DOCUMENTATION DE L'EXTENSION ARM

*Projet Génie Logiciel - **Equipe GL 23***

Boukhriss Nada / Dakhil Yahya / Elhjouji Salah /Et-tarraf Zineb / Tamouh Alae

JANVIER 2022

SOMMAIRE

1. Motivation

- Compilation vers machine “concrète”.
- Architecture d’ensembles d’instructions la plus répandue dans le monde: des smartphones et tablettes aux serveurs et systèmes embarqués.
- Application des acquis en assembleur RISC-v en 1a.
- Inspiration de la génération de code d’IMA pour générer du ARM 32-bits.

2. Pistes de réflexion initiales

1. Essayer de refaire la chaîne de compilation de Deca vers ARM, mais c’est non seulement comparable à faire un Projet GL additionnel, mais il est aussi complètement indiqué qu’il ne faudra que dupliquer la génération de code dans la dernière étape.
2. Sachant qu’il existe un compilateur de Deca vers LLVM, chercher une toolchain de compilation LLVM to ARM. Ici aussi, on doit refaire le travail de compilation deca avec une syntaxe LLVM qui est du travail refait.
3. Utiliser l’arbre syntaxique abstrait déjà généré à l’issue de la vérification contextuelle et le compiler directement en assembleur ARM. C’est la piste la plus raisonnable qui comporte le moins de travail ajouté et c’est celle qu’on a finalement choisi de suivre. On a opté pour une exécution simulée des exécutables ARM sur Ubuntu en utilisant qemu-user (voir la section suivante) plutôt que de compiler en “dur” et exécuter sur un des processeurs supportant ARM.

3. Guide de travail en mode ARM:

a. Installation de la toolchain

L'extension ARM porte sur la compilation de l'arbre syntaxique abstrait issu du programme Deca en code assembleur ARM aarch32. Pour cela, il faudra avoir en main la toolchain nécessaire; on a choisi de travailler avec la librairie GNU ARM-32bits qu'on installera dans un environnement Debian (à adapter pour les autres environnements Linux) en suivant les étapes ci-dessous. Les commandes shell sont précédées d'un $:

- S'assurer d'avoir l'accès root sur le profil utilisé pour l'installation de la toolchain.
- Mettre à jour le gestionnaire de paquets apt:
  $ sudo apt update -y && sudo apt upgrade -y

- Télécharger et installer les librairies et utilitaires GNU:
  $ sudo apt install qemu-user qemu-user-static build-essential
  $ sudo apt install gcc-arm-linux-gnueabihf binutils-arm-linux-gnueabihf binutils-arm-linux-gnueabihf-dbg

b. Routine de compilation et d'exécution

- Pour générer un fichier ARM assembly .s:
  - $ decac -arm hello.deca

- Toutes les compilations des fichiers.s se font en mode statique avec la commande suivante:
  - $ arm-linux-gnueabihf-gcc -static -o hello hello.s

- Pour exécuter l'exécutable hello généré :
  - $ ./hello

Commandes optionnelles:

- Pour désassembler un exécutable généré en .s:
  $ arm-linux-gnueabihf-objdump -d hello

- Pour compiler du C en assembleur .s:
  $ arm-linux-gnueabihf-gcc -S hello.c

c. Capacités et limitations du compilateur

A la différence du compilateur decac en mode IMA, le mode ARM ne supporte pas le calcul sur les flottants. Les deux modes restent cependant semblables sur le reste des spécifications. Voici une liste de ce que le compilateur ARM supporte:

- Déclaration de variables entières et booléennes.
- Affectation de valeurs à des variables déclarées.
- Calcul et comparaisons sur les entiers.
- Opérations sur les booléens.
- Lecture d'un entier entré par l'utilisateur (avec readInt()).
- Affichage d'entiers et de chaînes de caractères (print() / println()).
- Structures conditionnelles (if then else / while).
- Déclaration d'un flottant initialisé par un immédiat flottant et son affichage en décimal (pas grand intérêt).

Le compilateur ne supporte pas (en plus de ce qui est spécifié en mode IMA) :

- Les opérations sur les flottants immédiats et variables.
- La déclaration de classes et de méthodes.
- L'allocation dans le tas avec new.
- La déclarations de variables entières plus grandes que 2**31 -1 ou plus petites que -(2**31 -1).
- Les opérations sur les entiers faisant intervenir des immédiats plus grands que 2**16 ou plus petits que -2**16.

d. Erreurs rencontrées

Les erreurs fréquemment rencontrées sont de plusieurs types, levées:

1. à la compilation vers assembleur ARM.
2. à la compilation vers exécutable ARM.

3. à l'exécution.

La 1ère catégorie d'erreurs provient généralement d'un programme Deca fournit ne respectant pas la spécification du compilateur ARM ou bien d'une erreur de ligne de commandes.
Cas notables:

- Les erreurs relevées par le compilateur en mode IMA.
- Intervention des flottants dans le calcul d'une expression ou comme immédiats.
- Présence d'Instructions/Déclarations relatives à l'Objet (new, this class..).
- Déclarations d'entiers signés ne pouvant pas tenir sur 32 bits.

Les erreurs de la catégorie 2 sont plus rares et concernent généralement une erreur sur le nom de fichier ou options de la commande; parfois même une mauvaise installation de la toolchain.

Pour la catégorie 3, les erreurs sont celles intervenant au sein des instructions dans l'exécutable. Ils peuvent intervenir dans le cas de:

- Pile pleine.
- Intervention d'immédiats entiers ne tenant pas sur 16 bits (spécification des registres ARM). Le message d'erreur suivant est affiché: Error: invalid constant (xxxx) after mixup.
- Intervention d'immédiats en écriture flottante.

D'autres erreurs concernant les définitions des labels, la syntaxe des instructions ARM.. ne doivent normalement pas intervenir à l'exécution.

e. Remarques

- printx et print sont équivalentes pour les variables flottantes.
- La lecture d'une entrée de l'utilisateur peut donner l'impression qu'elle affiche en même temps cette entrée, alors que ce n'est que sa trace. En effet, elle ne fait pas partie de l'output de l'exécution mais est juste superposée au reste de l'affichage du terminal.
- Le compilateur ARM ne gérant pas les overflows des opérations, il faudra faire attention aux opérandes donnés. Plus précisément, si le résultat d'une opération ne tient pas sur 32 bits, alors il reçoit la valeur

2**31 -1. Et si une opération sur deux opérandes de même signes donne un résultat de signe contraire, le résultat est quand même retourné (C'est le cas quand le résultat de l'opération est supérieur à 2**31 -1).

f. Comprendre un fichier .s

Forme générale d'un fichier .s généré par decac -arm:

Un programme assembleur ARM en .s se compose principalement de deux parties: la section "text" qui contient les instructions et "data" qui contient les variables déclarées. L'entrée du programme est dénotée par le label "main:". La sortie se fait par l'appel système SVC #0.
Des variables passe-partout sont déclarées au début de chaque .data et seront utilisées ou pas dans le code généré.

- n: contient le caractère de saut de ligne.
- printfloatxxxxxxxx: chaîne d'affichage des flottants.
- printintxxxxxxxx: chaîne d'affichage des entiers.
- scannerxxxxxxxx: chaîne de lecture d'entiers.

Quand une variable est déclarée, un label contenant son nom concaténé à un nombre est créé pour définir le type et le contenu de l'espace mémoire occupé par la variable.
Le programme est parsemé de commentaires marqués par un @ en début de ligne.

i. Directives rencontrées

.arch : spécifie l'architecture de sur laquelle sera exécuté le code compilé (ici armv8-a).
.extern : déclare des fonctions qui seront appelées des bibliothèques standards GNU (on appellera printf et scanf systématiquement à la génération de code même s'ils ne seront pas toujours utilisées).
.section : déclare une section.
.**text** : déclare la section contenant les instructions.
.global : annonce le label pour l'accès global depuis la bibliothèque standard GNU.

.data : déclare la section contenant les variables.
.align : indique à l'assembleur qu'il doit ajouter une quantité de padding pour aligner l'espace mémoire suivant sur une adresse multiple de 32 bits.
.word : déclare un mot de 32 bits en mémoire.
.double : déclare un mot de 64 bits en mémoire.
.asciz : déclare une chaîne de caractères ascii en mémoire se terminant par le caractère de fin "\0".
.- "label" : indique le taille de la zone mémoire vers laquelle pointe "label".

ii. Listes d'instructions ARM utilisées

Les Ri dénotent des registres, [Ri, #imm] leurs offsets, #imm des immédiats, =label l'adresse marqué par un label.
Les opérations unaires sont de la forme: **INST OP**
Les opérations binaires sont de la forme: **INST OP1 OP2**
Les opérations tertiaires sont de la forme: **INST OP1 OP2 OP3**

MOV Ri OP2 : Transfère le contenu de OP2 dans Ri avec OP2 soit Rj ou #imm.

MVN Ri OP2 : Similairement à MOV transfère le contenu de OP2 dans Ri mais en performant une négation bit à bit sur OP2.

LDR Ri OP2 : Charge le contenu de l'espace mémoire référencé par OP2 dans Ri. OP2 peut être [Rj, #imm] ou =label.

STR Ri OP2 : Opération inverse de LDR, stocke le contenu de Ri à l'adresse d'OP2.

SVC #IMM : Provoque une interruption système de type #IMM. Par exemple si IMM = 0, c'est l'interruption de sortie (exit).

ADD Ri Rj OP3 : Stocke le résultat de Op3 + Rj dans Ri. Positionne les bits V d'overflow signé et C de carry. Le carry signifie que le résultat ne tient pas dans le registre. OP3 peut être Rj ou #IMM.

SUB Ri Rj OP3 : Stocke le résultat de Rj - OP3 dans Ri. Positionne les bits d'overflow signé et de carry.

MUL Ri Rj Rk : Stocke le résultat de Rj * Rk dans Ri. Ne provoque pas d'overflow.

SDIV Ri Rj Rk : Stocke le résultat du quotient entier de Rj par Rk dans Ri. Ne provoque pas d'overflow.

CMP Ri OP2 : Compare OP2 au contenu de Ri (effectue Ri-OP2) et positionne les flags N (négatif), Z (zéro) V et C. OP2 est soit Rj ou #IMM.

B OP : branchement inconditionnel vers l'adresse à laquelle pointe OP (label ou autre).

Bcc OP : branchement si cc est vrai vers l'adresse à laquelle pointe OP (label ou autre). CC appartient à {EQ, NE, GT, LT, GE, LE, VS, VC, CC, CS…}.

BL OP : branchement inconditionnel vers l'adresse à laquelle pointe OP (label ou autre) avec sauvegarde de l'adresse de retour. Utile pour l'appel de fonctions, on l'utilisera pour afficher et lire avec printf et scanf.

4. Génération de code

On a essayé d'intégrer le cycle de génération de code ARM à celui de IMA existant.
Pour donner à l'utilisateur l'option de choisir vers quelle machine compiler, on a ajouté l'option "-arm" à ajouter à la compilation avec "decac". Cette option provoque le lancement de la méthode de CodeGenARMProgram (au lieu de CodeGenProgram pour IMA) qui va générer la totalité du code. On a donc dû dupliquer la plupart des méthodes de Codegen en CodeGenARM et créer des classes générant les instructions assembleurs ARM qu'on a ajouté au pseudocode comme LDR, MOV et SVC au début. Le hiérarchies des classes, la logique d'évaluation des instructions/expressions et d'attribution de registres sont les mêmes qu'en IMA, mais plusieurs adaptations se sont imposées:

- Les caractères de commentaire sont différents pour les deux assembleurs ("@" pour ARM). On a dû donc créer une méthode de génération de commentaire ARM qui affiche "@" au lieu de ";".
- Les arguments des instructions assembleurs ont leurs ordres inversés dans ARM (LDR RO, R1 fait le travail de LOAD R1, R0), on a fait les

adaptations nécessaires au niveau des classes d'instructions assembleurs.
- Création de la classe Directive comme attribut de Line et qui complète l'ensemble de lignes possibles à écrire en assembleur (imitant les directives ARM).
- On peut appeler des fonctions de C pour l'affichage et la lecture (printf, scanf).
- La déclaration des variables et leur stockage en mémoire se fait dans une section différente du code principal (.data). On attend donc la fin de la génération du code des instructions pour ajouter celle des variables en fin de programme ARM à l'aide de la méthode addARMData.

a. Affichage d'une chaîne de caractères

Le premier test pour afficher la chaîne Hello World nécessite des instructions d'affichage, un point d'entrée du code et une instruction de sortie. Cette dernière, imitant l'instruction HALT dans IMA, se fait par utilisation du registre R7 et une interruption système. Le point d'entrée est dénoté par le label "_start". Pour afficher la chaîne déclarée (avec sa longueur) en section .data, il faut la charger dans R1 et appeler la 4ème interruption système. Cette méthode ne marche que pour l'affichage des chaînes ascii, on ne peut donc pas afficher les autres types de data. Comme alternative, on a choisi d'appeler "printf" définie dans la librairie C ce qui nous a imposé de changer le point d'entrée en "main" et de compiler avec gcc au lieu de l'assembleur/éditeur de lien ARM (as/ld).

A la différence d'IMA, il n'y a pas de commande unaire qui affiche directement la chaîne passé en paramètre (WSTR) dans ARM. L'appel à printf exige le stockage de l'adresse de la chaîne dans R0. Par conséquent, la logique adoptée pour afficher les chaînes de caractères est comme suit. Chaque chaîne à afficher est stockée dans une structure de données Java (LinkedList) pour être ensuite retrouvée à la fin de la génération des instructions et au début de la section .data. Le label référant à la chaîne est créé en enlevant tous les espaces et les caractères spéciaux de la chaîne. Ainsi afficher "He%%o Wor%d" créera le label "HeoWord" qui contiendra la directive .asciz "He%%o Wor%d" len .-HeoWord.
Le calcul du label à partir du string donné se fait en 2 fois, une fois avant le stockage pour générer l'instruction LDR avec le label comme argument et une autre fois pour créer le label dans le .data. Même si ce n'est pas optimal, cela

nous a semblé préférable à stocker le string et le label dans deux structures différentes ou dans un hashmap.
Cependant, la création d'une arraylist contenant les labels déjà générés est nécessaire pour éviter les labels dupliqués (erreur ARM) quand on essaie d'afficher plusieurs fois la même chaîne par exemple.

Cas particuliers à noter:

- Le label "n" est toujours généré et contient le caractère de saut de ligne "\n".
- Si le label calculé est vide ou bien égal à "n", un label "filler" est stocké avec le string puis est traité pour générer un label valide unique.

Les classes d'instructions héritent de la classe Instruction. Pour les instructions binaires ARM, l'ordre des opérandes a été inversé dans les constructeurs car le sens de l'instruction est différent d'IMA.
Classes instructions déclarées:

- SVC : hérite d' UnaryInstructionImmInt, référence l'instruction ARM: SVC.
- LDR : hérite de BinaryDvalToReg, référence LDR.
- MOV : hérite de BinaryDvalToReg, référence MOV (DvalToReg même si MOV ne supporte pas le déplacement d'addresses).
- BL : hérite d' UnaryInstruction, référence BL.

Classes pseudocode déclarées:

- Directive: hérite directement d' Operand, ajoutée comme Attribut de Line permet la création de lignes qui ne sont ni des labels ni des instructions. Elle ajoute la chaîne passée en paramètre au programme.
  Grâce à elle, on peut maintenant représenter les directives ARM représentant les sections, ou les chaînes de caractères par exemple .data, .extern, .asciz, .align …
- Varcall: hérite de Daddr car il permet de référencer un label (emplacement mémoire) pour ensuite y charger des données ou les charger dans un registre. Affiche =label dans le programme.

Attributs ajoutés:

- armComment: dans AbstractLine pour pouvoir générer des commentaires ARM en "@". Ajout d'un booléen qui indique si le mode de commentaire doit être celui d'IMA ou d'ARM.
- stringData: dans DecacCompiler contient les chaînes de caractères à afficher.
- labels: dans Main contient les labels déjà générés.

```
.section .text
.global _start
_start:
        MOV R0, #1
        LDR R1, =Hello
        LDR R2, =len
        MOV R7, #4
        SVC #0
        MOV R7, #1
        SVC #0
.data
Hello:
.ascii "Hello\n"
len = .-Hello
```

```
.section .text
.extern printf
.global main
main:
    LDR R0, =Hello
    BL printf
    @saut de ligne
    LDR R0, =n
    BL printf
    @sortie
    MOV R7, #1
    SVC #0
.data
.align
Hello:
.asciz "Hello"
len = .-Hello
.align
n:
.asciz "\n"
len = .-n
```

Différence entre les deux stratégies d'affichages pour un println("hello")

b. Déclaration de variables et affectation de valeurs

Pour la déclaration de variables, on se trouve dans la même situation que dans la première partie: les valeurs déclarées en début de code doivent générer des labels en section .data.
De manière similaire à IMA, la déclaration de variable peut contenir une initialisation ou pas. Dans les deux cas, on associe à la variable un objet Varcall qui désignera son emplacement mémoire. Si initialisation il y a, la valeur issue de l'évaluation de l'expression d'initialisation va être stockée dans l'emplacement assigné à la variable via un LDR puis STR. La génération des labels est simple et se fait par un parcours de la liste des déclarations de variables contenue dans le Main.
Pour différencier entre les chaînes de caractères à afficher et les labels de variables, ces derniers sont concaténés à MAX_INT. Par exemple, si on déclare la variable hello, et on essaie d'afficher "hello", on trouvera les labels "hello" (chaîne) et "hello2147483647" dans la section .data.

A l'affectation d'une valeur à une variable, on ne fait que changer la valeur stockée à l'adresse du VarCall. Par contre, maintenant, on a besoin d'une nouvelle structure pour stocker les valeurs à affecter. On utilise cette structure (arraylist) pour assigner des valeurs aux labels qui n'ont pas été initialisées.
Comme la structure d'attribution de registres est la même que celle utilisée dans l'implémentation d'IMA, la valeur stockée puis déclarée dans le VarCall désignera le numéro du dernier registre utilisé. Ce n'est pas grave, car ce

n’est que de l’affichage mais la valeur réelle de la variable est bien dans l’adresse spécifiée par VarCall.

Classes instructions déclarées:
- STR : hérite de BinaryDvalToReg, référence STR.

Classes pseudocode déclarées:
- ARMRegisterOffset : hérite de RegisterOffset et redéfinit son affichage pour qu’il soit compatible avec ARM.

Attributs ajoutés:
- assigns : dans DecacCompiler contient les valeurs d’affectations.

c. Représentation/Affichage des entiers signés

Il n’est pas difficile de représenter les entiers sur ARM. Les variables entières sont déclarées dans le .data comme .word c’est à dire mot de 32 bits. Les immédiats par contre doivent tenir sur 16 bits car c’est la taille que supportent les registres Ri.
Pour afficher des entiers, il faut charger les caractères de formatage “%i” dans le registre R0, puis charger l’entier à afficher dans le registre R1 pour enfin appeler printf. En pratique, un label “printint2147483647” est réservé à la chaîne “%i” pour afficher tous les entiers.

d. Opérations arithmétiques

La structure d’évaluations des opérations binaires et unaires a déjà été établie pour IMA. Pour ARM, on ne fait que dupliquer les méthodes de génération de code avec une différence de traitement entre les Identifiers et le reste des expressions. Les adresses des variables doivent d’abord être chargées dans un Ri, puis la valeur de la variable est extraite de l’offset du de Ri pour le calcul ou l’affichage.
Comme expliqué dans la section précédente, l’utilisation d’immédiats plus grands que 2**16 dans les opérations va lever une erreur au niveau des registres.
Pour des raisons qu’on va citer plus tard, les opérations sur les flottants ne sont pas supportées par notre compilateur. Par conséquent, les opérations arithmétiques implémentées sont l’addition, soustraction, multiplication, quotient et reste de deux entiers signés.

Les instructions arithmétiques ARM requièrent 3 registres, donc une nouvelle classe pseudocode a été créée pour les englober.

Classes instructions déclarées héritent tous de TertiaryARMInstructions (sauf MVN) et possèdent le suffixe ARM pour les différencier des instructions IMA :

- ADDARM : référence ADD.
- SUBARM : référence SUB.
- MULARM : référence MUL.
- SDIVARM : référence SDIV.
- MVN : hérite de BinaryDvalToReg, référence MVN.

Classes pseudocode déclarées:

- TertiaryARMInstructions : hérite d' Instruction, la méthode getName est redéfinie pour enlever le suffixe ARM à l'affichage.

L'instruction SDIV ne figure pas sur la version ARM qui est exécutée par défaut. En analysant le résultat de compilation d'une division en C vers ARM, on a remarqué la directive .arch armv8-a ajouté en tête de fichier. Depuis, on travaille en armv8 pour supporter la division entière.
L' opération modulo n'a pas d'instruction associée, mais elle est implémentée en combinant la division entière, la multiplication et la soustraction sur plusieurs registres. L'équation associée est : a%b = a - E(a/b)*b
De même, le moins unaire est équivalent à calculer le complément à 2 de l'opérande, c'est-à-dire, l'inversion bit à bit et lui ajouter 1. C'est ici que MVN intervient.

Les opérations d'addition et de soustraction peuvent générer deux types d'overflow:
Le carry (flag C) signifie que le résultat de l'opération ne tient pas sur 32 bits et donc la valeur 2**31 - 1 est stockée à sa place. Le signed overflow (flag V) est positionné à vrai quand une opération entre deux opérandes de même signe produit un résultat de signe contraire.
La multiplication et l'addition ne génèrent jamais d'overflow dans le mode ARM.
La gestion de l'overflow n'a pas été implémentée dans notre compilateur, l'utilisateur a donc intérêt de ne pas utiliser des valeurs trop grandes (pouvant dépasser 32 bits) durant les calculs pour ne pas avoir de résultats bizarres.

e. Opérations arithmétiques

Puisque on utilise les mêmes BooleanLiteral qu'en IMA, on a choisi de préserver la même logique de traitement booléens comme branchement et on a adapté les méthodes de génération de code. Un booléen évalué produit un branchement s'il est "faux".
ARM possède des instructions AND et ORR qui correspondent aux "et" et "ou" bit à bit et qu'on aurait pu utiliser pour traduire les "et" "ou" logiques vu que "vrai" correspond à #1 et "faux" à #0.
C'est la même histoire pour les comparaisons: les opérandes sont chargés, on les compare puis on branche si le résultat est "faux".
Ici aussi, on distingue entre le cas où on doit stocker le résultat de la comparaison dans une variable booléenne et quand on doit simplement brancher. Dans le premier cas, une comparaison avec #0 est nécessaire pour déterminer le résultat de la comparaison.
On utilise l'instruction IMA CMP qu'on a adapté pour inverser l'ordre des opérandes.
Les instructions de branchement restent les mêmes car les codes conditions de comparaison sont les mêmes en ARM sauf pour le branchement inconditionnel qu'on a dû définir dans la classe d'instruction B.
Pour les structures conditionnelles, aucune adaptation n'a été nécessaire.

f. Entrées de l'utilisateur

La fonction readInt est supportée en mode ARM pour la lecture d'entiers. Elle requiert l'appel de scanf qui se charge effectivement de lire l'entier et de le stocker dans l'adresse voulue. Concrètement, il faut au préalable avoir défini un espace mémoire consacré à l'entier lu pour le charger dans R1. Si le résultat de la lecture va être stocké dans une variable, le label est déjà créé au nom de la variable. Si le résultat de lecture va intervenir dans une opération ou être affiché directement, un label va être généré sous le nom de "read" + un nombre aléatoire. Dans ce cas, ces labels sont stockés dans une arraylist qui interviendra dans la génération du .data.

Attributs ajoutés:

- readVars : dans DecacCompiler contient les Varcall des lectures sans variables.

g. Le cas des flottants

Les flottants peuvent être stockés en mémoire selon plusieurs degrés de précision. En IMA, les flottants sont stockés comme mots de 32 bits selon la norme IEEE-754. Sur ARM, selon les versions, les flottants peuvent être interprétés comme mots de 16, 32 ou 64 bits et des modules de calculs sur les flottants peuvent ou pas être inclus dans le compilateur. Malgré nos recherches et essais, la représentation en float ou half n'a pas été possible. Ces formats ne sont pas reconnus par la fonction d'affichage printf, on a donc décidé de conserver la représentation de double précision (64 bits). Cette représentation pose le problème suivant: Comment manipuler des données codées sur 64 bits par un compilateur 32 bits?
Deux idées de réponses viennent à l'esprit:

- On peut utiliser un des modules complémentaires sur les flottants. Par exemple, VFP (Vector floating point) présente des pseudo instructions pour traiter les flottants de double précision (VMOV, VLDR, VADD…). On peut spécifier le module de traitement des flottants utilisé en ligne de commande avant la compilation (en ajoutant -mfpu=vfp3 -mfloatabi=hard par exemple) ou dans le fichier .s à l'exécution en ajoutant la directive .mfpu.
  On a essayé plusieurs combinaisons de versions d'ARM et de modules de flottants mais aucune n'a produit le résultat attendu et tous résultent en erreurs d'incompatibilité en contradiction avec ce qu'indique les documentations d'ARM. Quand VMOV a été reconnue par notre compilateur, elle n'a accepté que des immédiats entiers par exemple.
- Comme alternative, on peut choisir de considérer les doubles de 64 bits en 2 mots de 32 bits (moitié haute et basse) et les stocker chacun dans un registre. Cela nécessiterait de redéfinir la structure d'attribution de registres et les opérations arithmétiques dans le code ou bien de créer des méthodes complémentaires qui traitent les flottants séparément pour s'adapter avec le besoin de sauvegarder 2 registres par flottant. De plus, il faudra redéfinir chaque opération arithmétique comme le résultat de deux autres et essayer de reconstituer le double voulu. On a finalement pas eu le temps à consacrer pour implémenter les calculs sur les flottants même si notre compilateur supporte la déclaration de variables flottantes et leurs affichages.

En absence d'opérations binaires sur les flottants et d'immédiats flottants, la déclaration de variables flottantes présente peu d'intérêt. Quand même, on l'a implémenté comme fonctionnalité additionnelle de notre compilateur.
La déclaration des flottants se fait de la même façon que pour les entiers, la génération se fait dans la section .data, à la différence près que les flottants sont déclarés avec la directive ".double" au lieu de ".word". Les FloatLiteral génèrent toujours un immédiat flottant dont la valeur est affiché en hexadécimal et donc n'est pas reconnu par le compilateur ARM. On a dû créer la un nouveau type d'immédiat double qui conserve l'écriture décimal qui sera affiché à la suite du .double.

Classes pseudocode déclarées:

- ImmediateDouble : hérite de DVal, utilisée pour générer les déclarations des flottants en ARM.

Le label "printfloat2147483647" déclare le caractère de formatage "%f" pour afficher les flottants. Lui aussi est généré systématiquement à chaque compilation. Pour afficher un flottant, il faut charger son adresse dans R1, sa partie haute dans R2 et sa partie basse dans R3 pour enfin appeler printf. L'affichage est toujours en décimal même si on utilise printx.

5. Validation du compilateur

Pour valider les capacités du compilateur du compilateur ARM, on a construit une base de tests qui contient les tests IMA compatibles avec ARM et d'autres complémentaires qui testent les limites d'ARM.
En pratique, chaque fonctionnalité implémentée pour ARM est testée sur plusieurs cas différents pour s'assurer de sa conformité aux spécifications. Les tests sont ensuite ajoutés au répertoire /src/test/ARM qui contient les tests valides, invalides, et interactifs.
Cette base de test n'a pas été incorporée dans le processus automatisé de validation d'IMA. Les tests sont donc toujours lancés un par un "à la main".
La base de tests ARM étant très petite par rapport à celle d'IMA, l'impact énergétique dû au testing est vraiment minime.
Concernant l'efficacité de cette méthode de validation, elle est loin d'être parfaite. L'opération modulo par exemple ne marchait pas dans quelques cas qui n'ont été remarqués qu'après le rendu final. De même pour un problème concernant l'affichage de quelques chaînes de caractères spéciaux. Aussi, le

readInt présente des résultats faux quand appelé dans des expressions complexes d'opérations arithmétiques.
Sinon, pour le reste, toutes les spécifications de l'extension ont été validées.

6. Qualité du code

Le code produit pour implémenter l'extension ARM dans la structure de compilation fournie est pour la plupart solide et clair. A l'exception de quelques adaptations majeures mentionnées dans les parties précédentes, la plupart du code est de la duplication des méthodes de génération de code déjà écrites pour IMA.
Le coding style est respecté: les noms de variables sont pertinents au possible et les noms de méthodes et classes ajoutées contiennent ARM pour les distinguer du travail fait pour IMA.
Néanmoins, le code présente quelques lacunes au niveau algorithmique dont on est conscient. Pour résoudre le problème de génération de code de la section .data, on a choisi de mobiliser quelques structures Java pour stocker les noms des labels et les valeurs des variables. Au final, 3 Lists ont été créés pour pallier la séparation entre l'instruction d'affichage et la chaîne qui va être affichée. Bien que les structures soient bien choisies selon leur utilisation et parcourues par des for each; il est clair qu'une solution utilisant moins de structures aurait pu être possible.
Comme alternative, le code généré pour le .data aurait pu être positionné avant la section d'instructions pour .text par une manière similaire à l'ajout des TSTO en IMA (encore une fois, il y aura pas mal de code à ajouter et de parcours de listes à faire).
Les parties de traitements de chaînes de caractères pour les noms de labels sont un peu complexes et leur compréhension n'est pas facile surtout avec le manque de commentaires. Aussi, instanceof a été beaucoup utilisée pour identifier les types de quelques expressions principalement car le traitement des variables est différent de celui des autres expressions.
La plupart de ces choix résultent du fait qu'on a pas trop voulu perturber la structure établie et refaire différemment un travail déjà effectué.

7. Analyse bibliographique

La documentation sur les architectures ARM n'est pas rare sur Internet, c'est l'une des architectures les plus répandues. Cependant, la difficulté de se renseigner vient de la grande diversité des versions ARM, des modules qu'elles supportent et même de la syntaxe de leurs assembleurs. Il faut donc trouver des sources qui s'intéressent à la même version d'ARM et toujours tester si ce qui y est mentionné est bien applicable à la version installée.

La première étape est l'installation de la toolchain de compilation. Pour cela, le guide suivant a été suffisant. En plus du guide d'installation, le site était une source essentielle pour la compréhension d'ARM car il contient des cours sur l'assembleur ARM.

On a aussi utilisé des documents et sites qui comportent les listes d'instructions et directives ARM ainsi que la structure des registres et de la pile etc..

- https://documentation-service.arm.com/static/5f8dacc8f86e16515cdb865a?token=
- https://sourceware.org/binutils/docs/as/
- https://profile.iiita.ac.in/bibhas.ghoshal/COA_2021/lecture_slides/arm_inst.pdf
- https://www.ic.unicamp.br/~celio/mc404-2014/docs/gnu-arm-directives.pdf

Les sites https://developer.arm.com/ et https://www.keil.com/ contiennent de la documentation compréhensive pour toutes les architectures ARM. Encore faut-il savoir trouver ce dont on a besoin: parmi les centaines de pages de doc, la plupart référencent des instructions et modules qu'on n'utilise pas.

StackOverflow nous a bien sûr été utile bien que 90% des réponses concernent des versions différentes d'ARM et n'étaient donc pas applicables dans notre cas. L'utilisation du debugger et la compilation d'un code C en assembleur ARM restait pour nous la meilleur façon d'analyser comment notre compilateur ARM fonctionnait. Par exemple, c'est comme ça qu'on a remarqué que l'instruction SDIV n'existait pas avant la version armv8.

8. Pistes d’améliorations

- La gestion de l’overflow est un aspect important de l’arithmétique des entiers en assembleur. L’utilisateur attend que les opérations produisent des résultats cohérents pourvu que les valeurs en jeu tiennent sur 32 bits. Avec plus de temps, c’est cet aspect de l'extension qui aurait été prioritaire en implémentation.
- La recherche sur la gestion des flottants nous a pris beaucoup de temps comparé à ce qu’on a finalement pu implémenté. S’approprier un module complémentaire de gestion de flottants ou bien établir une politique de calcul sur les double en est la première étape.
- Optimiser le nombre d’instructions générées. Par exemple utiliser AND/ORR pour les opérations sur les booléens.
- Intégrer l’utilisation systématique de la pile dans les calculs pour s’adapter au manque de registres.
- Établir une politique de validation bien plus fiable et efficace que l’actuelle en multipliant les tests et les automatisant.