Projet Génie Logiciel :

Document sur l'extension

Groupe 7 Equipe 34

BESSET Noé
FALL Rokhaya Yvette
GILLOT Aurélie
LESPINE Marilou
ROCHE Faustine

24 janvier 2024

Table des matières

Introduction

La documentation ci-dessous fournit des explications sur l'extension ARM qui est intégrée à notre projet de compilateur pour le langage Deca. Les fonctionnalités du compilateur, initialement conçu pour le langage assembleur fictif IMA, ont été étendues grâce à l'extension ARM. Ce choix stratégique est fondé sur notre désir de rendre notre compilateur compatible avec l'architecture ARMv7, une architecture beaucoup utilisée dans l'industrie des dispositifs embarqués, des smartphones aux systèmes embarqués complexes.

La documentation est conçue pour être complète et couvre plusieurs aspects de l'extension ARM, depuis sa spécification précise jusqu'à la validation de son intégration dans le compilateur. Les points abordés comprennent une description approfondie des décisions de conception et d'architecture, une spécification complète de l'extension, une analyse bibliographique pour contextualiser nos choix et des explications sur les algorithmes utilisés. De plus, nous présenterons la méthode de validation que nous avons utilisée et les résultats obtenus lors de cette étape importante du développement.

Cette documentation est destinée à devenir une ressource indispensable pour les développeurs, les contributeurs et toute personne intéressée par l'extension ARM de notre compilateur Deca. Les lecteurs comprendront mieux comment l'extension a été conçue, intégrée et validée en lisant ce document, ce qui améliore la transparence et la reproductibilité de notre travail. Nous espérons que cette documentation servira de guide complet pour ceux qui souhaitent comprendre, développer ou optimiser l'utilisation de l'extension ARM dans le contexte du compilateur Deca.

1 Spécification de l’extension

Qu’est ce que l'ARM ?

La société ARM Holdings a développé une architecture de processeur RISC (Reduced Instruction Set Computing) appelée ARM. L'acronyme ARM signifie à l'origine "Acorn RISC Machine". Cependant, depuis 1990, ARM est généralement considérée comme une machine "Advanced RISC" pour refléter l'évolution et l'expansion de cette architecture.

De nombreux dispositifs, tels que les smartphones et les tablettes, les systèmes embarqués, les dispositifs IoT (Internet des objets) et même certains serveurs, utilisent les processeurs d'ARM, qui sont très efficaces et économes en énergie. L'architecture ARM se distingue par son jeu d'instructions réduit et son optimisation pour des performances élevées tout en conservant une efficacité énergétique remarquable. Les processeurs basés sur l'architecture ARM sont largement utilisés dans l'industrie électronique en raison de leur puissance, de leur taille et de leur compacité.

Bien que varié, l'ensemble d'instructions ARM comprend plusieurs instructions couramment utilisées qui sont nécessaires à la programmation des processeurs ARM. Voici quelques-unes des commandes les plus courantes, accompagnées d'une brève explication de leur rôle :

MOV (Move) :
- Description : Déplace une valeur d'un registre vers un autre.
- Exemple : `MOV R1, #10` déplace la valeur 10 dans le registre R1.

ADD (Addition) :
- Description : Ajoute deux valeurs et stocke le résultat dans un registre.
- Exemple : `ADD R2, R1, #5` additionne la valeur de R1 avec 5 et stocke le résultat dans R2.

CMP (Compare) :
- Description : Compare deux valeurs sans modifier de registre, en affectant uniquement les indicateurs de drapeau.
- Exemple : `CMP R1, R2` compare les valeurs dans R1 et R2.

LDR (Load Register) :
- Description : Charge une valeur depuis la mémoire dans un registre.
- Exemple : `LDR R4, [R3]` charge la valeur à l'adresse stockée dans R3 dans le registre R4.

ARMv7

Dans notre choix d'intégrer l'architecture ARMv7 à notre extension ARM pour le compilateur Deca, il est essentiel de souligner que cette architecture 32 bits a succédé à l'architecture ARM11. Plusieurs profils sont inclus dans l'ARMv7, y compris le Cortex-A spécialisé dans les applications, le Cortex-M destiné aux voyages et le Cortex-R axé sur le temps réel. Chacun de ces profils répond à des exigences particulières et offre une grande flexibilité pour être utilisé dans divers domaines technologiques.

Plusieurs facteurs stratégiques ont influencé notre choix de centrer notre extension sur l'ARMv7 plutôt que sur l'architecture ARMv8, 64 bits. Bien que l'ARMv8 offre une architecture 64 bits plus récente, nous avons choisi délibérément l'ARMv7 pour plusieurs raisons.

Pour commencer, l'ARMv7 demeure une architecture moderne et fonctionnelle qui n'est pas obsolète. Sa pertinence dans de nombreuses applications est à l'origine de son adoption continue dans divers domaines. Nous nous assurons d'offrir une solution répondant aux besoins actuels sans les limitations liées à la gestion 64 bits en optant pour l'ARMv7.

De plus, nous évitons les difficultés liées à la prise en charge du 64 bits en choisissant délibérément l'ARMv7. Les architectures 64 bits présentent des problèmes supplémentaires pour la gestion de la mémoire, la taille des données et la compatibilité avec les logiciels existants. Nous simplifions le processus de développement en gardant l'ARMv7 tout en maintenant la compatibilité avec les dispositifs et les systèmes déjà en place qui reposent sur cette architecture 32 bits.

Bien sûr, la fonctionnalité 64 bits de l'ARMv8 améliore grandement la performance. Pour donner un exemple, supposons que nous ayons deux entiers de 64 bits, appelés A et B, et que nous voulions les additionner. Sur ARMv8, cette opération pourrait être effectuée de manière atomique avec une seule instruction d'addition. Cela réduit la complexité du code et améliore les performances, notamment pour les applications qui utilisent des entiers longs ou des opérations sur des données volumineuses.

Cependant, pour exécuter la même tâche sur l'ARMv7, chaque moitié des entiers 64 bits nécessiterait plusieurs instructions, ce qui entraînerait une surcharge de code et une utilisation moins efficace des ressources du processeur.

Le choix de la version 7 de ARM est donc une option pragmatique qui, malgré qu'elle réponde aux contraintes actuelles et simplifie le développement, peut être perçue comme une limitation. La raison de cette limitation est que le code assembleur généré pourrait ne pas bénéficier de toutes les optimisations disponibles avec l'architecture ARMv8 plus récente. Cependant, cette décision a été prise en tenant compte d'une approche qui équilibrait la facilité de développement, la compatibilité avec les systèmes existants basés sur l'ARMv7 et

le compromis relatif à certaines optimisations spécifiques de l'ARMv8 qui pourraient ne pas être exploitables dans notre contexte d'utilisation.

Motivations et buts

Pertinence du langage ARM : Le langage ARM est largement utilisé dans divers domaines technologiques, tels que les appareils mobiles, les appareils embarqués et l'Internet des objets (IoT). Nous voulons répondre à la demande croissante de performances optimisées sur les dispositifs équipés de processeurs ARM en incluant l'extension ARM dans notre compilateur Deca, offrant ainsi une solution adaptée aux réalités actuelles des industries technologiques.

Optimisation des performances : l'objectif principal de l'intégration de l'architecture ARM est d'améliorer les performances des dispositifs ARM particuliers en optimisant la génération de code. Les programmes compilés fonctionneront plus rapidement en raison de cette optimisation, ce qui améliore l'expérience utilisateur et l'utilisation des ressources matérielles.

Compatibilité avec les fonctionnalités spécifiques à ARM : La prise en charge des fonctionnalités uniques de l'architecture ARM sera incluse dans l'extension ARM que nous développons. Cela comprend la gestion efficace des registres spécialisés, l'utilisation des instructions SIMD (une seule instruction, plusieurs données) et d'autres fonctionnalités spécifiques à ARM. L'objectif est de tirer pleinement parti des capacités de l'architecture afin d'exécuter les programmes plus efficacement.

Compatibilité et portabilité : Bien que notre extension vise à optimiser le code pour les processeurs ARM, nous nous concentrons sur la portabilité. Notre objectif est de s'assurer que les applications compilées suivent les normes Java et le langage Deca, permettant ainsi leur utilisation sur divers processeurs ARM sans compromettre la cohérence du code.

Réduire la consommation d'énergie : L'optimisation du code pour les processeurs ARM vise également à réduire la consommation énergétique des applications en réponse aux préoccupations croissantes en matière de durabilité. Le but de cette initiative est d'utiliser les capacités d'optimisation énergétique de l'architecture ARM, ce qui contribuera à des solutions logicielles plus respectueuses de l'environnement.

Gestion des instructions spécifiques : l'extension ARM de notre compilateur Deca Java intégrera efficacement les instructions spécifiques à cette architecture dans le flux de compilation. L'objectif est de produire un code performant et correct en utilisant les caractéristiques d'ARM afin de garantir une exécution optimale des programmes générés.

2 Analyse bibliographique

ARMv7 est largement répandue dans les appareils mobiles et les systèmes embarqués depuis de nombreuses années. De nombreux dispositifs encore en circulation utilisent cette architecture. C'est pour cette raison que la documentation sur la version 7 de cette architecture est assez large, et nous avons pu trouver de nombreux cheat sheets afin de nous documenter sur les différentes instructions existantes.

En annexe 1 se trouve un des cheat sheets que nous avons utilisé afin de nous informer sur les différentes instructions comprises dans l'ARMv7. Dans ce cheat sheet se trouve chaque instruction existante dans l'ARMv7, mais aussi leur définition ainsi que les arguments impliqués. Cette documentation est claire et précise, puisque pour chaque argument est indiqué quel est son type, que ce soit un registre, une valeur immédiate ou une adresse. Cela nous a permis, pour chaque instruction en IMA, de retrouver un équivalent en ARMv7 afin d'arriver aux mêmes objectifs d'implémentation. Bien sûr, la plupart des instructions de cette doc ne nous a pas servi pour que ce nous voulions faire dans le temps imparti.

Prenons l'exemple de l'instruction ADD en IMA : **ADD dval, Rm : Rm <- V[Rm] + V[dval]**

```java
package fr.ensimag.ima.pseudocode.instructions;

import fr.ensimag.ima.pseudocode.BinaryInstructionDValToReg;
import fr.ensimag.ima.pseudocode.DVal;
import fr.ensimag.ima.pseudocode.GPRegister;

/**
 * @author Ensimag
 * @date 01/01/2024
 */
public class ADD extends BinaryInstructionDValToReg {
    public ADD(DVal op1, GPRegister op2) {
        super(op1, op2);
    }
}

```

ADD est donc une instruction binaire dans le langage IMA, puisqu'elle prend seulement 2 arguments, qui sont une DVal (à gauche) qui correspond à tout ce qui peut contenir une valeur, et un GPRegister correspondant à un registre (à droite).

Cette instruction, en langage ARM, est une instruction trinaire puisqu'elle prend 3 arguments (2 registres et un DVal) : **ADD Rm, Rn, dval : Rm <- V[Rn] + V[dval]**

src > main > java > fr > ensimag > arm > pseudocode > instructions > J ADD.java > ...

```java
package fr.ensimag.arm.pseudocode.instructions;

import fr.ensimag.arm.pseudocode.TernaryInstructionDValToRegARM;
import fr.ensimag.arm.pseudocode.DValARM;
import fr.ensimag.arm.pseudocode.GPRegisterARM;

/**
 * Adapted for ARM v7
 */
public class ADD extends TernaryInstructionDValToRegARM {
    public ADD(GPRegisterARM rd, GPRegisterARM rn, DValARM operand2) {
        super(rd, rn, operand2); // 3 operands for ARM
    }

    //operand2 can be a register, an immediate value or an offset
    //offset example : LDR R0, [R1, #4] (means load in r0 the value present
    //in the adress contained in r1 plus an offset of 4)

}
```

Il a donc fallu créer une nouvelle classe ADD dédiée à l'instruction du langage assembleur ARM, qui n'étend non plus une classe instruction binaire mais une classe instruction ternaire (qui n'existait pas pour IMA puisque les instructions prenaient maximum 2 arguments).

Pour un second exemple d'instructions qui diffèrent entre IMA et ARM, nous avons le branchement inconditionnel.
En IMA, l'instruction se présente de cette manière : **BRA dval : PC <- V[dval]**

```java
package fr.ensimag.ima.pseudocode.instructions;

import fr.ensimag.ima.pseudocode.BranchInstruction;
import fr.ensimag.ima.pseudocode.Label;

/**
 *
 * @author Ensimag
 * @date 01/01/2024
 */
public class BRA extends BranchInstruction {

    public BRA(Label op) {
        super(op);
    }

}
```

En ARM, c'est le nom du branchement inconditionnel n'est pas BRA mais seulement B. L'instruction se présente donc comme-ceci : **B dval : PC <- V[dval],** mais les arguments restent les mêmes.

```java
package fr.ensimag.arm.pseudocode.instructions;

import fr.ensimag.arm.pseudocode.BranchInstructionARM;
import fr.ensimag.arm.pseudocode.LabelARM;

/**
 * Corresponds to the BRA instruction for IMA. Unconditional Branch instruction in ARM v7.
 * @author Ensimag
 * @date 01/01/2024
 */
public class B extends BranchInstructionARM {

    public B(LabelARM op) {
        super(op);
    }

}
```

Parlons désormais des différences dans l'utilisation des registres entre l'architecture ARM et l'architecture IMA (Instruction Memory Architecture).

L'assembleur IMA utilise les registres suivants :

- GB (Global Base Register), qui contient à tout instant l'adresse précédant celle du premier mot de la pile
- LB (Local Base Register), qui contient l'adresse de la base locale de la pile
- SP (Stack Pointer), qui contient l'adresse du sommet de la pile

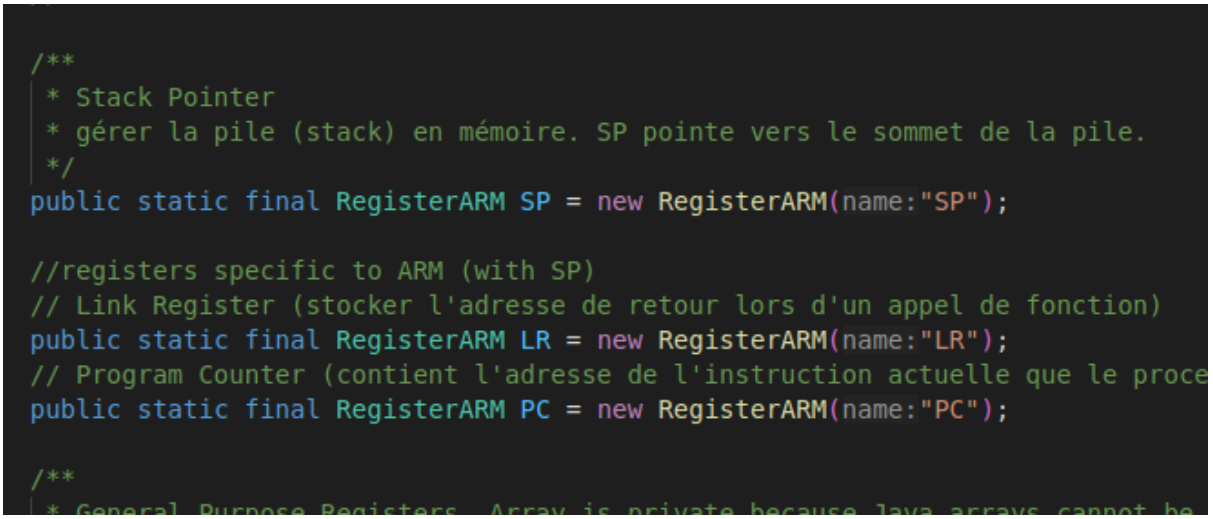
```java
/**
 * Global Base register
 */
public static final RegisterARM GB = new RegisterARM(name:"GB");
/**
 * Local Base register
 */
public static final RegisterARM LB = new RegisterARM(name:"LB");

//don't know if we still need GB and LB

/**
 * Stack Pointer
 * gérer la pile (stack) en mémoire. SP pointe vers le sommet de la pile.
 */
public static final RegisterARM SP = new RegisterARM(name:"SP");
```

Les registres GB et LB ne sont pas utilisés en ARM. Ajoutés au registre SP, sont les registres suivants :

- LR (Link Register), qui stocke l'adresse de retour lors d'un appel de fonction
- PC (Program Counter), qui contient l'adresse de l'instruction actuelle que le processeur est en train d'exécuter.



```java
/**
 * Stack Pointer
 * gérer la pile (stack) en mémoire. SP pointe vers le sommet de la pile.
 */
public static final RegisterARM SP = new RegisterARM(name:"SP");

//registers specific to ARM (with SP)
// Link Register (stocker l'adresse de retour lors d'un appel de fonction)
public static final RegisterARM LR = new RegisterARM(name:"LR");
// Program Counter (contient l'adresse de l'instruction actuelle que le proce
public static final RegisterARM PC = new RegisterARM(name:"PC");

/**
 * General Purpose Registers. Array is private because java arrays cannot be
```

La gestion de la pile est fortement influencée par cette différence fondamentale dans l'utilisation des registres. Pour garantir une exécution correcte des programmes, il est nécessaire d'adapter l'approche de génération de code assembleur car les pointeurs de pile diffèrent entre les architectures. Pour garantir la compatibilité et l'efficacité du code entre les différentes architectures, cela nécessite une réflexion approfondie.

Les fichiers Java du pseudo-code IMA ont dû être modifiés pour s'adapter à l'architecture ARM. Par exemple, la méthode display() du fichier Line.java a été reprise. Au départ, cette méthode était destinée à afficher les commentaires en IMA, mais elle a été modifiée pour tenir compte de la structure différente des commentaires en ARM. En raison des différences dans la représentation des commentaires entre les deux architectures, des adaptations étaient nécessaires, ce qui nécessitait des ajustements minutieux pour garantir la cohérence du code et son bon fonctionnement sur l'architecture ARM.

3 Nos choix de conception, d'architecture et d’algorithmes.

Il faut suivre plusieurs étapes pour modifier le générateur de code afin de produire du code ARM plutôt que du code IMA. Pour spécifier cette modification, nous avons d'abord ajouté une option -arm à la commande decac. Il a ensuite fallu créer de nouvelles classes pour représenter les instructions ARM et le pseudocode associé. Il a également fallu ajouter de nouvelles méthodes pour la version ARM du codeGenInst(), telles que le codeARMGen(), afin de générer du code ARM de la même manière que les méthodes pour le code IMA qui étaient actuellement utilisées.

Nous avons également examiné comment utiliser la logique pour imprimer des chaînes de caractères. Une idée est de créer une nouvelle classe nommée ListAllString dans la section de génération de code (codegen). Cette classe pourrait être utilisée comme liste de chaînes pour stocker toutes les chaînes de caractères qui sont imprimées. De plus, chaque élément de cette liste devrait inclure le nombre de chaînes, permettant de créer des noms de variables distincts, comme "message12" pour la chaîne 12.

Enfin, à la fin du processus de génération de code, nous voulions parcourir la liste des chaînes stockées dans la classe ListAllString et les ajouter à la section.data du fichier de sortie, en veillant à ce que les noms de variables soient uniques en utilisant le nombre associé à chaque chaîne.

En utilisant une approche modulaire et organisée, ces ajustements devraient permettre à votre compilateur de prendre en charge la génération de code ARM et l'impression correcte des chaînes de caractères.

A titre d’exemple, on traduirait println(“Hello”, “!”) par :

```
.text
.global main

main:
   MOV R7, #4
   MOV R0, #1
   MOV R2, #5
   LDR R1, =message1
   SWI 0

   MOV R7, #4
   MOV R0, #1
   MOV R2, #1
   LDR R1, =message2
   SWI 0
```

```
    MOV R7, #4
    MOV R0, #1
    MOV R2, #1
    LDR R1, =newLine
    SWI 0

    MOV R7, #1
    SWI 0

.data
    message1: .ascii "Hello"
    message2: .ascii "!"
    newLine: .ascii "\n"

.end
```

Par manque de temps, nous n'avons pas pu implémenter le code qui nous permettrait de compiler en ARM l'impression de chaînes de caractères, mais nous tenions tout de même à partager la réflexion faite pendant le temps qu'il nous restait.

4 Notre méthode de validation

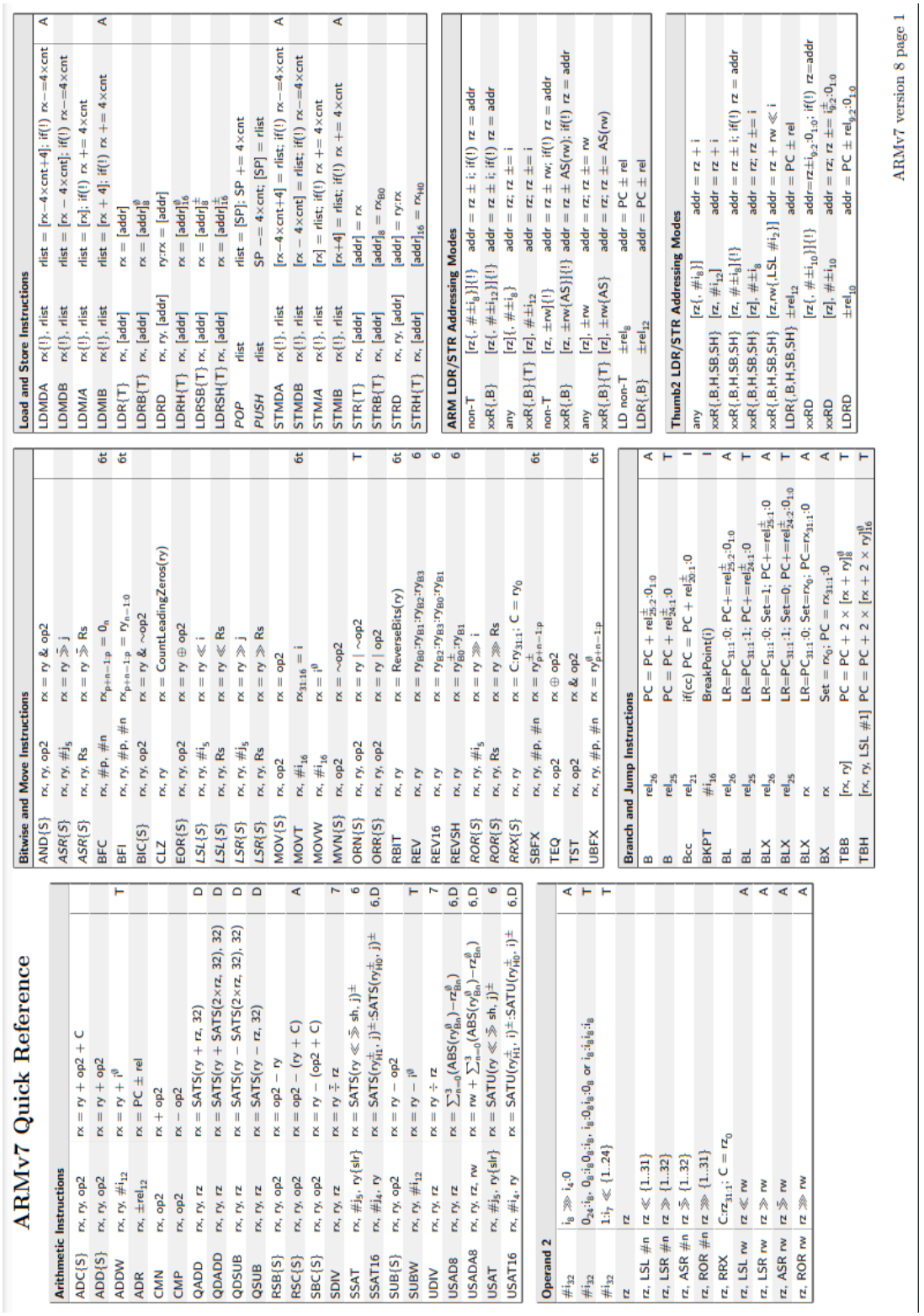
Pour la validation, nous avons décidé de reprendre les tests que nous avions écrit pour codegen en écrivant un nouveau script similaire à **exec_GenCondTests** qui lance les commandes suivantes puis vérifie les erreurs et les résultats :

- decac -arm example.deca
- arm-linux-gnueabi-as -o example.o example.s
- arm-linux-gnueabi-gcc -o example example.o
- qemu-arm -L /usr/arm-linux-gnueabi example

Annexes

Annexe 1 : Cheat sheet pour la version 7 de ARM

Source : (https://courses.cs.washington.edu/courses/cse469/20wi/armv7.pdf)



ARMv7 Quick Reference

Arithmetic Instructions

ADC{S}	rx, ry, op2	rx = ry + op2 + C	
ADD{S}	rx, ry, op2	rx = ry + op2	
ADDW	rx, ry, #i_{12}	rx = ry + i^0	T
ADR	rx, ±rel_{12}	rx = PC ± rel	
CMN	rx, op2	rx + op2	
CMP	rx, op2	rx − op2	
QADD	rx, ry, rz	rx = SATS(ry + rz, 32)	D
QDADD	rx, ry, rz	rx = SATS(ry + SATS(2×rz, 32), 32)	D
QDSUB	rx, ry, rz	rx = SATS(ry − SATS(2×rz, 32), 32)	D
QSUB	rx, ry, rz	rx = SATS(ry − rz, 32)	D
RSB{S}	rx, ry, op2	rx = op2 − ry	
RSC{S}	rx, ry, op2	rx = op2 − (ry + C)	A
SBC{S}	rx, ry, op2	rx = ry − (op2 + C)	
SDIV	rx, ry, rz	rx = ry ÷ rz	7
SSAT	rx, #i_5, ry{,sh}	rx = SATS(ry ≪ ≫ sh, j)±	6
SSAT16	rx, #i_4, ry	rx = SATS($ry^{\pm}_{Hi}$, j)±:SATS($ry^{\pm}_{Hi}$, j)±	6,D
SUB{S}	rx, ry, op2	rx = ry − op2	
SUBW	rx, ry, #i_{12}	rx = ry − i^0	T
UDIV	rx, ry, rz	rx = ry ÷ rz	7
USAD8	rx, ry, rz	rx = $\sum_{n=0}^{3}$(ABS(ry^0_{Bn})−rz^0_{Bn})	6,D
USADA8	rx, ry, rz, rw	rx = rw + $\sum_{n=0}^{3}$(ABS(ry^0_{Bn})−rz^0_{Bn})	6,D
USAT	rx, #i_5, ry{,sh}	rx = SATU(ry ≪ ≫ sh, j)±	6
USAT16	rx, #i_4, ry	rx = SATU($ry^{\pm}_{Hi}$, i)±:SATU($ry^{\pm}_{Hi}$, i)±	6,D

Operand 2

#i_{12}	$i_8 \gg i_4$:0	A
#i_{32}	0_{24}:i_8, 0_{31}:i_8:0_8:i_8, i_8:0_8:i_8:0_8 or i_8:i_8:i_8:i_8	T
#i_{32}	$1:i_7 \ll \{1..24\}$	T
rz	rz	
rz, LSL #n	rz ≪ {1..31}	
rz, LSR #n	rz ≫ {1..32}	
rz, ASR #n	rz ≫ {1..32}	
rz, ROR #n	rz ⋙ {1..31}	
rz, RRX	C:$rz_{31:1}$; C = rz_0	
rz, LSL rw	rz ≪ rw	A
rz, LSR rw	rz ≫ rw	A
rz, ASR rw	rz ≫ rw	A
rz, ROR rw	rz ⋙ rw	A

Bitwise and Move Instructions

AND{S}	rx, ry, op2	rx = ry & op2	
ASR{S}	rx, ry, #i_5	rx = ry ≫ j	
ASR{S}	rx, ry, Rs	rx = ry ≫ Rs	
BFC	rx, #p, #n	$rx_{p+n-1:p} = 0_n$	6t
BFI	rx, ry, #p, #n	$rx_{p+n-1:p} = ry_{n-1:0}$	6t
BIC{S}	rx, ry, op2	rx = ry & ~op2	
CLZ	rx, ry	rx = CountLeadingZeros(ry)	
EOR{S}	rx, ry, op2	rx = ry ⊕ op2	
LSL{S}	rx, ry, #i_5	rx = ry ≪ i	
LSL{S}	rx, ry, Rs	rx = ry ≪ Rs	
LSR{S}	rx, ry, #i_5	rx = ry ≫ j	
LSR{S}	rx, ry, Rs	rx = ry ≫ Rs	
MOV{S}	rx, op2	rx = op2	
MOVT	rx, #i_{16}	$rx_{31:16}$ = i	6t
MOVW	rx, #i_{16}	rx = i^0	
MVN{S}	rx, op2	rx = ~op2	
ORN{S}	rx, ry, op2	rx = ry | ~op2	T
ORR{S}	rx, ry, op2	rx = ry | op2	
RBIT	rx, ry	rx = ReverseBits(ry)	6t
REV	rx, ry	rx = ry_{B0}:ry_{B1}:ry_{B2}:ry_{B3}	6
REV16	rx, ry	rx = ry_{B2}:ry_{B3}:ry_{B0}:ry_{B1}	6
REVSH	rx, ry	rx = $ry^{\pm}_{B0}$:ry_{B1}	6
ROR{S}	rx, ry, #i_5	rx = ry ⋙ i	
ROR{S}	rx, ry, Rs	rx = ry ⋙ Rs	
RRX{S}	rx, ry	rx = C:$ry_{31:1}$; C = ry_0	
SBFX	rx, ry, #p, #n	rx = $ry^{\pm}_{p+n-1:p}$	6t
TEQ	rx, op2	rx ⊕ op2	
TST	rx, op2	rx & op2	
UBFX	rx, ry, #p, #n	rx = $ry^0_{p+n-1:p}$	6t

Branch and Jump Instructions

B	rel_{26}	PC = PC + $rel^{\pm}_{25:2}$:$0_{1:0}$	A
B	rel_{25}	PC = PC + $rel^{\pm}_{24:1}$:0	T
Bcc	rel_{21}	if(cc) PC = PC + $rel^{\pm}_{20:1}$:0	I
BKPT	#i_{16}	BreakPoint(i)	I
BL	rel_{26}	LR=$PC_{31:1}$:0; PC+=$rel^{\pm}_{25:2}$:$0_{1:0}$	A
BL	rel_{25}	LR=$PC_{31:1}$:1; PC+=$rel^{\pm}_{24:1}$:0	T
BLX	rel_{26}	LR=$PC_{31:1}$:0; Set=1; PC+=$rel^{\pm}_{25:1}$:0	A
BLX	rel_{25}	LR=$PC_{31:1}$:1; Set=0; PC+=$rel^{\pm}_{24:2}$:$0_{1:0}$	T
BLX	rx	LR=$PC_{31:1}$:0; Set=rx_0; PC=$rx_{31:1}$:0	A
BX	rx	Set = rx_0; PC = $rx_{31:1}$:0	A
TBB	[rx, ry]	PC = PC + 2 × [rx + ry]$^8_0$	T
TBH	[rx, ry, LSL #1]	PC = PC + 2 × [rx + 2 × ry]$^{16}_0$	T

Load and Store Instructions

LDMDA	rx{!}, rlist	rlist = [rx−4×cnt+4]; if(!) rx−=4×cnt	A
LDMDB	rx{!}, rlist	rlist = [rx − 4×cnt]; if(!) rx−=4×cnt	
LDMIA	rx{!}, rlist	rlist = [rx]; if(!) rx += 4×cnt	
LDMIB	rx{!}, rlist	rlist = [rx + 4]; if(!) rx += 4×cnt	A
LDR{T}	rx, [addr]	rx = [addr]	
LDRB{T}	rx, [addr]	rx = [addr]$^8_0$	
LDRD	rx, ry, [addr]	ry:rx = [addr]	
LDRH{T}	rx, [addr]	rx = [addr]$^{16}_0$	
LDRSB{T}	rx, [addr]	rx = [addr]$^{\pm 8}_0$	
LDRSH{T}	rx, [addr]	rx = [addr]$^{\pm 16}_0$	
POP	rlist	rlist = [SP]; SP += 4×cnt	
PUSH	rlist	SP −= 4×cnt; [SP] = rlist	
STMDA	rx{!}, rlist	[rx−4×cnt+4] = rlist; if(!) rx−=4×cnt	A
STMDB	rx{!}, rlist	[rx − 4×cnt] = rlist; if(!) rx−=4×cnt	
STMIA	rx{!}, rlist	[rx] = rlist; if(!) rx += 4×cnt	
STMIB	rx{!}, rlist	[rx+4] = rlist; if(!) rx += 4×cnt	A
STR{T}	rx, [addr]	[addr] = rx	
STRB{T}	rx, [addr]	[addr]$_8$ = rx_{B0}	
STRD	rx, ry, [addr]	[addr] = ry:rx	
STRH{T}	rx, [addr]	[addr]$_{16}$ = rx_{H0}	

ARM LDR/STR Addressing Modes

non-T	[rz{, #±i_8}]{!}	addr = rz ± i; if(!) rz = addr
xxR{,B}	[rz{, #±i_{12}}]{!}	addr = rz ± i; if(!) rz = addr
any	[rz]{, #±i_8}	addr = rz; rz ±= i
xxR{,B}{T}	[rz], #±i_{12}	addr = rz; rz ±= i
non-T	[rz, ±rw]{!}	addr = rz ± rw; if(!) rz = addr
xxR{,B}	[rz, ±rw{,AS}]{!}	addr = rz ± AS(rw); if(!) rz = addr
any	[rz], ±rw	addr = rz; rz ±= rw
xxR{,B}{T}	[rz], ±rw{,AS}	addr = rz; rz ±= AS(rw)
LD non-T	±rel_8	addr = PC ± rel
LDR{,B}	±rel_{12}	addr = PC ± rel

Thumb2 LDR/STR Addressing Modes

any	[rz{, #i_8}]	addr = rz + i
xxR{,B,H,SB,SH}	[rz, #i_{12}]	addr = rz + i
xxR{,B,H,SB,SH}	[rz, #±i_8]{!}	addr = rz ± i; if(!) rz = addr
xxR{,B,H,SB,SH}	[rz], #±i_8	addr = rz; rz ±= i
xxR{,B,H,SB,SH}	[rz,rw{,LSL #i_2}]	addr = rz + rw ≪ i
LDR{,B,H,SB,SH}	±rel_{12}	addr = PC ± rel
xxRD	[rz{, #±i_{10}}]{!}	addr=rz±$i_{9:2}$:$0_{1:0}$; if(!) rz=addr
xxRD	[rz], #±i_{10}	addr = rz; rz ±= $i^{\pm}_{9:2}$:$0_{1:0}$
LDRD	±rel_{10}	addr = PC ± $rel^{\pm}_{9:2}$:$0_{1:0}$

Multiplication Instructions			
MLA	rx, ry, rz, rw	$rx = rw + ry \times rz$	
MLA{S}	rx, ry, rz, rw	$rx = rw + ry \times rz$	A
MLS	rx, ry, rz, rw	$rx = rw - ry \times rz$	6t
MUL	rx, ry, rz	$rx = ry \times rz$	
MUL{S}	rx, ry, rz	$rx = ry \times rz$	A
SMLAxy	rx, ry, rz, rw	$rx = rw + ry^{\pm}_{Hx} \bar{\times} rz^{\pm}_{Hy}$	D
SMLaD	rx, ry, rz, rw	$rx = rw + ry^{\pm}_{H0} \bar{\times} rz^{\pm}_{H0} \pm ry^{\pm}_{H1} \bar{\times} rz^{\pm}_{H1}$	6,D
SMLaDX	rx, ry, rz, rw	$rx = rw + ry^{\pm}_{H0} \bar{\times} rz^{\pm}_{H1} \pm ry^{\pm}_{H1} \bar{\times} rz^{\pm}_{H0}$	D
SMLaLD	rx, ry, rz, rw	$ry:rx \mathrel{+}= rz^{\pm}_{H0} \bar{\times} rw^{\pm}_{H0} \pm rz^{\pm}_{H1} \bar{\times} rw^{\pm}_{H1}$	6,D
SMLaLDX	rx, ry, rz, rw	$ry:rx \mathrel{+}= rz^{\pm}_{H0} \bar{\times} rw^{\pm}_{H1} \pm rz^{\pm}_{H1} \bar{\times} rw^{\pm}_{H0}$	D
SMLAL	rx, ry, rz, rw	$ry:rx \mathrel{+}= rz \bar{\times} rw$	
SMLAL{S}	rx, ry, rz, rw	$ry:rx \mathrel{+}= rz \bar{\times} rw$	A
SMLALxy	rx, ry, rz, rw	$ry:rx \mathrel{+}= rz^{\pm}_{Hx} \bar{\times} rw^{\pm}_{Hy}$	D
SMLAWy	rx, ry, rz, rw	$rx = rw + ry \bar{\times} rz^{\pm}_{Hy}$	D
SMMLa	rx, ry, rz, rw	$rx = rw \pm (ry \bar{\times} rz)_{63:32}$	6,D
SMMLaR	rx, ry, rz, rw	$rx = rw \pm (ry \bar{\times} rz + 0x80000000)_{63:32}$	D
SMMUL	rx, ry, rz	$rx = (ry \bar{\times} rz)_{63:32}$	6,D
SMMULR	rx, ry, rz	$rx = (ry \bar{\times} rz + 0x80000000)_{63:32}$	D
SMUaD	rx, ry, rz	$rx = ry^{\pm}_{H0} \bar{\times} rz^{\pm}_{H0} \pm ry^{\pm}_{H1} \bar{\times} rz^{\pm}_{H1}$	6,D
SMUaDX	rx, ry, rz	$rx = ry^{\pm}_{H0} \bar{\times} rz^{\pm}_{H1} \pm ry^{\pm}_{H1} \bar{\times} rz^{\pm}_{H0}$	D
SMULxy	rx, ry, rz	$rx = ry^{\pm}_{Hx} \bar{\times} rz^{\pm}_{Hy}$	D
SMULL	rx, ry, rz, rw	$ry:rx = rz \bar{\times} rw$	
SMULL{S}	rx, ry, rz, rw	$ry:rx = rz \bar{\times} rw$	A
SMULWy	rx, ry, rz	$rx = (ry \bar{\times} rz^{\pm}_{Hy})_{47:16}$	D
UMAAL	rx, ry, rz, rw	$ry:rx = ry + rx + rz \times rw$	D
UMLAL	rx, ry, rz, rw	$ry:rx \mathrel{+}= rz \times rw$	
UMULL	rx, ry, rz, rw	$ry:rx = rz \times rw$	

Parallel Instructions			
pADD16	rx, ry, rz	$for(n=0..1)\ rx_{Hn} = p(ry_{Hn} + rz_{Hn})$	6,D
pADD8	rx, ry, rz	$for(n=0..3)\ rx_{Bn} = p(ry_{Bn} + rz_{Bn})$	6,D
pASX	rx, ry, rz	$rx = p(ry_{H1} + rz_{H0}):p(ry_{H0} - rz_{H1})$	6,D
pSAX	rx, ry, rz	$rx = p(ry_{H1} - rz_{H0}):p(ry_{H0} + rz_{H1})$	6,D
pSUB16	rx, ry, rz	$for(n=0..1)\ rx_{Hn} = p(ry_{Hn} - rz_{Hn})$	6,D
pSUB8	rx, ry, rz	$for(n=0..3)\ rx_{Bn} = p(ry_{Bn} - rz_{Bn})$	6,D
SEL	rx, ry, rz	$for(n=0..3)\ rx_{Bn} = (GEn\ ?\ ry : rz)_{Bn}$	6,D

Parallel Instruction Prefixes	
Q	Signed operation, Results are saturated
S	Signed operation, Results are truncated
SH	Signed operation, Results are right shifted by one
U	Unsigned operation, Results are truncated
UH	Unsigned operation, Results are right shifted by one
UQ	Unsigned operation, Results are saturated

Packing and Unpacking Instructions			
PKHBT	rx, ry, rz{sl}	$rx = (rz \ll sh)_{H1}:ry_{H0}$	6,D
PKHTB	rx, ry, rz{sr}	$rx = ry_{H1}:(rz \bar{\gg} sh)_{H0}$	6,D
SXTAB	rx, ry, rz{rb}	$rx = ry + (rz \ggg sh)^{\pm}_{B0}$	6,D
SXTAB16	rx, ry, rz{rb}	$for(n=0..1)\ rx_{Hn}=ry_{Hn}+(rz \ggg sh)^{\pm}_{B2n}$	6,D
SXTAH	rx, ry, rz{rb}	$rx = ry + (rz \ggg sh)^{\pm}_{H0}$	6,D
SXTB	rx, ry{rb}	$rx = (ry \ggg sh)^{\pm}_{B0}$	6
SXTB16	rx, ry{rb}	$for(n=0..1)\ rx_{Hn} = (ry \ggg sh)^{\pm}_{B2n}$	6,D
SXTH	rx, ry{rb}	$rx = (ry \ggg sh)^{\pm}_{H0}$	6
UXTAB	rx, ry, rz{rb}	$rx = ry + (rz \ggg sh)^{\emptyset}_{B0}$	6,D
UXTAB16	rx, ry, rz{rb}	$for(n=0..1)\ rx_{Hn}=ry_{Hn}+(rz \ggg sh)^{\emptyset}_{B2n}$	6,D
UXTAH	rx, ry, rz{rb}	$rx = ry + (rz \ggg sh)^{\emptyset}_{H0}$	6,D
UXTB	rx, ry{rb}	$rx = (ry \ggg sh)^{\emptyset}_{B0}$	6
UXTB16	rx, ry{rb}	$for(n=0..1)\ rx_{Hn} = (ry \ggg sh)^{\emptyset}_{B2n}$	6,D
UXTH	rx, ry{rb}	$rx = (ry \ggg sh)^{\emptyset}_{H0}$	6

Exclusive Load and Store Instructions			
CLREX		ClearExclusiveLocal()	I,6k
LDREX	rx, [ry]	rx = [ry]; SetExclusiveMonitor	6k
LDREX	rx, [ry, #i_{10}]	$rx = [ry+i^{\emptyset}_{9:2}:0_{1:0}]$; SetExclusiveMonitor	T,6k
LDREXB	rx, [ry]	$rx = [ry]^{\emptyset}_{8}$; SetExclusiveMonitor	6k
LDREXD	rx, ry, [rz]	ry:rx = [rz]; SetExclusiveMonitor	6k
LDREXH	rx, [ry]	$rx = [ry]^{\emptyset}_{16}$; SetExclusiveMonitor	6k
STREX	rx,ry,[rz]	if(Pass) [rz] = ry; rx = Pass ? 1 : 0	6k
STREX	rx,ry,[rz,#i_{10}]	$if(Pass)\ [rz+i^{\emptyset}_{9:2}:0_{1:0}]=ry;\ rx=Pass?1:0$	T,6k
STREXB	rx,ry,[rz]	$if(Pass)\ [rz]_8 = ry_{B0};\ rx = Pass?1:0$	6k
STREXD	rx,ry,rz,[rw]	if(Pass) [rw] = rz:ry; rx = Pass?1:0	6k
STREXH	rx,ry,[rz]	$if(Pass)\ [rz]_{16} = ry_{H0};\ rx = Pass?1:0$	6k

System Instructions			
CPSI{D,E}	{aif}{, #mode}	{a}{i}{f} = (E ? 1: 0); MODE = mode	6
CPS	#mode	MODE = mode	6
ERET		PC = LR; CPSR = SPSR	7
HVC	#i_{16}	CallHypervisor(i)	7
MRS	rx, xPSR	rx = {CPSR,SPSR}	
MRS	rx, Rbanked	rx = Rbanked	7
MSR	xPSR, rx	{CPSR,SPSR} = rx	
MSR	Rbanked, rx	Rbanked = rx	7
MSR	xPSR_{cxsf}, i	$\{CPSR,SPSR\}_{f;s;x;c} = i_{f;s;x;c}$	A
MSR	xPSR_{cxsf}, rx	$\{CPSR,SPSR\}_{f;s;x;c} = rx_{f;s;x;c}$	
RFEdi	rx{!}	LDMdi rx{!}, {PC, CPSR}	
SMC	#i_4	CallSecureMonitor()	6k
SRSdi	SP{!}, #mode	STMdi SP_mode{!}, {LR, SPSR}	6

Special Instructions			
DBG	#i_4	DebugHint(i)	7
DMB	option	DataMemoryBarrier(option)	I,7
DSB	option	DataSynchronizationBarrier(option)	I,7
ISB	SY	InstructionSynchronizationBarrier(SY)	I,7
NOP			6k
PLD{W}	[addr]	PreloadData(addr)	
PLI	[addr]	PreloadInstr(addr)	7
SETEND	{BE/LE}	EndianState = {BE/LE}	I,6
SEV		SendEvent()	6k
SVC	#i_{24}	CallSupervisor()	A
UDF	#i_{16}	UndefinedException()	
WFE		WaitForEvent()	6k
WFI		WaitForInterrupt()	6k
YIELD		HintYield()	6k

Keys	
{S}	Optional suffix, if present update flags
{t}	Conditional for additional instructions (T or E)
{T}	LDR/STR instruction uses user privileges.
a	A or S to add or subtract operand.
x, y	Selects bottom (B) or top (T) half of register(s)
cc	Condition code (can suffix most ARM instructions)
di	DA, DB, IA or IB for decrease/increase before/after.
i, j	Immediate operand, range 0..max / 1..max+1
rx, ry, rz, rw	General register
Rbanked	Banked register
rlist	Comma separated list of registers within { }.
op2	Immediate or shifted register
xPSR	APSR, CPSR or SPSR
SAT{S,U}(x,b)	Saturated signed/unsigned b bit value
B{0,1,2,3}	Selected byte (bits 7:0, 15:8, 23:16 or 31:24)
H{0,1}	Selected half word (bits 15:0 or 31:16)
{rb}	Optional rotate (ROR 8, ROR 16 or ROR 24)
{slr}	Optional shift (LSL #{1..31} or ASR #{1..32})
{sl}	Optional left shift (LSL #{1..31})
{sr}	Optional right shift (ASR #{1..32})
{AS}	ARM shift or rotate (LSL/ROR #{1..31}, LSR/ASR #{1..32} or RRX)
$value^{\pm}$, $value^{\emptyset}$	Value is sign/zero extended
$\bar{\times}$ $\bar{\div}$ $\bar{\gg}$	Operation is signed

General Registers		
R0-R3		Arguments and return values (useable by Thumb16)
R4-R7		General purpose (must be preserved, useable by Thumb16)
R8-R11		General purpose registers (must be preserved)
R12	IP	Intra-procedure-call scratch register
R13	SP	Stack pointer
R14	LR	Return address
R15	PC	Program counter

Condition Codes		
EQ	Equal	Z
NE	Not equal	!Z
CS/HS	Carry set, Unsigned higher or same	C
CC/LO	Carry clear, Unsigned lower	!C
MI	Minus, Negative	N
PL	Plus, Positive or zero	!N
VS	Overflow	V
VC	No overflow	!V
HI	Unsigned higher	C & !Z
LS	Unsigned lower or same	!C | Z
GE	Signed greater than or equal	N = V
LT	Signed less than	N ≠ V
GT	Signed greater than	!Z & N = V
LE	Signed less than or equal	Z | N ≠ V
AL	Always (default)	1

DMB and DSB Options	
SY	Full system, Read and write
(SY)ST	Full system, Write only
ISH	Inner shareable, Read and write
ISHST	Inner shareable, Write only
NSH	Non-shareable, Read and write
NSHST	Non-shareable, Write only
OSH	Outer shareable, Read and write
OSHST	Outer sharable, Write only

Notes for Instruction Set	
6,6k,6t,7	Introduced in ARMv6, ARMv6k, ARMv6T2, or ARMv7
A	Only available in ARM mode
D	Not available on ARM-M without DSP extension
H	Thumb16 instruction can use high registers
I	Can't be conditional
S	Thumb16 instruction must have S suffix unless in IT block
T	Only available in Thumb mode

Thumb16 Bitwise and Move Instructions			
AND{S}	rx, ry	rx = rx & ry	S
ASR{S}	rx, ry, #j_5	rx = ry $\gg$ j	S
ASR{S}	rx, ry	rx = rx $\gg$ ry	S
BIC{S}	rx, ry	rx = rx & ~ry	S
EOR{S}	rx, ry	rx = rx $\oplus$ ry	S
LSL{S}	rx, ry, #i_5	rx = ry $\ll$ i	S
LSL{S}	rx, ry	rx = rx $\ll$ ry	S
LSR{S}	rx, ry, #j_5	rx = ry $\gg$ j	S
LSR{S}	rx, ry	rx = rx $\gg$ ry	S
MOV	rx, ry	rx = ry	H
MOVS	rx, ry	rx = ry	
MOV{S}	rx, #i_8	rx = $i^{\emptyset}$	S
MVN{S}	rx, ry	rx = ~ry	S
ORR{S}	rx, ry	rx = rx | ry	S
REV	rx, ry	rx = $ry_{7:0}$:$ry_{15:8}$:$ry_{23:16}$:$ry_{31:24}$	6
REV16	rx, ry	rx = $ry_{23:16}$:$ry_{31:24}$:$ry_{7:0}$:$ry_{15:8}$	6
REVSH	rx, ry	rx = $ry^{\pm}_{7:0}$:$ry_{15:8}$	6
ROR{S}	rx, ry	rx = rx $\ggg$ ry	S
SXTB	rx, ry	rx = $ry^{\pm}_{7:0}$	
SXTH	rx, ry	rx = $ry^{\pm}_{15:0}$	
TST	rx, ry	rx & ry	
UXTB	rx, ry	rx = $ry^{\emptyset}_{7:0}$	6
UXTH	rx, ry	rx = $ry^{\emptyset}_{15:0}$	6

Thumb16 Branch and Special Instructions			
B	rel_{12}	PC = PC + $rel^{\pm}_{11:1}$:0	
Bcc	rel_9	if(cc) PC = PC + $rel^{\pm}_{8:1}$:0	I
BKPT	#i_8	BreakPoint(i)	I
BL	rel_{23}	LR=$PC_{31:1}$:1; PC+=$rel^{\pm}_{22:1}$:0	
BLX	rel_{23}	LR=$PC_{31:1}$:1; Set=0; PC+=$rel^{\pm}_{22:2}$:$0_{1:0}$	
BLX	rx	LR=$PC_{31:1}$:1; Set=rx_0; PC=$rx_{31:1}$:0	
BX	rx	Set=rx_0; PC = $rx_{31:1}$:0	
CBNZ	rx, rel_7	if(rx ≠ 0) PC += $rel^{\emptyset}_{6:1}$:0	I,6t
CBZ	rx, rel_7	if(rx = 0) PC += $rel^{\emptyset}_{6:1}$:0	I,6t
CPSI{D,E}	{aif}	{a}{i}{f} = (E ? 1 : 0)	6
IT{t{t{t}}}	cc	if(cc) NextInstruction	I,6t
NOP			6k
SETEND	{BE/LE}	EndianState = {BE/LE}	I,6
SEV		SendEvent()	7
SVC	#i_8	CallSupervisor()	
UDF	#i_8	UndefinedException()	
WFE		WaitForEvent()	7
WFI		WaitForInterrupt()	7
YIELD		HintYield()	7

Thumb16 Arithmetic Instructions			
ADC{S}	rx, ry	rx = rx + ry + C	S
ADD{S}	rx, ry, #i_3	rx = ry + $i^{\emptyset}$	S
ADD{S}	rx, #i_8	rx = rx + $i^{\emptyset}$	S
ADD{S}	rx, ry, rz	rx = ry + rz	S
ADD	rx, ry	rx = rx + ry	H
ADD	rx, SP, #i_8	rx = SP + $i^{\emptyset}$	
ADD	SP, #i_9	SP = SP + $i^{\emptyset}_{8:2}$:$0_{1:0}$	
ADR	rx, rel_{10}	rx = PC + $rel^{\emptyset}_{9:2}$:$0_{1:0}$	
CMN	rx, ry	rx + ry	
CMP	rx, #i_8	rx − $i^{\emptyset}$	
CMP	rx, ry	rx − ry	H
MUL{S}	rx, ry	rx = rx × ry	S
RSB{S}	rx, ry, #0	rx = 0 − ry	S
SBC{S}	rx, ry	rx = rx − (ry + C)	S
SUB{S}	rx, ry, #i_3	rx = ry − $i^{\emptyset}$	S
SUB{S}	rx, #i_8	rx = rx − $i^{\emptyset}$	S
SUB{S}	rx, ry, rz	rx = ry − rz	S
SUB	SP, #i_9	SP = SP − $i^{\emptyset}_{8:2}$:$0_{1:0}$	

Thumb16 Load and Store Instructions			
LDM*IA*	rx{!}, rlist	rlist = [rx]; if(!) rx += 4×cnt	
LDMIA	SP!, rlist	rlist = [SP]; SP += 4×cnt	
LDR	rx, [ry{, #i_7}]	rx = [ry + $i^{\emptyset}_{6:2}$:$0_{1:0}$]	
LDR	rx, [SP{, #i_{10}}]	rx = [SP + $i^{\emptyset}_{9:2}$:$0_{1:0}$]	
LDR	rx, rel_{10}	rx = [PC + $rel^{\emptyset}_{9:2}$:$0_{1:0}$]	
LDR	rx, [ry, rz]	rx = [ry + rz]	
LDRB	rx, [ry{, #i_5}]	rx = [ry + $i^{\emptyset}$]$^{\emptyset}_8$	
LDRB	rx, [ry, rz]	rx = [ry + rz]$^{\emptyset}_8$	
LDRH	rx, [ry{, #i_6}]	rx = [ry + $i^{\emptyset}_{5:1}$:0]$^{\emptyset}_{16}$	
LDRH	rx, [ry, rz]	rx = [ry + rz]$^{\emptyset}_{16}$	
LDRSB	rx, [ry, rz]	rx = [ry + rz]$^{\pm}_8$	
LDRSH	rx, [ry, rz]	rx = [ry + rz]$^{\pm}_{16}$	
POP	rlist	rlist = [SP]; SP += 4×cnt	
PUSH	rlist	SP −= 4×cnt; [SP] = rlist	
STM*IA*	rx!, rlist	[rx] = rlist; rx += 4×cnt	
STMDB	SP!, rlist	SP −= 4×cnt; [SP] = rlist	
STR	rx, [ry{, #i_7}]	[ry + $i^{\emptyset}_{6:2}$:$0_{1:0}$] = rx	
STR	rx, [SP{, #i_{10}}]	[SP + $i^{\emptyset}_{9:2}$:$0_{1:0}$] = rx	
STR	rx, [ry, rz]	[ry + rz] = rx	
STRB	rx, [ry{, #i_5}]	[ry + $i^{\emptyset}$]$_8$ = $rx_{7:0}$	
STRB	rx, [ry, rz]	[ry + rz]$_8$ = $rx_{7:0}$	
STRH	rx, [ry{, #i_6}]	[ry + $i^{\emptyset}_{5:1}$:0]$_{16}$ = $rx_{15:0}$	
STRH	rx, [ry, rz]	[ry + rz]$_{16}$ = $rx_{15:0}$	

ARMv7-A & ARMv7-R System

Current Program Status Register (CPSR)			
M	0x0000001f	Processor Operating Mode	
T	0x00000020	Instruction set (JT: 00=ARM, 01=Thumb)	
F	0x00000040	FIQ exception masked	
I	0x00000080	IRQ exception masked	
A	0x00000100	Asynchronous abort masked	6
E	0x00000200	Big-endian operation	6
IT	0x0600fc00	IT state bits	6t
GE{3..0}	0x000f0000	SIMD Greater than or equal to	6
J	0x01000000	Instr set (JT: 10=Jazelle, 11=ThumbEE)	6
Q	0x08000000	Cumulative saturation bit	
V	0x10000000	Overflow condition flag	
C	0x20000000	Carry condition flag	
Z	0x40000000	Zero condition flag	
N	0x80000000	Negative condition flag	

Processor Operating Modes			
usr	0x10	User	
fiq	0x11	FIQ	
irq	0x12	IRQ	
svc	0x13	Supervisor	
mon	0x16	Monitor (Secure only)	S
abt	0x17	Abort	
hyp	0x1a	Hypervisor (Non-secure only)	V
und	0x1b	Undefined	
sys	0x1f	System	

Vectors	
0x00	Reset
0x04	Undefined instruction
0x08	Supervisor Call / Secure Monitor Call / Hypervisor Call
0x0c	Prefetch abort
0x10	Data abort
0x14	Hyp trap
0x18	IRQ interrupt
0x1c	FIQ interrupt

Notes for System Registers and Tables	
6,6k,6t,7	Introduced in ARMv6, ARMv6k, ARMv6T2, or ARMv7
A	Only present on ARM-A
B	Banked between secure and non-secure usage
R	Only present on ARM-R
S	Only present with security extensions (Implies 6k,A)
V	Only present with virtualization extensions (Implies 7,A)

System Control Register (SCTLR)			
M	0x00000001	MMU enabled	B
A	0x00000002	Alignment check enabled	B
C	0x00000004	Data and unified caches enabled	B
CP15BEN	0x00000020	CP15 barrier enable	7,B
SW	0x00000400	Enable SWP and SWPB instructions	6,B
Z	0x00000800	Program flow prediction enabled	B
I	0x00001000	Instruction cache enabled	B
V	0x00002000	High exception vectors	B
RR	0x00004000	Round Robin select (Non-Secure RO)	
HA	0x00020000	Hardware access flag enable	B,S
BR	0x00020000	Background region enable	7,R
WXN	0x00080000	Write force to XN	V
DZ	0x00080000	Divide by zero causes undefined instruction	7,R
UWXN	0x00100000	Unprivileged write forced to XN for PL1	V
FI	0x00200000	Fast Interrupts (Non-Secure RO)	6
VE	0x01000000	Interrupt Vectors Enable	6,B
EE	0x02000000	Exception Endianess	6,B
NMFI	0x08000000	Non-maskable FIQ support (RO)	6
TRE	0x10000000	TEX remap functionality enabled	B,S
AFE	0x20000000	Access flag enable	B,S
TE	0x40000000	Thumb exception enable	6t,B
IE	0x80000000	Big-endian byte order in instructions	7,R

Coprocessor Access Control Register (CPACR)		
CP{0..13}	$3 \ll (2 \times \{0..13\})$	CP{0..13} access (00=denied, 01=privileged mode only, 11=privileged or user mode)
TRCDIS	0x10000000	Disable CP14 access to trace registers
D32DIS	0x40000000	Disable use of D16-D31 registers
ASEDIS	0x80000000	Disable advanced SIMD functionality

CP15 System Control Registers			
SCTLR	c1,0,c0,0	System Control Register	
ACTLR	c1,0,c0,1	Auxiliary Control Register	6,B
CPACR	c1,0,c0,2	Coprocessor Access Control Register	6
SCR	c1,0,c1,0	Secure Configuration (Secure only)	S
SDER	c1,0,c1,1	Secure Debug Enable (Secure only)	S
NSACR	c1,0,c1,2	Non-Secure Access Control (Non-Secure RO)	S

CP15 Security Extension Registers (ARM-A Only)			
VBAR	c12,0,c0,0	Vector Base Register	B
MVBAR	c12,0,c0,1	Monitor Vector Base Address (Secure only)	
ISR	c12,0,c1,0	Interrupt Status Register (RO)	

Secure Configuration Register (SCR)			
NS	0x001	System state is non-secure unless in Monitor mode	
IRQ	0x002	IRQs taken to Monitor mode	
FIQ	0x004	FIQs taken to Monitor mode	
EA	0x008	External aborts taken to Monitor mode	
FW	0x010	CPSR.F writeable in non-secure state	
AW	0x020	CPSR.A writeable in non-secure state	
nET	0x040	Disable early termination	
SCD	0x080	Secure monitor call disable	V
HCE	0x100	Hyp Call enable	V
SIF	0x200	Secure instruction fetch	V

Non-Secure Access Control Register (NSACR)		
CP{0..13}	$1 \ll \{0..13\}$	CP{0..13} can be accessed in non-secure state
NSD32DIS	0x00004000	CPACR.D32DIS is fixed 1 in non-secure state
NSASEDIS	0x00008000	CPACR.ASEDIS is fixed 1 in non-secure state
RFR	0x00080000	Reserve FIQ mode for non-secure
NSTRCDIS	0x00100000	Disable non-secure access to CP14 trace regs

CP15 Memory System Fault Registers			
DFSR	c5,0,c0,0	Data Fault Status Register	B
IFSR	c5,0,c0,1	Instruction Fault Status Register	6,B
ADFSR	c5,0,c1,0	Auxiliary DFSR	7,B
AIFSR	c5,0,c1,1	Auxiliary IFSR	7,B
DFAR	c6,0,c0,0	Data Fault Address Register	B
IFAR	c6,0,c0,2	Instruction Fault Address Register	6,B
DRBAR	c6,0,c1,0	Data Region Base Address Register	R
IRBAR	c6,0,c1,1	Instruction Region Base Address Register	R
DRSR	c6,0,c1,2	Data Region Size and Enable Register	R
IRSR	c6,0,c1,3	Instruction Region Size and Enable Register	R
DRACR	c6,0,c1,4	Data Region Access Control Register	R
IRACR	c6,0,c1,5	Instruction Region Access Control Register	R
RGNR	c6,0,c2,0	MPU Region Number Register	R

CP15 Generic Timer Registers			
CNTFRQ	c14,0,c0,0	Counter Frequency Reg (Non-Secure RO)	7
CNTKCTL	c14,0,c1,0	Timer PL1 Control Register	7
CNTP_TVAL	c14,0,c2,0	PL1 Physical TimerValue Register	7,B
CNTP_CTL	c14,0,c2,1	PL1 Physical Timer Control Register	7,B
CNTV_TVAL	c14,0,c3,0	Virtual TimerValue Register	7
CNTV_CTL	c14,0,c3,1	Virtual TimerControl Register	7
CNTPCT	c14,0	Physical Count Register (RO)	7
CNTVCT	c14,1	Virtual Count Register (RO)	7
CNTP_CVAL	c14,2	PL1 Physical Timer CompareValue Register	7,B
CNTV_CVAL	c14,3	Virtual Timer CompareValue Register	7