étapes suivies pour générer le code ARM correspondant :

1. **Création des Étiquettes** :
   - Une étiquette de début (`startLabel`) est créée pour marquer le début de la boucle.
   - Une étiquette de fin (`endLabel`) est créée pour marquer la sortie de la boucle lorsque la condition n'est plus vérifiée.
2. **Évaluation de la Condition** :
   - L'expression conditionnelle est traduite en instructions ARM à l'aide de la méthode `ARMCodeGenInst`.
   - Le résultat est placé dans un registre général (`ARMGPRegister`).
3. **Comparaison de la Condition** :
   - Une comparaison est effectuée entre la valeur du registre contenant la condition et une valeur cible (`0` ou `1`, selon le cas).
   - Si la condition échoue, un saut est effectué vers `endLabel` à l'aide de l'instruction `BGE` (Branch if Greater or Equal).
4. **Exécution du Corps de la Boucle** :
   - Les instructions de la boucle sont traduites et insérées entre `startLabel` et `endLabel`.
   - À la fin du corps, un saut inconditionnel (`B`) est effectué vers `startLabel` pour réévaluer la condition.
5. **Gestion des Registres** :
   - Les registres utilisés pour la condition et les variables temporaires sont marqués comme inutilisés (`ARMRegister.setUnused()`) après la fin de la boucle.

Avantages de l'Implémentation

- **Respect des Conventions ARM** : Le code généré est compatible avec les conventions ARMv7-A, garantissant une portabilité sur des plateformes ARM standard.
- **Optimisation de l'Utilisation des Registres** : Les registres inutilisés sont correctement libérés pour éviter tout conflit avec d'autres parties du programme.
- **Modularité** : La méthode `ARMCodeGenInst` peut être étendue pour gérer des instructions plus complexes ou des optimisations spécifiques.

4 Limitations

1. **Politique de gestion des registres** : Notre approche initialie, quelque peu naïve, est en soi fonctionnelle. Cependant, elle rajoute beaucoup de complexité à gérer dans la génération de code elle-même, et ne nous permet pas de générer un code optimisé. Il est donc préférable, comme mentionné précédemment, d'explorer de meilleurs algorithmes.

2. **Portée des instructions** : Seules les instructions nécessaires au développement actuel sont disponibles. Des fonctionnalités avancées de l'architecture ARMv7-A, comme la gestion complète des niveaux d'exceptions ou des optimisations spécifiques, restent encore inexploitées. Cela limite les performances et l'évolutivité du compilateur.

3. **Performance** : Les algorithmes de génération de code actuels favorisent la simplicité et la généricité au détriment de la performance. Par exemple, les séquences d'instructions générées ne tirent pas encore pleinement parti des optimisations propres à ARM, comme l'utilisation des instructions SIMD ou des pipelines spécifiques.

4. **Compatibilité ARMv7-A** : Bien que la cible principale soit ARMv7-A avec le module VFP, des fonctionnalités spécifiques, comme la gestion avancée des registres flottants (`s0-s31`) et doubles (`d0-d15`), nécessitent encore des ajustements pour une compatibilité optimale.

5. **Tests sur matériel réel** : Actuellement, le code généré est principalement testé sur des simulateurs tels que `QEMU`. Bien que fonctionnels, ces tests doivent être complétés par des validations sur un matériel ARM réel. Un **Raspberry Pi**, compatible avec ARMv7-A, est disponible pour effectuer ces tests, mais son intégration dans le cycle de développement reste partielle.

6. **Incompatibilité avec ARMv8-A** : La compatibilité avec ARMv8-A (AArch64) est partielle et nécessiterait une transition significative. Cette transition inclut la gestion de 31 registres généraux 64 bits et l'adoption des conventions propres à cette architecture. Cependant, ces évolutions dépassent les objectifs du projet actuel.

5 Prochaines Étapes

Pour garantir une extension ARM pleinement fonctionnelle et robuste, plusieurs étapes clés doivent encore être réalisées :

- **Développer une nouvelle politiqque de gestion des registres** :
  - Cela permettra de simplifier la génération de code, et de produire un programme plus optimal, plus fonctionnel, plus modulaire et facilement testable.
- **Compléter les classes et méthodes ARM** :
  - Finaliser les classes dans le paquetage `fr.ensimag.arm.pseudocode`, notamment celles liées aux registres, instructions et étiquettes.
  - Établir une gestion optimale des registres flottants (`s0-s31`) et doubles (`d0-d15`) pour les calculs à précision variable.

- **Implémenter la génération de code ARM** :
  - Développer les méthodes spécifiques pour toutes les structures du langage Deca, comme les boucles, les conditions et les appels de fonctions.
  - Optimiser les instructions ARM générées pour minimiser les cycles d'exécution et l'utilisation de la pile.
- **Tests intensifs sur des plateformes ARM réelles** :
  - Utiliser un **Raspberry Pi** comme plateforme de test principale pour exécuter le code généré et valider son bon fonctionnement.
  - Effectuer des tests de performance et de compatibilité à l'aide de simulateurs comme `QEMU` avant la validation sur du matériel réel.
- **Documentation et validation** :
  - Documenter chaque fonctionnalité nouvellement ajoutée, notamment les spécificités des registres ARM et des conventions d'appel.
  - Valider les différentes étapes avec des cas de test étendus, couvrant à la fois les scénarios simples et complexes.

L'intégration du Raspberry Pi dans la boucle de test permettra de s'assurer que le compilateur génère du code réellement exécutable sur une architecture ARM, en reproduisant un environnement pratique et proche des usages réels.