

# Documentation de l'extension ARM

Projet GL

# Table des matières

<b>1</b>	<b>Spécification de l'extension</b>	<b>3</b>
1.1	Résumé	3
1.2	Description du target	3
<b>2</b>	<b>Analyse bibliographique</b>	<b>4</b>
<b>3</b>	<b>Choix de conception</b>	<b>4</b>
<b>4</b>	<b>Choix d'architecture</b>	<b>4</b>

# 1 Spécification de l'extension [10 Janvier 2022]

## 1.1 Aperçu

La spécification Deca décrit le langage et son implantation pour une machine abstraite IMA.

Ce document étend le compilateur pour compiler en ARM tout le langage Deca. Nous proposons deux spécifications:

- `arm-sans-objet`: spec pour le langage sans objet
- `arm-complet`: spec pour le langage complet

### 1.1.1 Buts

Pour le compilateur decac:

1. Gérer deux architectures (double back-end), tout en gardant une conception flexible qui permet d'ajouter d'autres architectures
2. Émettre du code adapté pour une machine ciblée qui permet le débogage et l'affichage d'erreur

Pour le langage Deca:

3. Avoir des résultats uniformes entre l'exécution d'un programme IMA et un programme ARM

### 1.1.2 Demandes du produit

- Une option `"-a"`, qui, quand utilisée, génère du code ARM pour la machine ciblée.

### 1.1.3 Suppositions

- Nous devons pouvoir étendre le compilateur dans le futur avec d'autres architectures.
- Nous ne nous soutiendrons que le target<sup>1</sup> ARM Cortex-A15, qui possède une architecture ARMv7-A 32 bits, de la mémoire flash et une RTC.
- Nous supposons s'exécuter sous un système Linux<sup>2</sup>.
- Pour la spécification `arm-complet`: Nous supposons avoir accès à un allocateur<sup>3</sup> et un ramasse-miette<sup>4</sup>.

---

<sup>1</sup> *target*: Machine ciblée par notre compilateur.

<sup>2</sup> Le programme compilé par decac sera le seul à s'exécuter sur la machine.

<sup>3</sup> *allocateur*: Bibliothèque pour la mémoire dynamique: implémente `malloc()` et `free()`.

<sup>4</sup> *ramasse-miette*: Bibliothèque permettant de libérer la mémoire allouée sans `free()`.

## 1.2 Spécification formelle

### 1.2.1 Spécification de arm-sans-objet

Le compilateur Decac doit implémenter l'option `-a`.

Tout programme Deca valide doit être compilé, avec l'option `-a`, en un programme assembleur `.s` compilable par l'assembleur GNU en code machine ARM, dont le comportement respecte la sémantique du programme décrite dans [Sémantique].

En mode `-a`, le compilateur decac doit accepter toutes les options spécifiées dans [Decac] sauf l'option `-r`. L'implémentation peut ajouter ses propres options spécifiques fonctionnant uniquement avec le mode `-a`.

Tout assembleur ARM généré doit être exécutable par une machine ARMv7-A 32 bits Cortex-A15.

Tout programme compilé en ARM doit, à l'exécution, afficher le même résultat qu'un programme compilé en assembleur machine abstraite<sup>5</sup>.

L'environnement d'exécution doit fournir un mécanisme de débogage. Le choix du mécanisme à mettre en œuvre et ses détails sont laissés à l'implémentation.

L'environnement d'exécution doit fournir un mécanisme pour afficher les erreurs à l'exécution spécifique à Déca. L'implémentation doit gérer toutes les erreurs décrites dans le manuel utilisateur. Le format des messages d'erreur est laissé à l'implémentation: en particulier, les messages d'erreur peuvent être différents des messages de l'implémentation ima.

### 1.2.2 Spécification de arm-complet

Le compilateur doit respecter la spécification `arm-sans-objet`.

L'environnement d'exécution doit fournir une implémentation des fonctions `malloc()`, `realloc()` et `free()`. Ces fonctions doivent allouer de la mémoire dynamiquement, réallouer de la mémoire allouée dynamiquement et libérer la mémoire allouée.

L'environnement d'exécution doit fournir un mécanisme de ramasse-miette et une fonction `gc_malloc()` permettant d'allouer de la mémoire par le ramasse-miette. L'implémentation doit garantir la libération de la mémoire non référencée.

---

<sup>5</sup> *assembleur machine abstraite*: L'assembleur généré par le compilateur sans option `-a`, exécutable par la machine abstraite et par son implémentation de référence ima.

## 2 Analyse bibliographique

- [ARM Developer](#)
- [QEMU](#)
- [Implémentation de ramasse-miette](#)
- TODO: a completer

## 3 Choix de conception

### 3.1 Composants

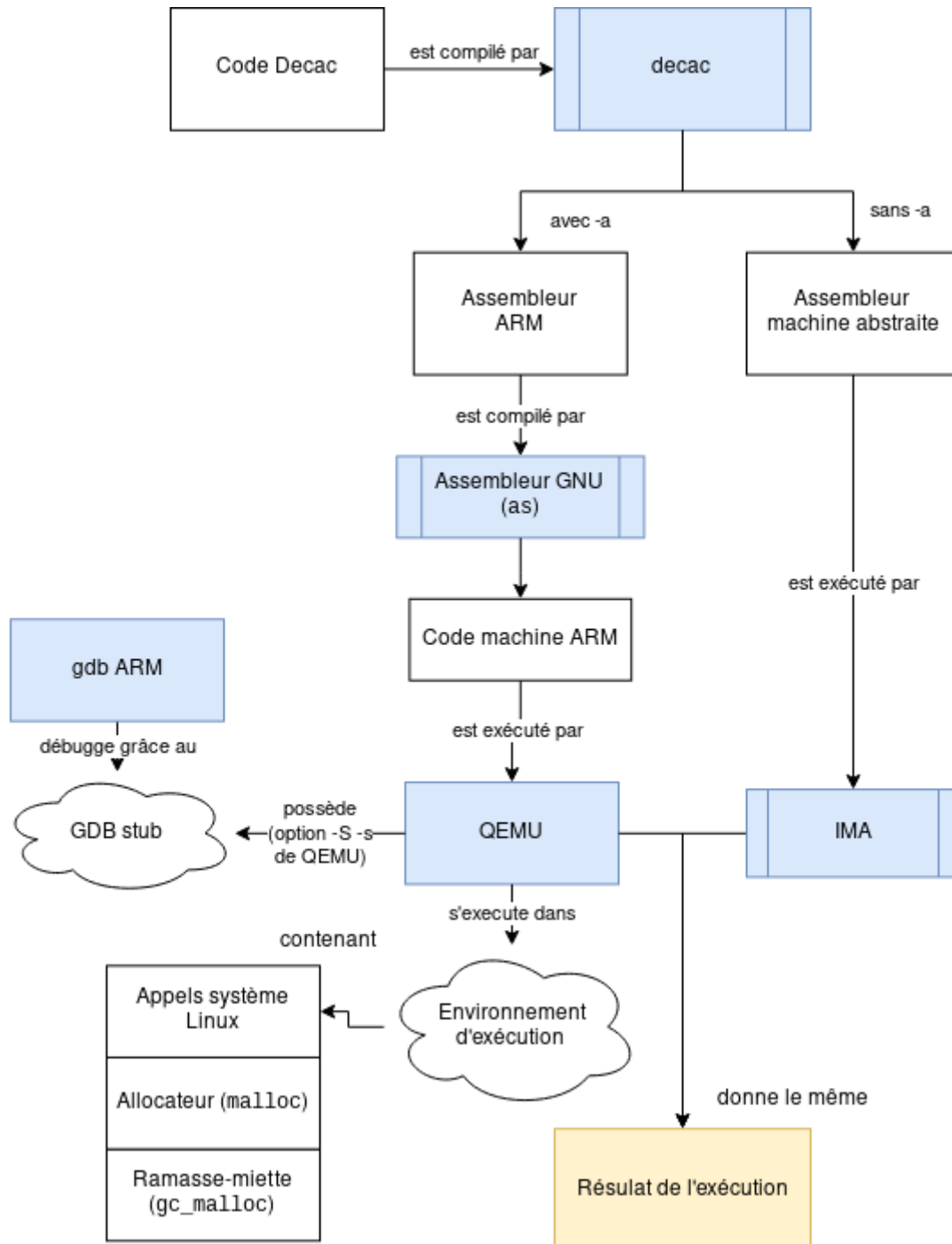


Figure 1: Descriptions des différents composants de l'extension ARM

## 3.2 Approche

### 3.2.1 Choix du target

Nous ciblons un processeur ARMv7-A. La ligne de processeurs "A" sont spécialisés dans le lancement des applications; en contraste avec "R" pour les processeurs Real-Time et "M" pour Microcontroller.

Beaucoup de nos choix sur la conception et l'exécution se sont fait sur cette supposition. Cette extension est adaptée pour les développeurs d'application, mais serait moins adaptée pour par exemple un micro-contrôleur qui a des demandes beaucoup plus lourdes et ne pourrait pas faire tourner Linux.

### 3.2.2 Spécification de l'environnement d'exécution

Nous avons d'abord considéré: comment exécuter le code? Il nous fallait une machine virtuelle, mais surtout un environnement d'exécution. Si nous faisons tourner nos programmes sur le processeur directement, nous n'aurions pas de moyen d'afficher les résultats, de déboguer, et de pouvoir attraper les erreurs. Il a donc fallu gérer ces points indépendamment.

#### 3.2.2.1 Gestion des affichages

Notre première approche a été d'utiliser un système Linux avec la bibliothèque standard C glibc, qui nous donnait la fonction `puts()`. Après des premiers tests, nous nous sommes rendus compte que cette approche rendait le binaire lourd.

Nous nous sommes ensuite rendu compte que nous pouvions utiliser les appels système Linux sous-jacents: `write()` pour écrire et `read()` pour lire, nous débarrassant ainsi de la libc.

La spécification de Linux est un choix contraignant pour l'environnement d'exécution. Nous l'avons fait dans le but de cibler des applications.

#### 3.2.2.2 Gestion des new pour arm-complet

Pour cette spécification, il faut implémenter l'équivalent de l'instruction `NEW` en assembleur machine abstraite pour allouer l'espace mémoire pour des objets. Nous pensions qu'il y aurait encore besoin de la libc pour faire cette instruction. Après des recherches, nous avons vu que nous pouvions utiliser notre propre allocateur et éviter la libc.

#### 3.2.2.3 Choix d'un garbage collector

Une fois que nous allouons de la mémoire, il faut la libérer, sous peine de subir des fuites de mémoire. Trois choix s'offrent à nous:

- libérer la mémoire explicitement (C, C++)

- implémenter un *borrow checker*, qui permet de suivre les possessions mémoire à la compilation, insérant pour nous des `free()` quand la mémoire n'est plus utilisée (Rust)
- utiliser un ramasse-miette (Java, Python)

La première option nécessiterait de changer le langage Deca, ce qui violerait la spécification initiale, et nous forcerait à maintenir une extension uniquement pour ARM. Nous avons rapidement écarté ce choix.

L'option du borrow checker semble attirante mais demande un travail d'implémentation sur le compilateur trop grand pour tout, par rapport au travail que nous pouvons fournir, et nous l'avons également écarté.

Nous avons donc choisi un ramasse-miette. Ce mécanisme peut être intégré à l'environnement d'exécution facilement, ne pose pas de considérations spéciales sur le compilateur, qui peut gérer les allocations comme les NEW pour l'assembleur machine abstraite, mais pose des demandes plus contraignantes sur l'environnement.

### 3.2.3 Choix de l'émulation de la machine ARM

Pour l'environnement d'exécution qui émule une machine ARM nous avons choisi QEMU, qui soutient l'architecture ARM, peut être exécuté en mode user (ce qu'il) et soutient le débogage nativement.

### 3.2.4 Toolchain de compilation et d'exécution

Pour vérifier que nous pouvions compiler du code ARM et l'exécuter, nous avons utilisé la [GNU Toolchain](#) de ARM. Cette toolchain nous fournit notamment l'assembleur GNU, qui nous permet dans un premier temps de pouvoir tester le code sans passer par notre compilateur.

### 3.2.5 Débogage du code ARM

Pour pouvoir déboguer, nous utilisons le GNU Debugger (gdb). Ceci a été notre premier choix, car il est simple à intégrer dans l'environnement tout en étant très complet. Un GDB pour notre target est fourni par la toolchain ARM.

Pour pouvoir lancer le débogueur, du côté client, gdb a besoin d'un [gdb stub](#), du code implémenté sur le target permettant la communication par port série et l'activation de gdb.

Le choix de QEMU nous a facilité la tâche: QEMU contient une [option -s -S](#) activant un GDB stub intégré.




## 4 Choix d'architecture

Puisque la class DecacCompiler a été conçue pour générer un IMAProgram et donc une représentation de programme assembleur bien spécifique, un refactoring était nécessaire pour découpler les deux.

### 4.1 L'interface OutputProgram

Son rôle consiste à relier le compilateur decac et un générateur de code qui implémente OutputProgram. On a compris que la classe DecacCompiler n'a besoin d'appeler que des méthodes d'affichage sur OutputProgram.



```
public interface OutputProgram {  
    /**  
     * @param s Stream for printing the program's text representation.  
     */  
    void display(PrintStream s);  
    /**  
     * Return the program in a textual form readable as a String.  
     */  
    String display();  
    /**  
     * @param comment Assembly language-agnostic way of writing comments.  
     */  
    void addComment(String comment);  
}
```

La génération de code consiste à appeler une méthode sur les éléments de l'arbre de syntaxe abstraite qui se charge d'ajouter les instructions correspondantes dans OutputProgram. La manière avec laquelle OutputProgram est construit n'est restreinte par aucune interface, puisque les programmes IMA et assembleur ARM ont des structures bien différentes (par exemple IMA ne présente pas de sections).

## 4.2 L'interface CodeGen

```
public interface CodeGen {  
    /**  
     * This method inserts the relevant assembly code in the program.  
     *  
     * @param program Abstract representation of the IMA assembly code.  
     */  
    void codeGen(IMAProgram program);  
    /**  
     * This method inserts the relevant assembly code in the program.  
     *  
     * @param program Abstract representation of the ARM assembly code.  
     */  
    void codeGen(ARMProgram program);  
}
```

Ceci permet d'énumérer les bakends supportés par le compilateur sous forme de méthodes prenant un type différent de `OutputProgram`. On a prévu de rendre l'interface `CodeGen` polymorphique par rapport à `OutputProgram`, comme ça le langage Java aurait enforcé le fait que `IMAProgram` et `ARMProgram` soit des `OutputProgram`, avec une seule méthode qui prend en paramètre un `OutputProgram`.

Malheureusement, Java considère que deux interfaces polymorphiques (instanciés avec des types différents) sont en fait le même type.

## 4.3 Pseudocode ARM

On a ajouté le package `arm.pseudocode` pour contenir les classes aidant la construction dynamique de code assembleur ARM en s'inspirant de la structure existante pour IMA, avec quelque modification. On prévoit un factoring des fonctionnalités communes entre les deux une fois les APIs se stabilisent.

Une fonctionnalité spéciale au package `arm.pseudocode` c'est la classe `Syscall`; elle permet la construction d'appels système génériques se modélisant comme des listes d'Operand.

# 5 Résultats de la validation de l'extension