

# Documentation de l'extension ARM pour le langage Deca

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

24 janvier 2025

## Table des matières

<b>Introduction</b>	<b>2</b>
<b>Spécification de l'extension ARM</b>	<b>3</b>
Définition de la cible . . . . .	3
Pourquoi le Cortex-A53 ? . . . . .	3
Objectifs : . . . . .	4
Défis techniques . . . . .	5
Plan d'action . . . . .	5
Méthode de validation . . . . .	6
Définition du langage d'assemblage ARMv8-A . . . . .	6
Execution simple séquentielle . . . . .	7
1. Phase In Order . . . . .	7
2. Étape d'Issue . . . . .	8
3. Phase Out of Order . . . . .	8
Registres . . . . .	8
Registres à usage général . . . . .	8
Registres pour les opérations flottantes et vectorielles . . . . .	9
Autres registres . . . . .	9
Modes d'adressages . . . . .	9
Opérations arithmétique et logiques . . . . .	10
Exemples . . . . .	11
Codes conditions . . . . .	12
Exemples . . . . .	12
Contrôle . . . . .	13
Branchement inconditionnel . . . . .	13
Branchement contionnel . . . . .	13
Appels de fonctions . . . . .	13
<b>Mise en place de l'environnement</b>	<b>13</b>
<b>Conception</b>	<b>14</b>
Compilation avec CLANG et GCC . . . . .	14
Commandes pour la Compilation . . . . .	14
Différences entre Clang et GCC dans le Contexte de Deca . . . . .	15

Manières de procéder . . . . .	15
Étapes initiales : gestion des variables entières . . . . .	15
Ajout des opérations arithmétiques et booléennes . . . . .	15
Gestion des branchements conditionnels . . . . .	15
Prise en charge des flottants . . . . .	16
Documentation et inspirations . . . . .	16
Utilisation de la libc . . . . .	16
Implémentation . . . . .	16
Gestion de la mémoire . . . . .	16
Mise en place de la pile et des normes . . . . .	17
Méthodes utilitaires . . . . .	17
Mise en place de tests . . . . .	17
UML . . . . .	17
<b>Résultats obtenus</b>	<b>17</b>
<b>Conclusion</b>	<b>18</b>
Conclusion . . . . .	18

## 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 Moore : 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 quatre étapes principales : une spécification détaillée de l'extension 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 étions fixées étaient 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 souhaitons 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 par ces contraintes, nous avons le choix entre 3 processeurs 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 efficacité énergétique. Nous l'avons choisi comme cible matérielle pour plusieurs raisons :

1. **Retrocompatibilité** avec l'architecture ARMv7-A 32-bits
2. **Efficacité 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.

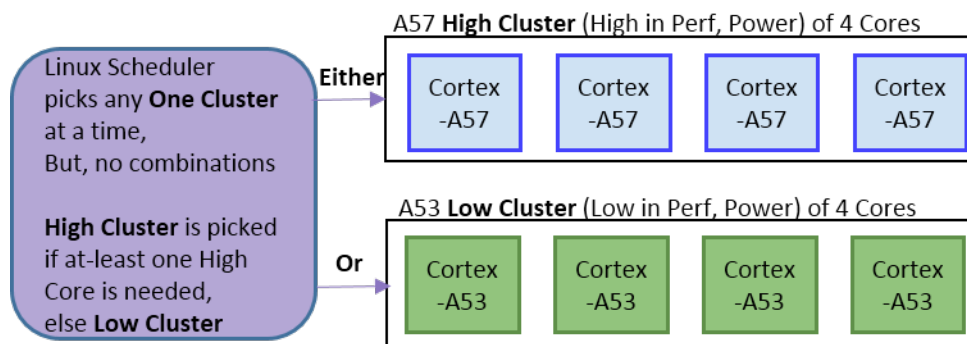


FIG. 1 : Changement de Cluster

## 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 atteint s'il remplit cette spécification :

1. Gestion des opérations arithmétiques (addition, soustraction, division, multiplication et modulo) sur les **entiers** et les **flottants** (via un cast implicite).
2. Gestion des opérations booléennes :
  - 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 bibliothèques 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 développements. Nous avons alors établi ces différentes étapes :

### 1. Spécification et Analyse préliminaire :

- Choix de la cible
- Étude de 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 chaîne 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”).

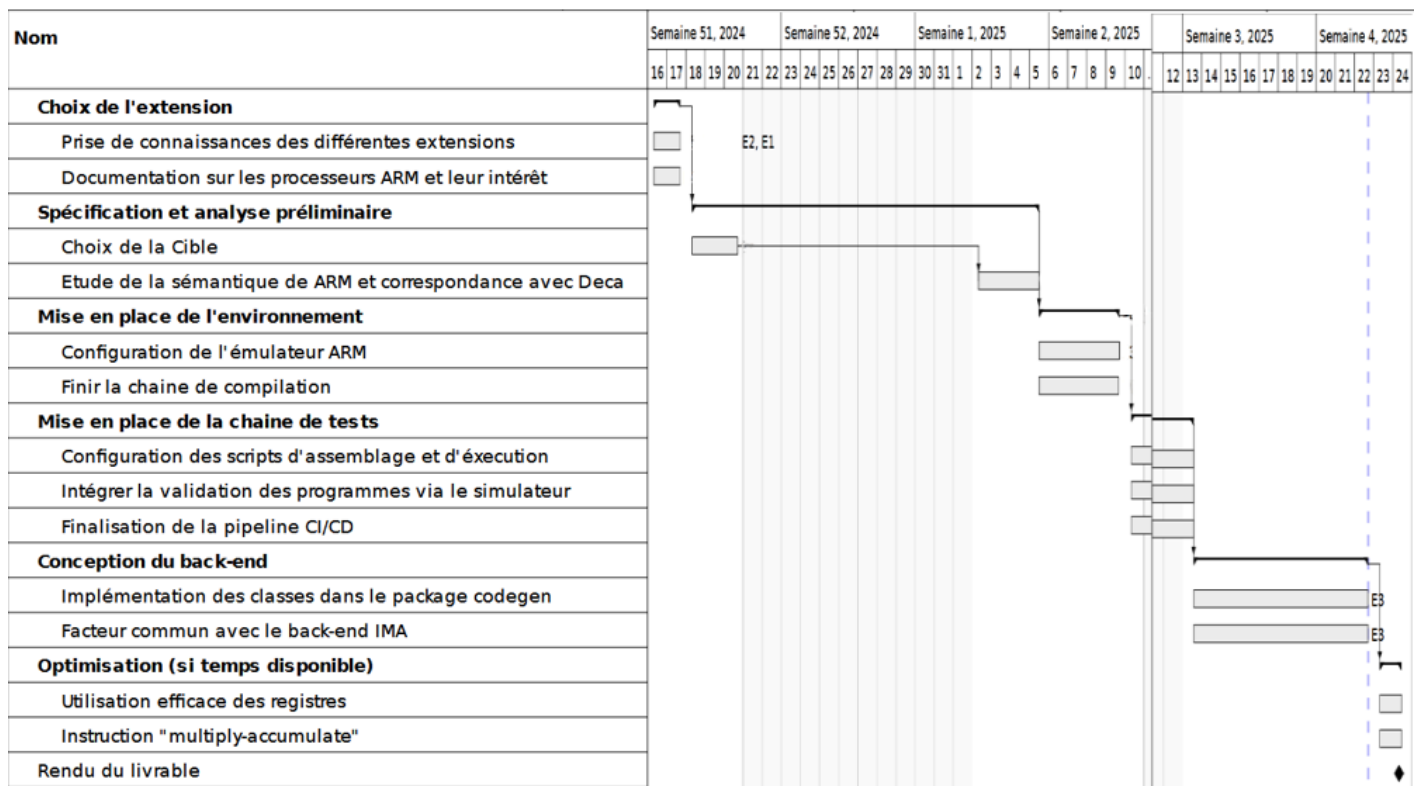


FIG. 2 : Diagramme de Gantt pour ARM

## 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.

## Définition du langage d'assemblage ARMv8-A

L'**Instruction Set Architecture** (ISA) constitue une partie essentielle du modèle abstrait d'un ordinateur. Elle définit l'ensemble des instructions que le processeur est capable d'exécuter et le mécanisme par lequel le logiciel contrôle le matériel.

Dans le cadre de l'extension de notre compilateur vers le langage d'assemblage cible (ARMv8-A), nous nous sommes appuyés sur les spécifications détaillées de l'ISA fournies par ARM. Ces documents officiels nous ont permis de comprendre les particularités de l'architecture ARMv8-A, d'adapter nos outils de développement et d'assurer la compatibilité avec notre processeur cible.

## Execution simple séquentielle

L'architecture ARM utilise historiquement le modèle **SSE (Simple Sequential Execution)**, où le processeur traite une instruction à la fois en suivant les étapes classiques : **Fetch**, **Decode**, et **Execute**. Ce traitement est séquentiel, c'est-à-dire que chaque instruction est exécutée dans l'ordre exact dans lequel elle apparaît en mémoire.

Cependant, avec les processeurs modernes, comme ceux basés sur l'architecture ARM-Cortex, cette approche a évolué. Désormais, ces processeurs utilisent des pipelines avancés capables de :

- **Traiter plusieurs instructions simultanément**, grâce à une exécution en parallèle.
- **Exécuter les instructions dans un ordre différent** de celui prévu initialement, en exploitant les ressources disponibles pour maximiser les performances.

Le schéma suivant illustre un exemple de pipeline pour un processeur ARM-Cortex :

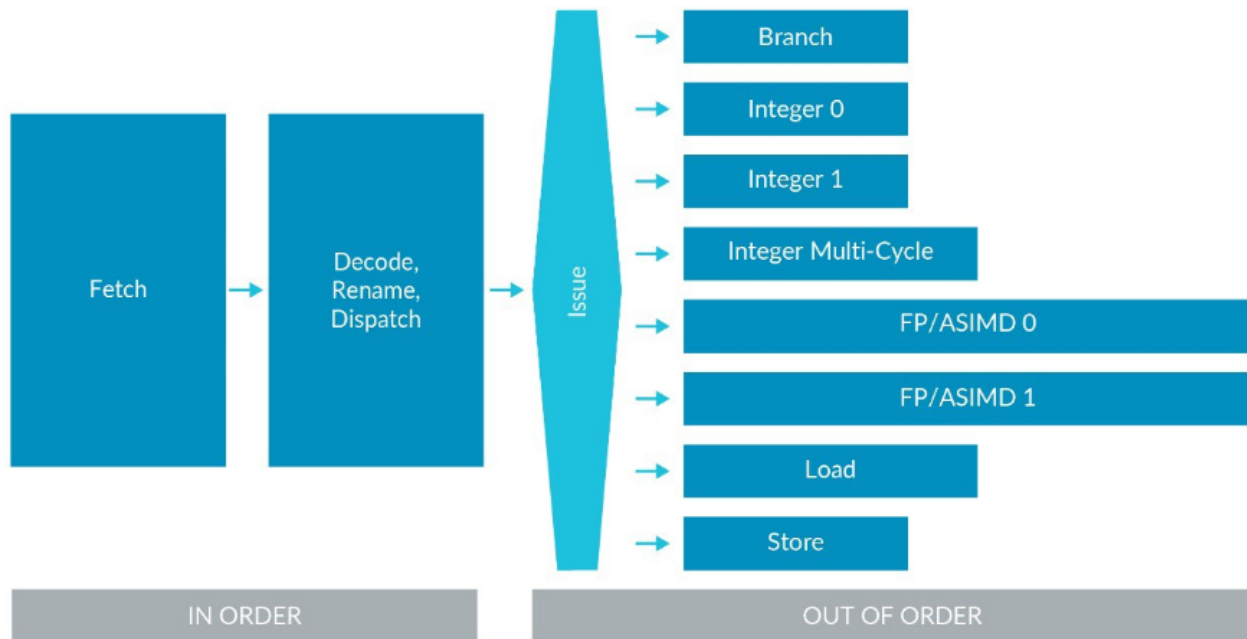


FIG. 3 : Pipeline ARM-Cortex

Cette architecture pipeline est divisée en deux grandes phases : **“In Order”** (traitement ordonné des instructions) et **“Out of Order”** (exécution désordonnée des instructions). Ces deux parties sont reliées par l'étape clé d'**issue**, qui distribue les instructions aux différentes unités d'exécution.

**1. Phase In Order** Dans cette phase, les instructions sont traitées dans l'ordre dans lequel elles apparaissent dans le code source. Elle inclut les étapes suivantes :

- **Fetch** : Cette étape récupère les instructions depuis la mémoire ou le cache, en identifiant celles qui doivent être exécutées.
- **Decode** : Les instructions sont décodées en micro-opérations compréhensibles par le processeur.
- **Rename** : Les registres utilisés par les instructions sont renommés pour éviter les conflits liés aux dépendances.

- **Dispatch** : Les instructions décodées et renommées sont placées dans une file d'attente et prêtes à être distribuées aux unités fonctionnelles.

**2. Étape d'Issue** L'étape d'Issue assure la transition entre les phases **In Order** et **Out of Order**. À ce stade, les instructions sont assignées aux unités d'exécution spécialisées en fonction de la disponibilité des ressources matérielles et des opérandes requis. Cette répartition permet une exécution potentiellement hors de l'ordre initial du programme.

**3. Phase Out of Order** Dans cette phase, les instructions sont exécutées de manière désordonnée, mais les résultats sont toujours restitués dans l'ordre prévu par le programme (grâce au mécanisme de **commit**). Les unités fonctionnelles spécialisées incluent :

- **Branch** : Gère les instructions de branchement (conditionnelles ou non), comme les boucles et les instructions conditionnelles.
- **Integer 0 et Integer 1** : Exécutent les opérations arithmétiques simples (addition, soustraction, etc.). La duplication de ces unités permet une exécution parallèle.
- **Integer Multi-Cycle** : Traite les opérations entières complexes (multiplication, division) nécessitant plusieurs cycles.
- **FP/ASIMD 0 et FP/ASIMD 1** : Spécialisées dans les calculs en virgule flottante (FP) et SIMD (Single Instruction Multiple Data), utiles pour les calculs vectoriels.
- **Load** : Gère les opérations de lecture en mémoire.
- **Store** : Gère les opérations d'écriture en mémoire.

Cette phase présente de nombreux avantages :

- **Amélioration des performances** : Les instructions sont exécutées dès que les ressources nécessaires sont disponibles, même si cela implique une exécution hors de l'ordre du programme.
- **Réduction des latences** : Les dépendances entre instructions sont gérées dynamiquement, minimisant les temps d'attente.
- **Utilisation optimale des ressources** : Les unités fonctionnelles du processeur sont exploitées de manière efficace, maximisant le débit d'exécution.

## Registres

### Registres à usage général

L'architecture ARM fournit 31 registres à usage général. Chaque registre peut être utilisé soit comme un registre 64 bits (nommé **X0** à **X30**), soit comme un registre 32 bits (nommé **W0** à **W30**). Les registres **W** représentent les 32 bits de poids faible des registres **X** correspondants. Lorsqu'une opération est effectuée sur un registre **W**, les 32 bits supérieurs du registre **X** associé sont mis à zéro.

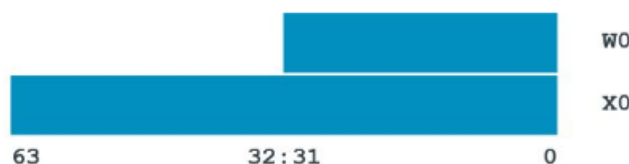


FIG. 4 : Registres à usage général



Le choix entre les registres **X** ou **W** détermine la taille de l'opération effectuée.

```
ADD W0, W1, W2  // Additionne W1 et W2, stocke le résultat 32 bits dans W0
ADD X0, X1, X2  // Additionne X1 et X2, stocke le résultat 64 bits dans X0
```

**Registres pour les opérations flottantes et vectorielles** L'architecture ARM inclut également un ensemble de 32 registres dédiés aux opérations en virgule flottante et vectorielles. Ces registres, nommés V0 à V31, sont chacun de 128 bits. Ils peuvent être adressés de différentes manières pour représenter des données de tailles variées (x représente le numéro de registre) :

- Bx : 8bits
- Hx : 16 bits
- Sx : 32 bits
- Dx : 64 bits
- Qx : 128 bits

Lorsqu'on utilise un registre **V**, le registre est traité comme un vecteur.

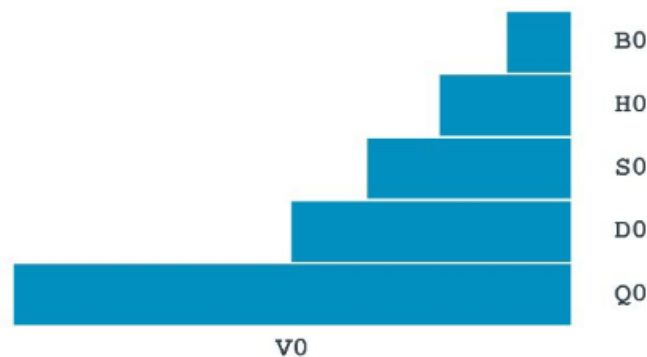


FIG. 5 : Registres pour les opérations flottantes et vectorielles

### Autres registres

Les registres ZXR et WZR lisent 0 et ignorent les écritures.

Le stack pointer SP peut être utilisé comme l'adresse de base pour les loads et les store (cf. [Adressage](#))

## Modes d'adressages

On dispose de plusieurs modes d'adressages :

- **Adressage registre** Wm ou Xm (avec m dans 0..30)
- **Adressage indirect par registre**

```
LDR X0, [X1]  // Charge la valeur située à l'adresse contenue
               // dans X1 dans le registre X0
STR X2, [X3]  // Stocke la valeur du registre X2 à l'adresse
```

```
// contenue dans X3
```

- **Adressage avec décalage** : Un décalage est ajouté à un registre pour calculer l'adresse effective. Déplacement immédiat :

```
LDR X0, [X1, #16] // Charge la valeur située à l'adresse (X1 + 16) dans X0
STR X2, [X3, #8]  // Stocke la valeur de X2 à l'adresse (X3 + 8)
```

Déplacement basé sur un autre registre :

```
LDR X0, [X1, X2] // Charge la valeur à l'adresse (X1 + X2) dans X0
```

- **Adressage pré-indexé**

```
LDR X0, [X1, #8]! // Incrémente X1 de 8, puis charge la valeur
                  // à cette nouvelle adresse dans X0
STR X2, [X3, #-4]! // Décrémente X3 de 4, puis stocke la valeur de X2
                  // à cette nouvelle adresse
```

- **Adressage post-indexé**

```
LDR X0, [X1], #8 // Charge la valeur située à l'adresse X1 dans X0
                  // puis incrémente X1 de 8
STR X2, [X3], #-4 // Stocke la valeur de X2 à l'adresse X3
                  // puis décrémente X3 de 4
```

- **Adressage relatif à PC**

L'adresse est calculée en ajoutant un décalage au compteur de programme (PC) :

```
ADR X0, label // Charge l'adresse du label dans X0
LDR X1, [X0, #4] // Charge la valeur à l'adresse (label + 4) dans X1

label:
    .word 0x12345678 // Valeur à charger
```

L'ensemble des modes d'adressages est regroupé dans ce tableau :

Type	Immediate Offset	Register Offset	Extended Register Offset
Simple register (exclusive)	[base{, #0}]	n/a	n/a
Offset	[base{, #imm}]	[base, Xm{, LSL #imm}]	[base, Wm, (S U)XTW {#imm}]
Pre-indexed	[base, #imm]!	n/a	n/a
Post-indexed	[base], #imm	n/a	n/a
PC-relative (literal) load	label	n/a	n/a

## Opérations arithmétique et logiques

Une opération arithmétique en ARMv8-A est de cette forme :

- **<operation>** : Spécifie l'opération effectuée par l'instruction. Les opérations possibles incluent :

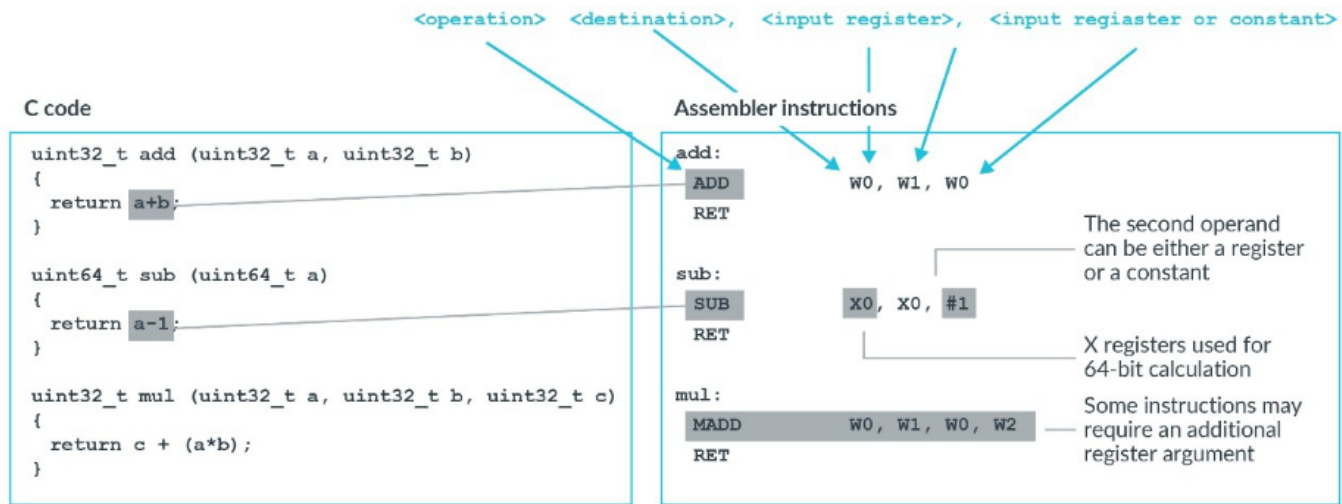


FIG. 6 : Format des opérations arithmétiques

- ADD : Addition.
- SUB : Soustraction.
- MUL : Multiplication.
- SDIV : Division avec entiers signés
- UDIV : Division avec entiers non-signés
- AND : ET logique (bitwise AND).
- ORR : OU logique (bitwise OR).

- **<destination>** : Définit le registre où sera stocké le résultat de l'opération. La destination est **toujours** un registre.
- **Opérande 1** : La première entrée de l'opération, qui est **toujours** un registre.
- **Opérande 2** : La seconde entrée de l'opération, qui peut être soit :
  - Un registre.
  - Une constante/immédiat (valeur codée directement dans l'instruction).

## Exemples

Voici quelques exemples d'instructions ARM en utilisant ce format :

```
ADD W0, W1, W2 // W0 <- W1 + W2
SUB W0, W1, W2 // W0 <- W1 - W2
MUL W0, W1, W2 // W0 <- W1 x W2
SDIV W0, W1, W2 // W0 <- W1 ÷ W2 en traitant W1 et W2 comme des entiers
    signés
UDIV W0, W1, W2 // W0 <- W1 ÷ W2 en traitant W1 et W2 comme des entiers
    non-signés
```

*Remarque* : ces instructions sont basiques et servent de base à beaucoup d'instructions dérivées comme MADD :

```
MADD W0, W1, W2, W3 // W0 <- W3 + (W1 x W2)
```

Dans ce document, nous ne présenterons pas l'ensemble des dérivations ni la totalité des instructions prises en charge par l'architecture ARMv8-A. Vous pourrez toutefois les consulter dans la documentation officielle référencée dans la bibliographie.

## Codes conditions

Les branchements conditionnels dans l'architecture ARMv8-A reposent sur l'état de certains *flags* mis à jour par les opérations dans l'ALU (*Arithmetic Logic Unit*). Ces flags sont :

- **N** : Negative
- **C** : Carry
- **V** : Overflow
- **Z** : Zero

Ces flags sont automatiquement mis à jour lorsqu'on ajoute le suffixe **S** à une instruction arithmétique ou logique (comme **SUBS** pour une soustraction).

Voici l'ensemble des codes conditions disponible dans ARMv8-A :

Encoding	Name (& alias)	Meaning (integer)	Meaning (floating point)	Flags
0000	EQ	Equal	Equal	Z==1
0001	NE	Not equal	Not equal, or unordered	Z==0
0010	HS (CS)	Unsigned higher or same (Carry set)	Greater than, equal, or unordered	C==1
0011	LO (CC)	Unsigned lower (Carry clear)	Less than	C==0
0100	MI	Minus (negative)	Less than	N==1
0101	PL	Plus (positive or zero)	Greater than, equal, or unordered	N==0
0110	VS	Overflow set	Unordered	V==1
0111	VC	Overflow clear	Ordered	V==0
1000	HI	Unsigned higher	Greater than, or unordered	C==1 && Z==0
1001	LS	Unsigned lower or same	Less than or equal	!(C==1 && Z==0)
1010	GE	Signed greater than or equal	Greater than or equal	N==V
1011	LT	Signed less than	Less than or unordered	N!=V
1100	GT	Signed greater than	Greater than	Z==0 && N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z==0 && N==V)
1110	AL	Always	Always	Any
1111	NV <sup>†</sup>	Always	Always	Any

## Exemples

```
SUBS W0, W2, #1 // W0 <- W2 - 1
```

Si le résultat de la soustraction est 0, alors le flag Z sera mis à 1 sinon il sera à 0.

Un branchement conditionnel peut ensuite être utilisé :

```
B.EQ label_equal // Branche si Z = 1 (W0 == 0)
B.NE label_not_equal // Branche si Z = 0 (W0 != 0)
```

L'instruction **CMP** pour Compare, met également à jour les flags ALU et est en réalité un alias pour **SUBS**

```
CMP W0, W1 //alias pour SUBS WZR, W0, W1
```

## Contrôle

### Branchement inconditionnel

B<label> réalise un branchement direct par rapport à PC en branchant à <label>.

### Branchement contionnel

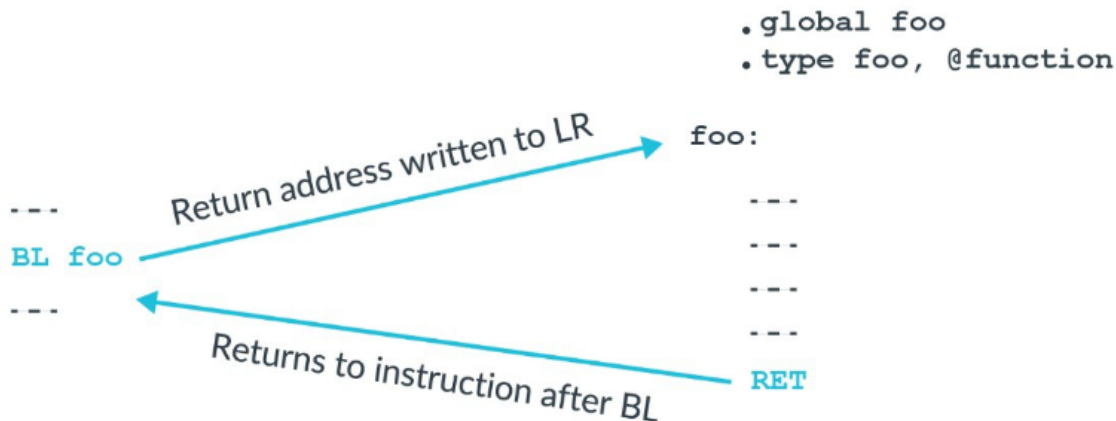
Un branchement conditionnel est de la forme B.<cond><label> et est la version condition de l'instruction B. Le branchement se fait si <cond> est true. <cond> représente un des codes conditions spécifiés plus [haut](#)

### Appels de fonctions

L'architecture ARMv8-A permet d'effectuer des appels de fonctions, qu'il s'agisse de fonctions définies par l'utilisateur ou de fonctions intégrées au langage C, comme `printf`. Lorsqu'une fonction est appelée, il est essentiel de disposer d'un mécanisme permettant de revenir à l'instruction suivante dans l'appelant une fois l'exécution de la fonction terminée.

Pour cela, les instructions de branchement simples (B ou BR) doivent être modifiées en **branchements avec lien** en ajoutant le suffixe L (ce qui donne BL ou BLR). Cette modification permet de sauvegarder l'adresse de retour (celle de l'instruction suivante après l'appel) dans le registre dédié à cet usage, le registre X30, également connu sous le nom de Link Register (LR).

À la fin de la fonction, l'instruction RET doit être utilisée pour effectuer un branchement indirect vers l'adresse de retour contenue dans le registre X30. Cela garantit un retour correct à l'appelant et la poursuite normale de l'exécution du programme.



## Mise en place de l'environnement

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

### 1. Cross-compilation avec `aarch64-linux-gnu-gcc` :

- Le compilateur `arm-linux-gnueabi-hf-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'exécution 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.

# Conception

## Compilation avec CLANG et GCC

Dans le cadre de notre projet, l'objectif principal était de compiler pour une cible ARM Cortex-A53 (architecture ARMv8). Le code assembleur généré devait ensuite être lié et assemblé à l'aide de GCC. Cependant, nous avons rencontré plusieurs limitations avec l'utilisation de GCC sur macOS. Ces problèmes nous ont conduits à privilégier l'usage de Clang, qui s'intègre mieux dans cet environnement.

Toutefois, cette transition n'a pas été sans difficultés : les spécificités du code assembleur ARM diffèrent légèrement entre les environnements utilisant Clang et ceux utilisant GCC. Afin de pallier ces divergences et de maximiser la compatibilité, nous avons décidé d'implémenter une option permettant de compiler directement pour un Mac équipé d'une puce Apple M2 (basée sur ARM64). Le code généré dans ce cas devrait également être compatible avec d'autres Macs utilisant des processeurs ARM64. La compatibilité peut être vérifiée via la commande suivante :

```
uname -m
```

## Commandes pour la Compilation

Pour générer un fichier compilé destiné à un Mac avec une puce M2, la commande à exécuter est la suivante :

```
decac --arm --M2 programme.s
```

En revanche, pour une cible utilisant GCC (par exemple, un environnement Linux avec une architecture ARMv8), la commande standard est :

```
decac --arm programme.s
```

Une fois le fichier assembleur .s généré, il peut être transformé en un exécutable à l'aide de Clang avec la commande suivante :

```
clang -arch arm64 -o executable programme.s
```

## Différences entre Clang et GCC dans le Contexte de Deca

Dans le cadre de notre compilateur Deca sans objet, deux différences notables entre Clang et GCC ont nécessité des ajustements spécifiques :

### 1. Appels de fonctions (noms des symboles)

- Avec **Clang**, les noms des fonctions nécessitent un préfixe par un underscore (`_`). Par exemple, les fonctions `main` et `printf` doivent être référencées respectivement en tant que `_main` et `_printf`.
- Avec **GCC**, ce préfixe n'est pas requis. Les mêmes fonctions sont appelées directement par leurs noms (`main`, `printf`).

### 2. Gestion des paramètres pour la fonction `printf`

- Dans un environnement Clang, les paramètres de la fonction `printf` doivent être placés sur la **pile** (via le registre **SP**, **Stack Pointer**).
- Avec GCC, les paramètres sont au contraire passés directement dans les registres dédiés, en commençant par les registres `x1`, `x2`, etc.

Dans les deux environnements, l'adresse de la chaîne de caractères utilisée dans l'appel à `printf` est systématiquement stockée dans le registre `x0`. Cette uniformité facilite la gestion de cette partie du code, mais les ajustements précédemment évoqués doivent être pris en compte pour assurer une compatibilité complète entre les deux compilateurs.

## Manières de procéder

Pour développer un compilateur ciblant l'architecture ARM64 pour un langage deca sans objet, nous avons adopté une approche méthodique et progressive. Cette démarche nous a permis de limiter les difficultés rencontrées à chaque étape tout en assurant une compréhension approfondie des concepts nécessaires à la génération de code de qualité.

### Étapes initiales : gestion des variables entières

Nous avons débuté par l'implémentation des fonctionnalités de base, à savoir la déclaration et l'initialisation de variables entières, suivies de leur affichage. Cette approche visait à simplifier les tests initiaux et à poser les fondations nécessaires pour les étapes suivantes. La mise en place de ces fonctionnalités a permis de valider la génération d'instructions assembleur élémentaires et d'assurer une bonne gestion des registres.

### Ajout des opérations arithmétiques et booléennes

Une fois les bases posées, nous avons intégré les opérations arithmétiques (addition, soustraction, multiplication et division) ainsi que les opérations booléennes (AND, OR, NOT, etc.). Ces ajouts ont nécessité une gestion plus complexe des registres et une meilleure compréhension du jeu d'instructions ARMv8.

### Gestion des branchements conditionnels

Les branchements conditionnels ont été implémentés dans une étape ultérieure. Cette partie impliquait la manipulation directe des drapeaux du processeur ARM et l'utilisation d'instructions telles que `CSET`, `SUBS` ou `TBNZ`. Ces instructions nous ont permis de structurer les blocs conditionnels `if-then-else` et d'implémenter des structures comme les boucles `while`.

## Prise en charge des flottants

L'ajout de la prise en charge des nombres flottants a constitué un défi majeur. Contrairement aux entiers, les flottants nécessitent l'utilisation des registres spécifiques et des instructions dédiées, comme **FADD** (addition flottante) ou **FSUB** (soustraction flottante). On a aussi dû adapter notre code afin qu'il puisse s'adapter à des variables de différentes tailles. Cet ajout nous a donc forcé à revoir une grande partie de notre code.

## Documentation et inspirations

Pour garantir une implémentation correcte, nous nous sommes inspirés des compilateurs GCC et Clang. Nous avons étudié la manière dont ces outils génèrent du code assembleur pour ARM64 en analysant les fichiers assembleurs produits à partir de programmes simples. Cela nous a permis de comprendre les bonnes pratiques en matière de gestion des registres, d'utilisation de la pile, et de respect des conventions ARM64.

## Utilisation de la libc

Afin de simplifier l'implémentation des fonctions d'affichage (comme la fonction **println**), nous avons décidé de nous appuyer sur la libc, une bibliothèque standard qui fournit des fonctionnalités essentielles. En utilisant la libc, nous avons pu accéder facilement à des fonctions comme **printf** pour afficher des variables entières ou flottantes. Cette approche présente plusieurs avantages :

- Réduction de la complexité du compilateur, en déléguant certaines tâches à une bibliothèque fiable et optimisée.
- Compatibilité accrue avec les environnements standards.

## Implémentation

Nous allons maintenant examiner la manière dont certaines parties du compilateur pour ARM ont été implémentées. Celui-ci s'inspire fortement du compilateur pour IMA, en réutilisant des principes similaires, notamment l'appel de méthode **codeGen** et des structures analogues. Par exemple, on retrouve des éléments tels que **ARMProgram**, **ARMDVal**, ou encore certaines instructions comme **ARMStore** et **ARMLoad**. Cependant, nous avons simplifié l'architecture en réduisant significativement le nombre de classes, en regroupant les instructions dans une seule classe : **ARMInstruction**.

Le cœur du compilateur réside dans la classe **ARMProgram**, qui se charge de générer les lignes de code ARM64. Cette classe gère également les registres sous forme de pile ainsi que la gestion de la mémoire.

## Gestion de la mémoire

La gestion de la mémoire est en réalité une gestion des offsets dans la pile. Chaque variable se voit attribuer un offset après la génération des instructions, ce qui permet à un même offset d'être partagé par plusieurs variables si elles ne sont pas utilisées simultanément.

Nous utilisons le registre **X29** pour stocker les variables, tandis que le registre **SP** est réservé à d'autres tâches, comme l'appel à **printf**. Pour déterminer la taille de la pile et bien l'initialiser, nous prenons en compte le nombre maximum de variables utilisées simultanément ainsi que le nombre de paramètres passés à **printf**. Enfin, la taille de la pile est arrondie à la puissance de deux la plus proche.



## Mise en place de la pile et des normes

La génération des lignes de début et de fin de code inclut l'initialisation de la pile, la sauvegarde des registres, ainsi que le respect de certaines normes, telles que `.cfi_startproc` et `.cfi_endproc`.

## Méthodes utilitaires

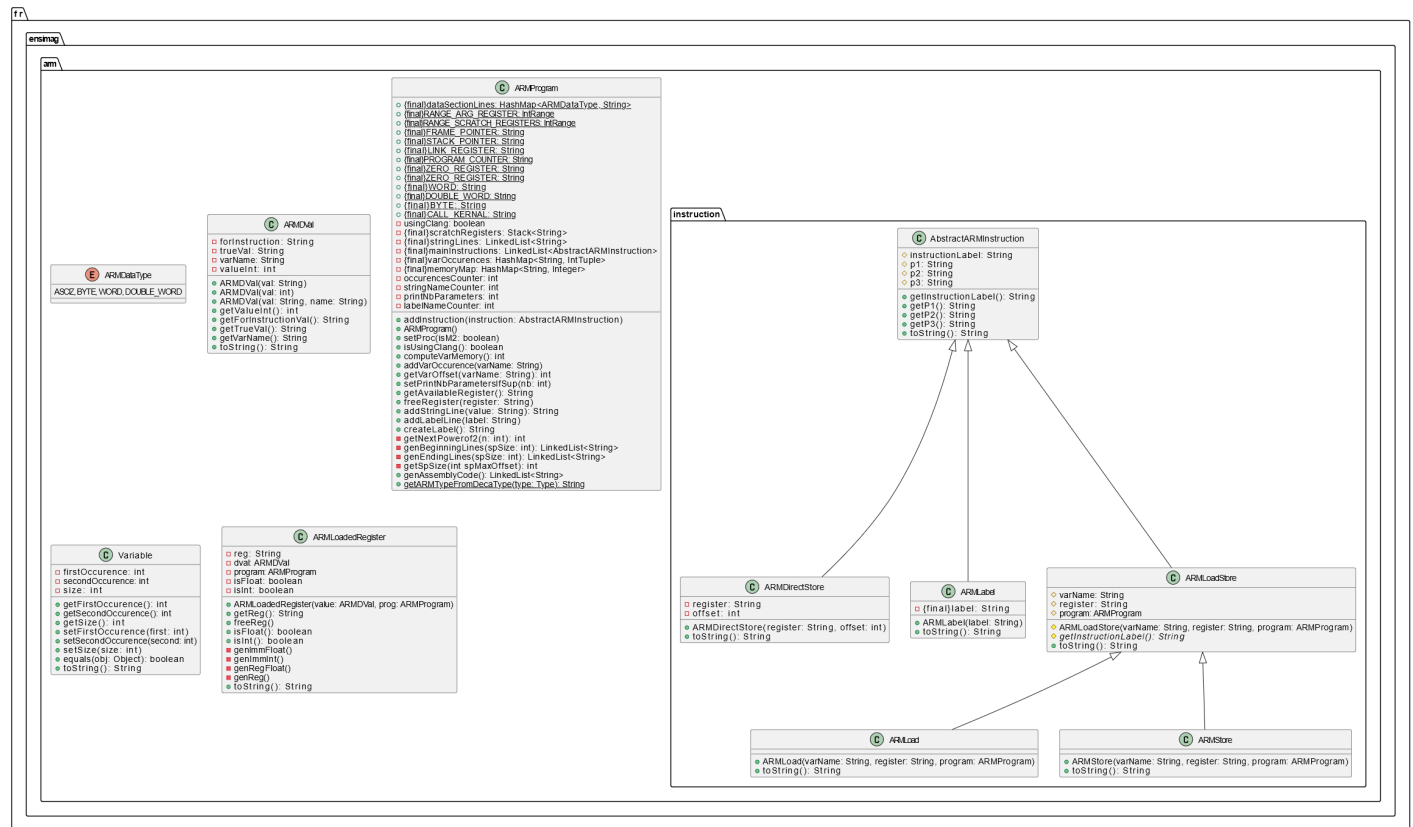
En plus de ces fonctionnalités, `ARMProgram` propose diverses méthodes utilitaires comme `getReadyRegister`, pour faciliter la gestion des registres.

## Mise en place de tests

Nous avons implémenté le test `verify-arm-output` qui utilise les tests DECA, et qui permet de passer à un test suivant lorsque l'un d'eux échoue. Ce mécanisme garantit des exécutions ordonnées et une meilleure gestion des erreurs tout au long du projet.

## UML

Voici un diagramme UML représentant les différentes classes utilisées pour cette extension :



## Résultats obtenus

Notre compilateur ARM est capable de compiler un sous-ensemble du langage DECA sans objets, comprenant les fonctionnalités suivantes :

- Déclaration de variables
- Opérations arithmétiques et booléennes
- Affichage avec `print`

- Branches conditionnelles

Le tout avec un bon niveau de robustesse.

Nous avons également implémenté un début de prise en charge des flottants, bien que cette fonctionnalité comporte encore plusieurs bugs. Néanmoins, elle permet de déclarer des flottants, d'effectuer des opérations arithmétiques avec ces derniers, ainsi que de réaliser des branchements conditionnels avec une robustesse correcte.

En revanche, l'implémentation de la fonction `read` n'a pas été réalisée.

Les problèmes rencontrés ont principalement découlé d'un manque d'abstraction dans le code, ce qui a rendu celui-ci difficile à maintenir et a complexifié l'ajout de nouvelles fonctionnalités. Les abstractions ont été introduites trop tard dans le développement, ce qui a engendré des ralentissements notables, notamment lors de la gestion des flottants. Les modifications fréquentes sur des instructions telles que `mov` et `fmov` ont exigé de repenser plusieurs parties du code. Cette situation nous a permis de réaliser l'importance d'une conception soignée dès le départ, pour éviter les problèmes à long terme et améliorer la maintenabilité du code.

Nous en avons tiré la leçon qu'il est crucial de prendre le temps nécessaire pour bien structurer le projet, notamment en introduisant des classes et des abstractions dès le début, comme cela a été fait dans le compilateur IMA. Une bonne planification et une gestion rigoureuse des abstractions permettent de mieux gérer l'évolution du code et de minimiser les difficultés liées aux changements futurs.

## Conclusion

### Conclusion

Cet extension du compilateur pour l'architecture ARM Cortex-A53 a permis d'explorer de nombreuses facettes du développement de compilateurs, notamment la gestion de la mémoire, l'optimisation des registres et la compatibilité entre différents environnements de compilation (Clang et GCC). Grâce à l'intégration progressive de fonctionnalités telles que la gestion des variables entières, les opérations arithmétiques, les branchements conditionnels et les flottants, nous avons pu tester et valider l'évolution du compilateur tout en identifiant les difficultés rencontrées à chaque étape.

Ce projet nous a permis de tirer des leçons sur l'importance de la planification, de la gestion des abstractions et de la maintenance du code dans un projet à long terme. En tirant parti des bonnes pratiques des compilateurs existants et en prenant soin de structurer le code de manière évolutive, nous avons pu développer une extension robuste.