Documentation de l'extension ARM

Projet GL

ELHAJ-LAHSEN Hugo
GONDET Matthias
LAVAL Jean
MAZOUZ Mahmoud
NADJA Mohamad

GRENOBLE INP UGA Ensimag

Table des matières

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[1] 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[2].
- Pour la spécification `arm-complet` : Nous supposons avoir accès à un allocateur[3].

[1] *target:* Machine ciblée par notre compilateur.
[2] Le programme compilé par decac sera le seul à s'exécuter sur la machine.
[3] *allocateur*: Bibliothèque pour la mémoire dynamique: implémente *malloc()* et *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].

Tout programme dans le sous-langage Deca[4] sans objet est accepté. Si un programme avec objet est compilé avec l'option `-a`, le comportement n'est pas spécifié.

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[5].

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, affichant toutes les erreurs décrites dans le manuel utilisateur.

L'environnement d'exécution doit fournir une implémentation de la fonction `printf()`.

1.2.2 Spécification de arm-complet

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

Tout le langage Deca est accepté.

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.

---

[4] *sous-langage*: un ensemble restreint du langage Deca. Le sous-langage sans objet représente Deca sans les classes.
[5] *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

- On Writing Tech Specs (Codeburst).

Ce document a été l'inspiration principale pour la structure de cette spécification. et nous suivons d'assez près leur structure de spec.

- ARM Developer

Source de référence pour ARM. Nous avons utilisé principalement leur référence pour le processeur ARMv7 et le FPU.

- Demystifying ARM Floating Point Compiler Options (Embedded Artistry)

Source simple pour comprendre le problème des flottants.

- Documentation QEMU

En particulier, nous avons utilisé la documentation QEMU user mode.

- GNU Toolchain (ARM)

Une toolchain de ARM que nous utilisons pour générer le code machine.

- Options ARM de GCC

Autre documentation utile pour voir les architectures disponibles ainsi que les options des flottants.

3 Choix de conception

3.1 Composants

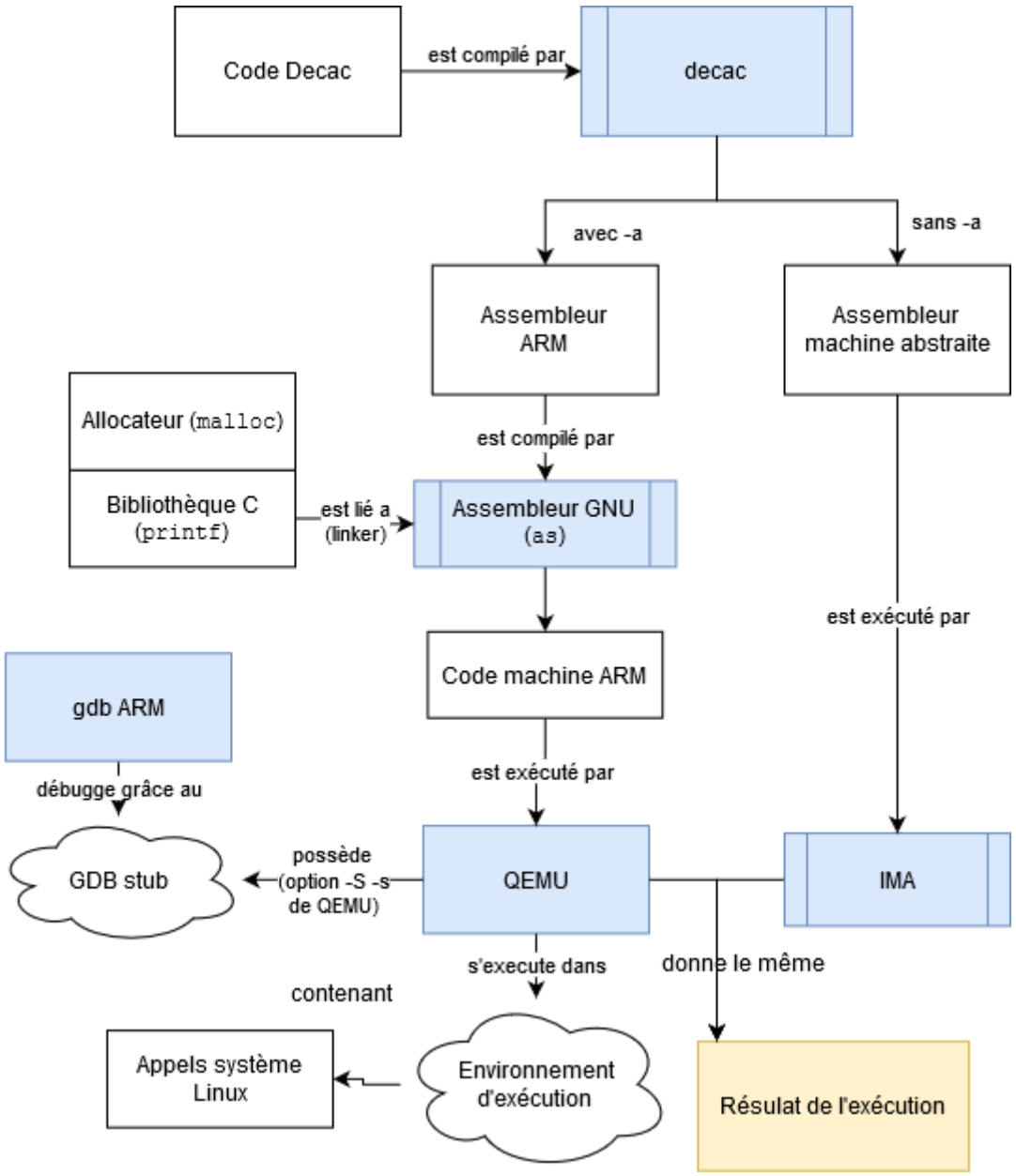

Figure 1: Descriptions des différents composants de l'extension ARM

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

Par rapport aux affichages, un point important s’est relevé lors de l’analyse: afficher les entiers et en particulier les flottants est compliqué. En effet, ARM ne possède pas de `WINT/WFLOAT` comme en assembleur machine abstraite: il nous faut gérer ces affichages.

Pour gérer les affichages, nous pouvions:

- écrire en assembleur tous les affichages

Cette option nous demande d’écrire du code pour gérer l’affichage du tty. De plus, nous n'avons accès à aucun formatage, sauf celui que nous écrivons nous même.

- utiliser les appels noyaux Linux `write()`
- utiliser Linux + la bibliothèque C pour `printf()`

	Noyau Linux / `write()`	**Linux + biblio C /** `printf()`
Avantages	• Binaire moins lourd, pas de libc linkée • Convention d’appel simple à respecter (syscall)	• Options de formatage intégrées
Inconvénients	• Pas d’options de formatage → écriture de gestion des flottant en assembleur	• Binaire plus lourd • Appels système plus compliqués • Respect passage flottant

La deuxième option, utiliser la bibliothèque C, malgré ses inconvénients, possède un gros avantage: nous évitons d'écrire notre propre code pour la gestion des flottants, et pouvons à la place utiliser du code testé au combat.

Cet avantage pour nous compensait tous les défauts, et nous avons donc fait le choix de la bibliothèque C.

3.2.2.2 Gestion des new pour arm-complet

Ayant choisi la bibliothèque du C, nous avons accès à un allocateur avec peu d'effort.

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) 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.

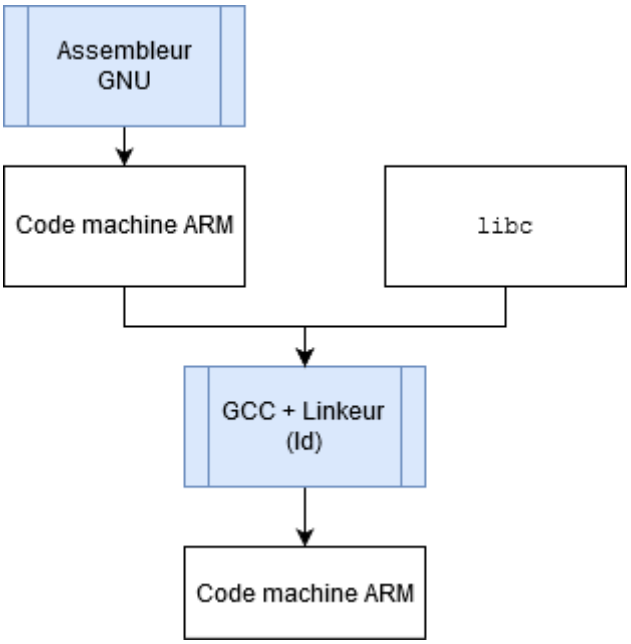

Figure 2: Détail de l'étape de compilation, après la sortie de l'assembleur

3.2.5 Gestion des flottants et compatibilité ABI

Notre compilateur doit également soutenir les opérations arithmétiques sur les flottants tout en respectant la sémantique de Deca: il doit s'arrêter quand des erreurs de calcul sur les flottants sont faites.

Or, ARM ne contient pas nativement des instructions sur les flottants. Il faut que le processeur possède un co-processeur Floating Point Unit (FPU). Notre target possède une FPU respectant la version VFPv4.

Du côté de GCC, vu que l'on compile du code de la libc avec notre binaire, il faut que les flottants soient compatibles.

Le compilateur de la GNU Toolchain contient deux options intéressantes[6] -mfloat-abi=<abi> et -mfpu=<flu>. L'option -mfpu permet de définir le FPU utilisé, que nous mettons à vfpv4. La *float-abi* est l'ABI[7] utilisé par le compilateur.

Dans notre chaîne de compilation, nous devons faire l'édition de liens[8] entre notre programme et la libc, pour pouvoir utiliser le `printf()`. Ceci nous force à avoir la même ABI pour notre programme et la libc. Trois options s'offrent à nous:

- soft: Pas d'instructions flottantes sont générées. Le compilateur utilise des appels de fonctions pour calculer les flottants.
- softfp: Garde les conventions d'appels de soft, mais utilise les instructions flottantes.
- hard: Utilise les conventions d'appel spécifiques au FPU.

Nous avons utilisé l'option hard qui coïncidait avec la libc.

3.2.6 Débogage du code ARM

Pour pouvoir débugger, 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 à 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.

[6] Pour comprendre les options point-flottant de GCC, nous avons utilisé principalement cette source: Demystifying ARM Floating Point Compiler Options (Embedded Artistry)
[7] *ABI: Application Binary Interface:* une interface bas niveaux entre les binaires. Ils définissent notamment pour nous les instructions, la convention d'appel, et les types de données que nous pouvons utiliser.
[8] *link:* Étape de la compilation d'un programme ou on crée un fichier exécutable à partir de fichiers objet.

4 Choix d'architecture

La classe DecacCompiler a été conçue pour générer un IMAProgram et donc une représentation de programme assembleur bien spécifique. Un refactoring est 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.

```java
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.

Nous avons découvert que les programmes IMA et assembleur ARM ont des structures bien différentes:

`.text`
`.global _start`
`_start:`
`    mov r0, #1`
`    ldr r1, =message`
`    ldr r2, =len`
`    mov r7, #4`
`    swi 0`
	`WSTR “Hello, world!”`
	`HALT`

```
    mov r7, #1
    swi 0

.data
message:
    .asciz "Hello, world!\n"
len = .-message
```

Dans le langage assembleur GNU, on a des symboles, des sections avec des significations spécifiques, et les instructions ARM sous l'assembleur GNU possèdent également une syntaxe différente, avec trois arguments, ou un nombre variable d'arguments.

Pour cette raison, la manière avec laquelle OutputProgram est construit n'est restreinte par aucune interface.

4.2 L'interface CodeGen

```java
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 backends 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 imposé que IMAPorgram 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 quelques modifications. 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 modélisent comme des listes d'Operand.

5 Résultats de la validation de l'extension

Notre extension porte uniquement sur la partie C du compilateur. Ainsi, nous avons pu réutiliser les tests codegen du compilateur, cette fois-ci sur la partie ARM. Chaque test codegen est lancé deux fois:

- une fois en IMA: compilé sans option -a et lancé par ima.
- une fois en ARM: compilé avec option -a et lancé par ./arm-env.sh run.

Nous nous attendions, pour chaque test, à avoir le même résultat pour les tests codegen pour IMA et pour ARM. Ce choix nous force, entre autres, à respecter exactement la sortie décrite dans la spécification et de choisir les mêmes messages d'erreur: notre système de test utilise un oracle qui vérifie le test à une trace.

Ces tests sont intégrés au `mvn test`. Nous voulions également l'intégrer à notre CI sur GitLab, mais nous avons rencontré un problème avec QEMU qui ne s'exécutent pas sous le docker de GitLab que nous n'avons pas réussi à résoudre. Les tests ARM doivent se lancer sur la machine par le développeur.

6 Documentation développeur

5 Mise en place

Mettre en place la toolchain: ./arm-env.sh install

6 Exemple de compilation

- Compiler son fichier avec l'option -a. decac -a hello.deca
- Lancer avec la toolchain: ./arm-env.sh run hello.s