Compilateur Decac

Documentation d'extension [ARM]

pour architecture ARMv6-fp

TEIMUR ABU ZAKI, ADRIEN BOUCHET, TROY FAU, PAUL MARTHELOT, HUGO ROBERT

Introduction

Ce document a pour objectif d'expliquer la conception de l'extension permettant la génération de code assembleur pour les architectures ARMv6-fp.

Choix de la version de ARM

La dernière version de ARM est actuellement la version 8 pour les processeurs 64 bits. La version implémentée par Decac n'est pas la plus récente dans l'objectif d'être compatible sur un maximum d'architectures tout en possédant les fonctionnalités nécessaires. La version de ARM utilisée repose donc sur un set d'instruction 32 bits et permet l'exécution de calcul sur flottants. Seules les versions 6 et 7 de ARM permettent les calculs flottants tout en maintenant un set d'instruction sur 32 bits [9], Decac implémente donc la version 6. A partir de cette version, le calcul avec flottant est correctement géré grâce à un set d'instruction optionnel. Nous utilisons l'extension VFP (Vector Floating Point) en tant qu'unité de calcul sur des flottants scalaires. Decac appelle la version VFPv2 [7] qui permet notamment la conversion entre float et double, nécessaire pour pouvoir appeler la fonction printf de la bibliothèque standard C.

Options de compilation

Pour la compilation[5] d'un fichier assembleur ARM généré par Decac, il est nécessaire d'effectuer deux opérations. Une phase de compilation :

arm-linux-gnueabihf-as -march="armv6+fp" -mfloat-abi=soft -g filename.s -o filename.o -Wall

-march="armv6" Permet de préciser la cible ARM. Cela permet au compilateur d'identifier les instructions possibles.

+fp Permet d'utiliser l'extension VFPv2 pour la gestion des calculs flottants.

-mfloat-abi=soft Permet de spécifier les conventions d'appel aux librairies de gestion de calculs sur flottants. 'soft' permet de générer du code qui fait ensuite appel aux opérations en flottants. L'option 'hard' aurait permis une gestion plus détaillée des calculs en générant directions des instructions hardware pour manipuler les flottants.

-g Permet l'affichage des informations de debug.

-Wall Permet de passer des arguments au compilateur.

Une phase d'éditions de liens vers les fonctions assembleurs.

arm-linux-gnueabihf-gcc -static -g filename.o -o filename -lc

-static Permet de ne pas effectuer d'édition de liens avec des librairies qui seraient partagées.[2]

-g Permet toujours l'affichage des informations de debug.

-lc Permet d'éditer des liens avec les fonctions de la bibliothèque standard de foncitons C.

Spécification de l'extension

Lexicographie

La lexicographie est identique à celle du langage Deca restreint à la partie sans objets à quelques exceptions près. Les nombres ne peuvent pas être affichés en hexadécimal. L'inclusion de fichier n'est pas gérée.

Syntaxes abstraite, contextuelle et de décompilation

La syntaxe acceptée est égale à celle du sous-langage Deca noté "Sans-Objet" et défini dans la spécification du langage Deca.

Semantique

La sémantique est la même que pour le langage Deca Sans-Objet. Pour le calcul flottant, on s'appuiera sur la nomre IEEE-754 et ils seront représentés en simple précision sur 32 bits. Cependant, nous utiliserons ici la fonction printf de la bibliothèque standard de C. Cette fonction prend en argument des flottants de type double précision sur 64 bits, c'est pourquoi notre code assembleur généré effectue une conversion avant l'affichage seulement. L'affichage en hexadécimal avec les fonction printx et printlnx n'est pas géré. Le reste de la sémantique est identique au langage Deca "Sans-Objet".

Choix de conception

Pour tirer profit de l'architecture déjà en place pour la génération de code assembleur pour la machine abstraite IMA, les classes ARM ont été placées dans le même répertoire que celles de IMA. Cependant elles ne font pas partie du même package. Classes ajoutées dans *pseudocode* :
Nouveaux constructeurs d'instructions :
ARMBinaryInstructionDValToReg
ARMBinaryInstructionStringToReg

ARMUnaryInstructionInt
ARMUnaryInstructionString
ARMTernaryInstruction

Nouvelles classes de gestion des registres :
ARMGPRegister
ARMLine
ARMRegister

Classe pour la gestion de la pile pour permettre les push de plusieurs registres :
ARMStackInstruction

Classe fille de GenericProgram : ARMProgram

Classe de gestion des erreurs et ajout de fonctions non disponibles dans le jeu d'instructions :
ARMErrorManager
ARMFunctionManager

Dans les classes préexistantes, les fonctions principalement modifiées sont les fonctions `codeGenInst` et `codeGenPrint` qui ont maintenant une version spécifique pour la génération d'assembleur ARM.

Définition de ARM et de son langage d'assemblage

0.1 Données et mémoires

Les explications suivantes suivent le modèle de description de la Machine Abstraite fourni dans le polycopié des spécifications du projet. Les types des valeurs manipulées sont les entiers, les flottants, les adresses (séparées en « adresses code » et « adresses mémoire »). La machine abstraite gère les nombres sur 32 bits, et utilise la représentation de la norme IEEE-754 pour les flottants.
La "mémoire physique" de la machine est logiquement partagée en 4 zones :

- **La zone registres** Elle est constituée des registres banalisés r0 à r15. Ils peuvent contenir des valeurs de tout type, et peuvent être lus ou modifiés. Les registres spécifiques de r11 à r15 sont réservés aux adresses spécifiques.
  r0 - r6 -> Registres généraux utilisé dans les méthodes
  r7 -> Registre général avec une fonction spécifique permettant de gérer les appels systèmes
  r8 - r10 -> Registres généraux
  r11 -> fp (frame pointer : pointe sur le fond de la pile)
  r12 -> ip (Intra procedural : un registre scratch qui en général ne stocke pas les paramètres de fonctions appelées)

r13 -> sp (Stack Pointer : pointe sur le haut de la pile, utilisé pour demander de l'espace sur la pile avec add sp, sp, #4 par exemple ou encore désallouer de l'espace sur la pile avec sub sp, sp, #4)

r14 -> lr (link register : registre contenant systématiquement l'adresse de l'instruction suivant l'instruction courante)

r15 -> pc (Registre contenant l'adresse de l'instruction courante + 8 ce qui est spéficique à ARM).
L'extension VFPv2 ajoute des registres en plus s0 à s31 pour stocker les flottants sur 32 bits et d0 à d31 pour stocker les flottants en double précision sur 64 bits.

CPSR -> Current Program Status Register (stocke les code conditions suite aux comparaisons)

- **La zone code (.text)** Elle contient les instructions du programme. À cette zone est associé un registre spécialisé, PC (compteur ordinal), qui contient les adresses successives des instructions à exécuter (appelées « adresses code »). PC peut être lu et modifié explicitement. PC+4 (resp. PC-4) est l'adresse de l'instruction suivant (resp. précédant) celle d'adresse PC.
- **La zone code (.data)** Elle contient l'ensemble des variables déclarées dans le programme. Ces variables y sont déclarés selon leur type (entier, flottant ou chaîne de caractères).
- **La pile** La zone mémoire est constituée de mots. À chaque mot est associée une « adresse mémoire ». Seuls certains mots sont consultables et modifiables ; on les appelle « mots adressables ». En zone pile, il s'agit des N mots constituant la pile (N n'est pas fixé a priori). Chaque mot adressable peut contenir une valeur de tout type et peut être lu ou modifié. La valeur d'un registre peut être envoyée dans la pile avec push r et récupérée avec pop r.
  Les éléments de mémorisation (registres banalisés et mots mémoires) sont « typés » dynamiquement. Initialement, tout est « indéfini » ; lors d'une modification d'un élément, le type de données est aussi mémorisé. Lors d'une opération, il y a vérification de compatibilité de type. Une valeur particulière, appelée null, est de type « adresse mémoire ». Elle représente une « absence d'adresse ». Avant l'exécution de la première instruction : — Le contenu de la pile ainsi que des registres R0 .. Rn est indéfini — PC est initialisé (par le chargeur) à l'adresse de la première instruction à exécuter dans la zone code.

0.2 Modes d'adressages

On dispose de 6 modes d'adressage suivants :

- **registre direct** Rm (m dans 0 .. 15)

- **registre indirect avec déplacement** [rx, #4], où x est entier $\in$ [0,15]. Le mode d'adressage désigne l'adresse mémoire (contenu de rx)+4.
- **immédiat #d** , où soit d est un littéral entier ou flottant en notation Deca éventuellement précédé d'un signe + ou - (auquel cas la valeur désignée est l'entier ou le flottant correspondant), soit d est la séquence de 4 caractères null (donc le mode d'adressage est #null), auquel cas la valeur désignée est l'adresse mémoire null.
- **Etiquette** etiq, ou etiq est une étiquette de programme en langage d'assemblage. etiq désigne l'adresse code de la première instruction qui suit l'étiquette.

0.3 Spécificités de l'assembleur ARM

L'architecture des registres ARM est particulière car elle est scindée en 3 types de registres. les registres r sont les registres standards et permettent de stocker des valeurs sur 32 bits interprétées comme des entiers signées. Les registres s peuvent être utilisés pour stocker des valeurs sur 32 bits qui peuvent être interprétées comme des flottants. Les registres d possèdent 64 bits et permettent de stocker des flottants double précision.

L'extension VFPv2 donne accès aux registres s et d non disponibles dans les versions standards de ARM. De plus, les flags utilisés lors des opérations de comparaisons sont dédoublés par l'extension. Les calculs sur flottants nécessitent beaucoup de conversions. Avant chaque opération entre flottants, il nous faut déplacer les valeurs des registres standards d vers les registres spécifiques aux calculs flottants s. Les instructions des opérations sont aussi différentes selon que l'opération se fait entre flottants des registres s ou entiers signés des registres d.

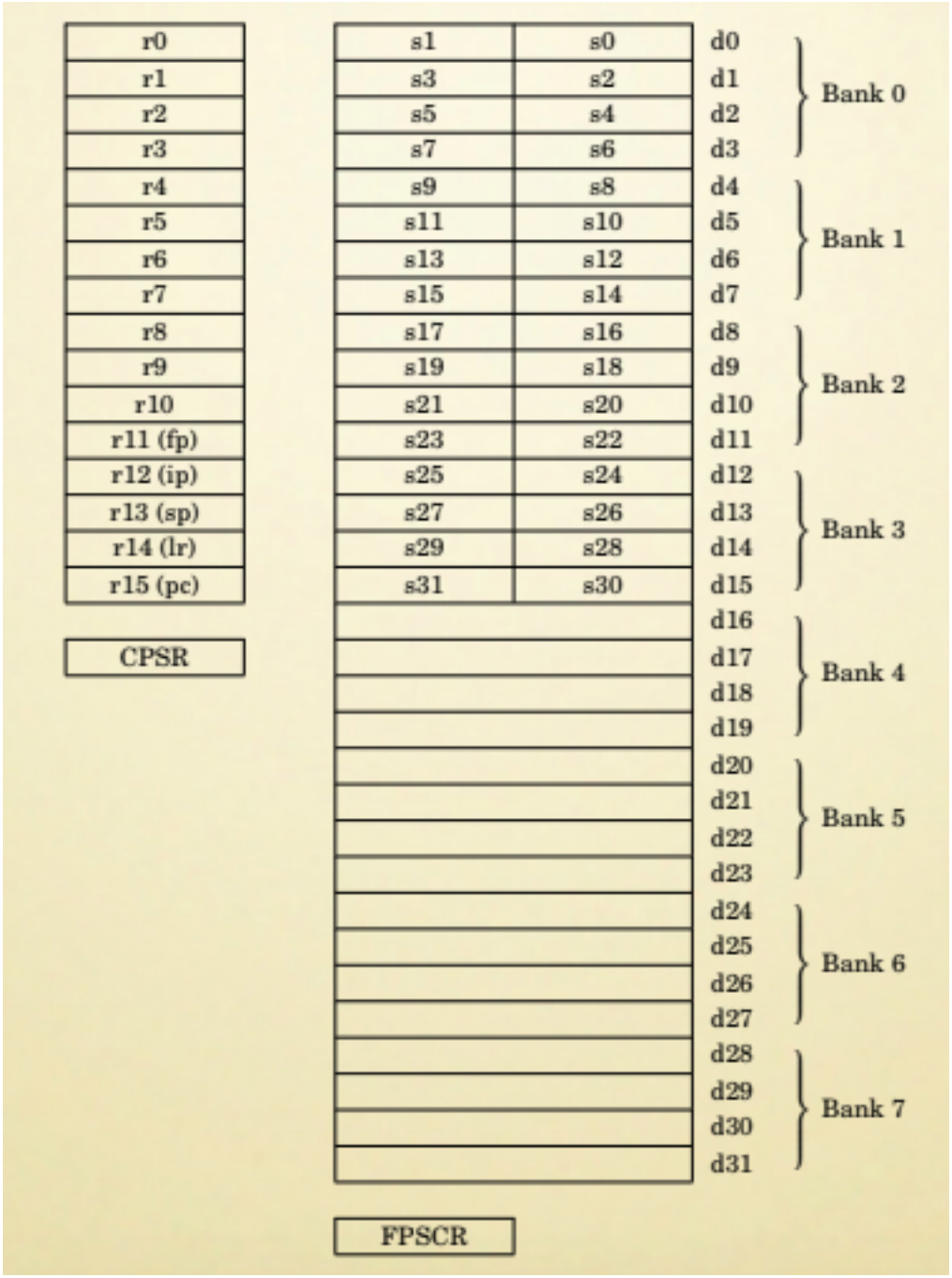

FIGURE 1 – Registres disponibles après extension VFPv2 [1]

Comparaison entre entiers / boolean

Pour les entiers et les booleans (true = 1, false = 0) on utilise l'instruction *cmp* qui met à jour les flags CPSR.
Les flags CPSR ont leur équivalents en version flottante. Il s'agit des flags FPSCR.

Current Program Status Register équivalent aux codes condition

Flag	Description
N (negatif)	Activé si le résultat d'une instruction renvoie un nombre négatif
Z (zero)	Activé si le résultat d'une instruction renvoie 0
C (Carry)	Activé si le résultat d'une instruction nécessite un 33 bit pour être correctement représenté
V (Overflow)	Activé si le résultat d'une instruction renvoie une valeur non représentable en code complément à 2 sur 32 bits
E (Endianbit)	0 si la version de ARM est en mode little endian, 1 pour big endian mode
T (Thumbbit)	Activé si ARM est en mode Thumb, désactivé en mode ARM classique
M (Modebits)	Spécifie le niveau d'accès au matériel (USR, SVC, ...)
J (Jazelle)	Autorise l'exécution de bytecode Java si activé

Codes conditions			
Code	Définition (vcmp)	Définition (cmp)	Flags testés
eq	égal à	égal à	Z==1
ne	Non ordonné, ou non égal	Non égal	Z==0
cs ou hs	Plus grand que, égal à ou non ordonné	Plus grand que ou égal à (non signé)	C==1
cc ou lo	Moins que	moins que (non signé)	C==0
mi	Moins que	Négatif	N==1
pl	Plus grand que, égal à, ou non ordonné	Positif ou zéro	N==0
vs	Non ordonné (un des arguments est nul)	Overflow signé	V==1
vc	Non ordonné (Aucun des arguments n'est nul)	Pas d'overflow signé	V==0
hi	Plus grand que ou non ordonné	Plus grand que (non signé)	(C==1) && (Z==0)
ls	Moins que ou égal à	Moins que ou égal à (non signé)	(C==0) || (Z==1)
ge	Plus grand ou égal à	Plus grand ou égal à (signé)	(N==V)
lt	Moins que ou non ordonné	Moins que (signé)	(N !=V)
gt	Plus grand que	Plus grand que (signé)	(Z==0) && (N==V)
le	Moins que, égal à ou non ordonné	Moins que, ou égal (signé)	(Z==1) || (N !=V)
al (ou rien)	Toujours exécuté	Toujours exécuté	Non testé

source : [3]

Comparaison entre flottants

```
vmov s0, r0 // deplacement des valeurs vers les registres s des flottants
vmov s1, r1
vcmp.f32 s0, s1 // comparaison de 2 valeurs flottantes sur 32 bits
vmrs APSR_nzcv, fpscr // deplacement des flags de comparaison \\
des flottants vers celui des entiers.
```

[6]

Set d'instructions ARMv6 utilisé par decac

Instructions de branchement			
Instruction	Définition	Equivalent IMA*	Exemple deca/ARM
B	Branchement à une adresse cible	BRA	BRA etiquette/ b etiquette
BL	Appel d'une fonction	BSR	BSR etiquette/ BL etiquette
BLX	Appel d'une fonction via un registre	BSR	BSR R0/ blx r0
BX	Branchement à une adresse cible	BRA	BRA R0/ bx r0

*les instructions ARM n'ont pas toujours d'équivalent parmi les instructions IMA.

[6]

Instructions d'opérations arithmétiques sur entiers			
Instruction	Définition	Equivalent IMA*	Exemple deca/ARM
ADD	Addition entre registres	ADD	ADD R1, R0/ add r0, r0, r1
SUB	Soustraction entre registres	SUB	SUB R1, R0/ sub r0, r0, r1
MUL	Multiplication entre registres	MUL	MUL R1, R0/ mul r0, r0, r1
NEG	négation du contenu d'un registre	OPP	OPP R1, R0/ neg r0, r1

[6]

Instructions de gestion mémoire			
Instruction	Définition	Equivalent IMA*	Exemple deca/ARM
MOV	Déplacement de contenu d'un registre	LOAD	LOAD R0, R1 / mov r1, r0
POP	Supprime la dernière valeur enregistrée dans la pile pour la mettre dans un registre	POP	POP RO/ pop rd ou pop r0, r1
PUSH	Ajoute la valeur du registre sur la pile	PUSH	PUSH R0 / push rs ou push r0, r1
STR	Chargement du contenu d'un registre à une adresse en mémoire	STORE	STORE R0, adr_memoire/ str r0, =adr_memoire
LDR	Chargement du contenu d'une adresse mémoire dans un registre	LOAD	LOAD R1, R1/ ldr r1, =x puis ldr r1, [r1]

[6]

Les booléens sont codés comme des entiers. 0 pour false et 1 pour true.
Le and se fait par addition des deux registres et comparaison à 2.
Le or fonctionne sur le même principe en comparant à 1.
Pour les instructions de de comparaison <, >, <=, >=, nous générons d'abord le code de comparaison entre registre `cmp` puis nous chargeons de manière conditionnelle le registre r0 avec la valeur 1 en utilisant le conditions après l'instruction mov.
Exemple : moveq, movgt, ...

Instructions booléennes non disponibles directement en ARM			
Instruction	Définition	Equivalent IMA*	Exemple deca/ARM
AND	and standard avec résultat dans r0 (r1 et r2 pour les operandes)	=>	ADD R1, R2 puis CMP R2, #2 puis SEQ R0 / mov r0, #0 puis add r2, r1, r2 puis cmp r2, #2 puis moveq r0, #1
R1 > R2	Comparaison	=>	CMP R2, R1 puis SLT R0 / mov r0, #0 puis cmp r2, r1 puis movlt r0, #1
R1 >= R2	Comparaison	=>	CMP R2, R1 puis SLE R0 / mov r0, #0 puis cmp r2, r1 puis movle r0, #1
R1 < R2	Comparaison	=>	CMP R2, R1 puis SGT R0 / mov r0, #0 puis cmp r2, r1 puis movgt r0, #1
R1 <= R2	Comparaison	=>	CMP R2, R1 puis SGE R0 / mov r0, #0 puis cmp r2, r1 puis movge r0, #1
NOT	Inversion de la valeur d'un registre	=>	CMP #0, R1 puis SEQ R0/ mov r0, #1 puis sub r0, r0, r1
OR	or standard avec résultat dans r0 (r1 et r2 pour les operandes)	=>	ADD R1, R2 puis CMP R2, #0 puis SNE R0 / mov r0, #0 puis add r2, r1, r2 puis cmp r2, #0 puis movgt r0, #1

[6]

Instructions de calculs flottants			
Instruction	Définition	Equivalent IMA*	Exemple deca/ARM
VADD.F32	Addition entre registres	ADD	ADD R0, R1/ vmov s1, r1 puis vmov s2, r2 puis vadd.f32 s0, s1, s2 puis vmov r0, s0
VSUB.F32	Soustraction entre registres	SUB	SUB R0, R1/ vmov s1, r1 puis vmov s2, r2 puis vsub.f32 s0, s1, s2 puis vmov r0, s0
VMUL.F32	Multiplication entre registres	MUL	MUL R0, R1/ vmov s1, r1 puis vmov s2, r2 puis vmul.f32 s0, s1, s2 puis vmov r0, s0
VDIV.F32	Division entre registres	DIV	DIV R0, R1/ vmov s1, r1 puis vmov s2, r2 puis vdiv.f32 s0, s1, s2 puis vmov r0, s0

[6]

Algorithme

Litéraux flottants

En tenant compte que l'instruction *vmov* qui concerne les flottants ne peut pas prendre un second paramètre un immediat (qu contraire de l'instruction originiale *vmov*) , on était obligé de stocker tous les litéraux flottants rencontrés dans le programme deca dans une section .data avec chacun un label unique. Par exemple si on considère une instruction rencontrée pendant le programme comme **x = 3.2**, on aura ce code ci-dessus :

```
.data
labelunique: .float 3.2
.text
ldr r0, =x
ldr r1, =labelunique
vldr s0, [r1]
vmov r1, s0
str r1, [r0]
```

Par contre si on aura une instruction **x = 3** on pourra faire juste :

```
.text
ldr r0, =x
mov r1, #3
str r1, [r0]
```

Division

Les instructions disponibles dans la version 6 de ARM ne couvrent pas toutes les opérations arithmétiques. Ne sont pas couverts par les instructions de base la division euclidienne ainsi que le modulo. La spécification du langage Deca implique l'exécution d'une division euclidienne et le renvoi du quotient lorsque l'utilisateur demande une division entre deux entiers (`entier1 / entier2`).
La spécification de Deca demande aussi l'implémentation du modulo (`entier1 % entier2`). L'instruction `div` existe bel et bien en assembleur ARM mais retourne le quotient sous forme de flottant.
Voici l'algorithme qui a été mis en place pour la gestion des divisions euclidiennes : Les valeurs absolues de la dividende doivent être chargées dans **r0** et le diviseur dans **r1**. Après branchement vers la fonction `divide`, le quotient se trouve dans **r2** et le reste dans **r0**, après la retour de la fonction on rétabli le signe attendu du resultat.

```
divide:
        push {lr}
        mov r2, #0
divide_iter:
        cmp r0, r1
        poplo {pc}
        sub r0, r0, r1
        add r2, r2, #1
        bl divide_iter
```

Affichages

Pour les affichages on a choisi d'utiliser la librairie C, en particulier **printf** qui peut afficher les entiers, flottants et string. On va détailler pour chaque cas ce qu'on fait :

strings Pour le string qu'on on va afficher, on crée une section *.data* qui va contenir une label unique qui pointe vers l'ASCII du string. Puis pour la convention d'appel de printf on doit passer l'adresse à **r0** avant l'appel. Le code ARM sera donc :

```
.data
labelunique: .asciz string
```

```
.text
ldr r0, =labelunique
bl printf
```

entiers Pour les entiers, puisqu'on affiche toujours entier par entier on a un a une label dans la section *.data* qui existe toujours appelée *int* qui contient le string "%i" qui, par convention de **printf** va afficher le premier argument de type int en entier signé.Donc on charge ce label en **r0**, puis la valeur dans **r1**, supposons la valeur de l'entier est déjà dans **r2**, le code sera :

```
.data
int: .asciz "%i"
.text
ldr r0, =int
mov r1, r2
bl printf
```

flottants Pour les flottants, puisqu'on affiche toujours entier par entier on a un a une label dans la section *.data* qui existe toujours appelée *flottant* qui contient le string "%f" qui, par convention de **printf** va afficher le premier double passé en argument (stocké dans 2 registres en commença par **r2**) donc le premier double doit être dans le registre **r2**,**r3** et le reste doit être passé dans la pile.Pour transformer un flottant(32 bits) vers un double (64 bits) on utilise les registres et les instruction de *VFP*.En particulier *vmov* (équivalent du *mov* mais permet des registres **s0-s30** et aussi avec 3 opérandes cette instruction va copier les données du registre 64-bit du droite vers les 2 registres 32-bit de gauche.), *vcvt.f64.f32* (transfert vers le registre **d0-d15** de gauche la valeur flottante qui est dans un registre **s0-s31** à droite) . Supposons la valeur du flottant est déjà chargée dans **r2**, le code sera :

```
.data
flottant: .asciz "%f"
.text
ldr r0, =flottant
vmov s0, r2
vcvt.f64.f32 d0, s0
vmov r2, r3, d0
bl printf
```

Inputs

Entiers Pour les inputs, nous utilisons la fonction scanf de la librairie standard de C. Cette fonction prend en argument une adresse à laquelle stocker la valeur de l'entrée. Voici un exemple sur un fichier deca simple :

```
{
int x = readInt();
print(x);
}
```

Après compilation avec `decac`, on obtient le fichier assembleur ARM suivant :

```
/*start main program*/

.text
.global main
.extern printf
.extern scanf

/*ARM program*/
/*Beginning of variables declaration*/

.data
x: .int 0

.text
_varDeclAssign: /*fonction d'assignation de leur valeur a chacune de
/*declarees dans data au dessus. */
        push {lr}
        ldr r0, =int /*Chargement du caractere de formatage*/
        ldr r1, =tmpint /*Chargement de l'adresse de recuperation*/
        bl scanf    /*appel a la fonction de lecture de l'entree*/
        ldr r1, =tmpint    /* chargement de l'adresse de l'input*/
        ldr r0, [r1]    /*Lecture du contenu de l'input*/
        ldr r1, =x      /*Chargement du contenu de l'input*/
        str r0, [r1]    /*Stockage du contenu de l'input*/
        pop {pc}

/*Beginning of main ARM instructions:*/
main:
        push {ip, lr}
        bl _varDeclAssign
        ldr r0, =int
        ldr r1, =x
        ldr r1, [r1]
        bl printf
        pop {ip, pc}

.data
```

```
int: .asciz "%i"
tmpint: .int 0

/*end main program*/
```

Validation

Pour valider les programmes assembleurs générés, nous avons repris les tests de la partie SansObjet pour la machine IMA. Nous avons choisi d'utiliser les fichiers de tests *.deca* qui se trouvent dans **src/test/deca/codegen/valid/SansObjet** pour valider le bon fonctionnement de notre extension sur des cas où la sortie attendue est la même que celle de IMA. Pour valider la gestion d'erreur, nous n'avons pas trouvé comment récuperer une erreur d'overflow arithmetique sur des flottants. C'est déjà compliqué pour les entiers [4] parce que les flags ne sont pas mis à jour après des opération arithmétiques entre registres. Donc, les tests dans **src/test/deca/codegen/invalid/SansObjet** qui testent la division par 0 sont les seuls tests utilisés pour valider la gestion d'erreur de notre extensions ARM .

Résultats de la validation de l'extension

Les tests conduits passent et ont permis de détecter quelques erreurs que nous avons corrigées. En particulier la différence entre l'affichage des flottants de IMA et ARM, donc on a décidé de créer des fichier *.ARMres* qui vont stocker le résultat attendu de l'exécution d'un fichier assembleur ARM résultant de notre compilateur *decac*.

Implémentation de la partie SansObjet

Certains noms de variables sont interdits à l'utilisateur car ils servent déjà d'identifiant de variables temporaires nécessaires au bon fonctionnement du code assembleur.

- **Stockage des caractères de formatage pour l'affichage :** flottant, int, newline
- **Stockage temporaire des inputs :** tmpint, tmpfloat
- **Messages d'erreur :** division_by_zero_msg, float_arithmetic_overflow_msg, input_output_error_msg

Implémentation de la partie Objet

La partie nécessaire à la gestion des objets et des méthodes n'a pas été implémentée. Cependant, si l'utilisateur souhaite développer cette partie, voici quelques ressources [8].La principale différence avec le fonctionnement de la machine abstraite IMA est l'absence de tas. Toutes les variables temporaires sont stockées dans la pile. Les méthodes possèdent leur propres étiquettes. La difficulté est de bien sauvegarder les registres qui pourraient être écrasés avec l'appel d'une méthode.

Cycles

S : Cycles séquentiels

N : Cycles non séquentiels

I : Cycles internes

Instruction	nombre de cycles
BX	2S + 1N
B, BL	2S+N
CMP Normal Avec registre défini avec un décalage Avec PC réécrit Avec registre défini avec décalage + PC réécrit 2S + 1N + 1l	Varie selon les registres 1S 1S + 1l 2S + 1N
MUL	1S+ ml (m nombre de cycles 8 bits requis pour finir la multiplication varie en fonction du nombre et de la valeur des opérandes)
LDR	1S + 1N + 1l
STR	2N
SVC	2S + 1N

Références

[1] *ARM Vector Floating Point Processor*. 2022. URL : https://booksite.elsevier.com/9780128036983/content/Lecture%20Slides/Chapter_09.pdf.

[2] *arm-linux-gnueabihf-gcc-4.4(1) - GNU project C and C++ compiler*. URL : https://explainshell.com/explain/1/arm-linux-gnueabihf-gcc-4.4.

[3] *Conditions ARMv6*. 2022. URL : https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/condition-codes-4-floating-point-comparisons-using-vfp.

[4] *Fonctionnement de l'instruction MUL*. 2022. URL : https://developer.arm.com/documentation/dui0473/m/dom1361289882394.

[5] *gcc.gnu.org 3.19.5 ARM options*. URL : https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html#ARM-Options.

[6] *Instruction ARMv6*. 2022. URL : https://developer.arm.com/documentation/ddi0419/c/Application-Level-Architecture/The-ARMv6-M-Instruction-Set?lang=en.

[7] Phillip JOHNSTON. *Demystifying ARM Floating Point Compiler Options*. 2020. URL : https://embeddedartistry.com/blog/2017/10/11/demystifying-arm-floating-point-compiler-options/.

[8] *Procédures en langage d'assemblage*. 2022. URL : https://lig-membres.imag.fr/sicard/crALM/cours%207%20Procedure%20assembleur.pdf.

[9] Communauté WIKIPEDIA. *Architecture ARM*. 2022. URL : https://fr.wikipedia.org/wiki/Architecture_ARM.