Documentation sur l'extension
Groupe GL-24

Janvier 2023

Contents

1 Motivation :

Notre choix pour l'architecture ARM a été pris en se basant sur plusieurs raisons. Tout d'abord, l'architecture ARM est largement utilisée dans les systèmes embarqués ainsi que dans les systèmes industriels, tels que les téléphones, les ordinateurs, l'IOT. Ce qui en fait un choix pertinent pour un projet de compilateur.

Elle est aussi connue pour sa consommation d'énergie faible ainsi qu'une grande possibilité à être intégrée dans des systèmes hétérogènes. Ce choix respecte ainsi les valeurs de notre équipe en ce qui concerne le développement durable.

De plus, développer un compilateur qui marche dans un très grand nombre de machines nous paraît très intéressant. Cela nous permettra aussi d'apprendre beaucoup plus sur l'architecture et de prendre en considération les nuances entre les différentes machines.

D'un autre côté, la compréhension de l'architecture ARM est actuellement une compétence qui est très recherchée dans l'industrie informatique. L'utilisation de cette architecture est de plus en plus répandue dans les entreprises. En tant qu'ingénieur en informatique, il est important de connaître le fonctionnement de cette architecture et s'ouvrir sur les projets basés sur cette architecture.

En utilisant cette extension pour notre projet de compilateur, nous avons eu l'occasion de nous familiariser avec cette architecture et de développer nos compétences en conséquence, ce qui sera bénéfique pour notre avenir professionnel.

2 Organisation :

Pour la partie extension de notre projet de compilateur, nous avons adopté une méthode agile en utilisant des sprints pour organiser notre travail. Nous avons travaillé sur l'extension dès le début du projet jusqu'à sa fin en nous concentrant sur la génération de code pour l'architecture ARM.

- Pour le sprint 1 :Nous avons commencé par nous documenter sur l'architecture

ARM, ses caractéristiques et avons défini la méthode pour exécuter notre travail en utilisant l'émulateur des systèmes Qemu.
Nous avons également défini les paquetages nécessaires pour le projet et l'organisation du code pour réaliser nos objectifs.

- Pour le sprint 2 : Nous avons entamé le code en organisant plus les paquetages nécessaires et en effectuant les tâches qui concernent les 3 étapes (A , B et C) pour pouvoir exécuter le code "println('hello')".

- Pour le sprint 3 : Nous avons poursuivi le travail en nous concentrant sur la partie sans-objet, notamment les calculs arithmétiques. Nous avons également effectué des tests pour vérifier que le code était correct.

- Pour le sprint 4 : Nous avons finalisé le travail en intégrant toutes les parties et en effectuant les tests finaux. Nous avons également finalisé la documentation nécessaire pour que les utilisateurs puissent utiliser efficacement le compilateur généré mais surtout pour faciliter l'utilisation de notre produit.

3 Spécification de l'extension :

Notre extension a pour objectif de générer du code pour l'architecture ARM . Pour cela, nous avons choisi d'utiliser l'émulateur QEMU pour pouvoir exécuter notre code généré. Nous avons choisi d'ajouter les fonctionnalités suivantes à notre compilateur:

- Gestion de la partie sans-objet : ceci englobe les calculs arithmétiques (addition, soustraction, multiplication, division) ainsi que les opérations de décalage, la gestion des boucles et des conditions . Ainsi que l'affichage des résultats.

Ceci est consacré par l'ajout de l'option [-arm] pour le compilateur decac. Cela permet de compiler des fichiers .deca vers un assembleur compatible avec une machine ARMv7 32 bits .
Pour installer QEMU sur un ordinateur sous Linux, il est nécessaire de suivre les étapes suivantes :

- Ouvrez un terminal sur votre ordinateur et tapez la commande "sudo apt install gcc-arm-linux-gnueabi" pour installer les outils de compilation pour l'architecture ARM.

- Ensuite tapez la commande "sudo apt install qemu-user" pour installer QEMU qui est l'émulateur utilisé pour exécuter notre code généré.
- Une fois que les deux paquetages sont installés, pour exécuter votre fichier assembleur généré, vous pouvez utiliser la commande "./run_arm fichier.ass" où "fichier.ass" est le nom de votre fichier assembleur. Cette commande va utiliser l'émulateur QEMU pour exécuter le code contenu dans le fichier assembleur spécifié. Le script run_arm permet d'exécuter les assembleurs de l'ARM.

Il est important de noter que ces étapes d'installation et d'exécution sont valables pour les systèmes d'exploitation de type Linux. Pour d'autres systèmes d'exploitation, les étapes d'installation et d'exécution peuvent varier

4 Analyse bibliographique :

L'analyse bibliographique permet de comprendre les différents choix de conception, d'architecture et d'algorithmes que nous avons faits pour développer notre extension. Elle permet aussi de justifier les choix techniques utilisés . Pour l'analyse bibliographique, On s'est appuyé principalement sur les sources suivantes :

- " Les processeurs ARM - Architecture et langage d'assemblage : un livre de Jacques Jorda qui englobe les spécificités de l'architecture ARM.
- https://azeria-labs.com/writing-arm-assembly-part-1/: Ce site présente les bases de code assembleur ARM en ce qui concerne les différentes instructions possibles.
- https://chromium.googlesource.com/chromiumos/docs/+/master/constants/ syscalls.mdarm-32_bit_EABI : Ce site explique la procédure des appels du systemes qui nous permettent d'interagir avec l'entrée/sortie de l'utilisateur
- https://developer.arm.com/documentation/dui0068/b/ARM-Instruction-Reference : Il s'agit de la documentation de l'architecture ARM: Nous avons utilisé les documents de spécification de l'architecture ARM pour comprendre les caractéristiques de cette architecture et les instructions de jeu d'instructions qu'elle prend en charge.

En se documentant à travers les sources mentionnées ci-dessus, nous avons vite compris, que la machine IMA nous cache beaucoup de choses pour des raisons de simplicité. Et que des fonctionnalités qui peuvent sembler triviales à travers la machine IMA nécessite de bien comprendre les spécificités de l'architecture pour pouvoir les implémenter pour l'architecture ARM.

Notamment, les interactions avec le système d'exploitation, la gestion des interruptions, des opérations arithmétiques tels que la division et le modulo, les spécificités du codage des immédiates qui n'est pas aussi simple que IMA. Ainsi que la manipulation des flottants.

4.1 Division et modulo:

En effet, l'architecture ARM ne prend pas en charge l'instruction de division directement. C'est pourquoi, dans notre analyse bibliographique, nous avons également cherché des algorithmes pour gérer les divisions sur cette architecture.

Nous avons trouvé différents algorithmes tels que l'algorithme de division par soustraction ainsi que "algorithme de division d'approximation de Newton-Raphson.

En comparant entre ces différents algorithmes en fonction de leur de leur précision, on a pu choisir celui qui conviendrait le mieux à notre extension

4.2 Gestion des registres:

Notre analyse bibliographique a également porté sur la compréhension des registres en ARM. L'architecture ARM utilise des registres pour stocker les données utilisées par les instructions.

Il existe différents types de registres, tels que les registres généraux, les registres de contrôle et les registres de pointeur. Chacun de ces registres a des fonctions spécifiques et doit être utilisé de manière appropriée pour maximiser les performances

4.3 L'ensemble des instructions:

Nous avons également examiné les instructions de base utilisées en assembleur ARM. Ces instructions comprennent des opérations arithmétiques, des opérations de décalage, des opérations de comparaison, des sauts et des opérations de gestion de la mémoire.

Ces instructions peuvent être classifiées en instructions de branchement, de traitement de données ... La structure d'une instruction dépend de son type ainsi que le nombre des opérandes qu'elle nécessite.

4.4 Les modes ARM et THUMB:

Parmi les raisons qui jouent un rôle important dans l'efficacité de l'architecture ARM, on retrouve le fait qu'il nous présente deux modes possibles.

Le mode ARM dans lesquelles les instructions sont en 32 bits, et le mode THUMB dans laquelle les instructions sont codées en 16 bits. Nous on a pris le choix de travailler sur 32 bits pour une généralisation.

4.5 Le codage des immediats:

Les instructions sont codées en utilisant des champs de bits différents , ce qui permet de spécifier les différentes parties de l'instruction, comme les opérandes et les opérations à effectuer.

Pour respecter la forme des instructions de l'architecture ARM lors des opérations, nous disposons de 12 bits pour coder les immediats. 8 bits pour coder la mantisse et 4 bits pour coder le décalage qui est une rotation de deux pas.

4.6 La manipulation des flottants:

Pour pouvoir manipuler les nombres flottants, nous avons besoin d'avoir une unité de calcul flottant (FPU). La FPU est une unité de calcul dédiée qui est utilisée pour effectuer des opérations sur des nombres flottants, telles que les soustractions, les additions, les multiplications et les divisions.

Les nombres flottants sont plus complexes à manipuler qu'un nombre entier, la FPU permet de gagner en performance pour les calculs de nombres flottants, en utilisant des techniques de calculs plus adaptées à la nature de ces nombres.

4.7 Les branchements:

Les instructions de branchement permettent de manipuler le flot du code et d'effectuer des appels à des étiquettes qui peuvent être des fonctions, ou des sous programmes. Ils sont utilisés pour créer des structures de contrôle telles que les boucles et les conditions.

4.8 Structure du code:

Enfin, nous avons mis en œuvre une organisation générale du code assembleur en ARM. L'assembleur ARM utilise des sections différentes pour organiser le code. On cite par exemple la section data qui permet de déclarer les variables.

Il y a également la section global qui permet la définition des fonctions . Une autre section disponible est la section start qui indique le point d'entrée du programme. Ces différentes sections permettent de garder une structure claire, organisée et simple à comprendre dans le code source.

En résumé, à travers cette analyse bibliographique nous a permis de comprendre les spécificités de l'architecture ARM, les algorithmes disponibles pour gérer les divisions, les registres utilisés, les instructions de base et l'organisation générale du code assembleur.

Cette compréhension nous a facilité nos choix de conception pour ainsi pouvoir la mise en place de notre extension de manière efficace et optimisée.

5 Choix de conception, d'architecture et d'algorithmes:

On a choisi de mettre en place les paquetages arm.pseudocode et arm.pseudocode.instructions : ces paquetages qui contiennent les classes gérantes les spécificités liées à la mémoire de l'architecture ARM et aux instructions .

Pour le paquetage arm.pseudocode, il contient les classes suivantes:

- RegisterArm : cette classe définit les registres nécessaires pour l'architecture ARM . En effet, il existe différents types de registres qui ont chacune des rôles spécifiques:

- Le Global Base Register (GBR) est utilisé pour stocker l'adresse de base du segment de données global.
- Le Link Register (LR) est utilisé pour stocker l'adresse de retour lors d'un appel de fonction.
- Le Stack Pointer (SP) est utilisé pour stocker l'adresse de la dernière valeur empilée sur la pile.
- Le Program Counter (PC) est utilisé pour stocker l'adresse de la prochaine instruction à exécuter.
- L'Application Program Status Register (APSR) contient des informations sur l'état du système, comme les flags de comparaison.
- Enfin, les General Purpose Registers (GPR) sont utilisés pour stocker des données génériques et peuvent être utilisés pour stocker des valeurs intermédiaires lors de calculs.

Arm	Spécificités
R0 - R10	General Purpose
R11	Frame Pointer (FP)
R12	Intra Procedural Call
R13	Stack Pointer(SP)
R14	Link Register
R15	Program Counter (PC)
CPSR	Current Program State Register

Il contient également les méthodes nécessaires pour l'appel d'un registre suivant sa notation (getR(int)) ainsi que des raccourcis de commodité pour R[0] et R[1] pour accéder plus facilement au registre R[0] et R[1] qui sont parmi les registres généraux à usage général (GPR) utilisés pour stocker des données génériques dans l'architecture ARM .

- OperandArm : La classe OperandArm définit les opérandes pour les instructions de l'architecture ARM. Ces opérandes permettent de décrire les emplacements de mémoire sur lesquels les instructions opèrent.

Elle facilite également la manipulation et la vérification des opérandes lors de l'exécution des instructions.

- DValArm : une classe abstraite qui représente un opérande d'une instruction ARM. Elle hérite de la classe OperandArm et définit les méthodes abstraites "toString" et "display" . Ça permet d'avoir une opérande flexible qui peut être un registre ou un immédiat.

- LineArm : Il s'agit d'une classe qui représente une ligne de code dans un programme ARM. Elle peut contenir une instruction ARM, un commentaire ou un label.

- La méthode display(PrintStream s) a pour but d'afficher la ligne de code sous forme de chaîne de caractères qui respectent les spécifications de l'architecture ARM.
- Un constructeur de ligne de code "LineArm(LabelArm l, InstructionArm i, String s)" qui initialise chacun de ces éléments.
- La méthode "getInstruction()" qui retourne l'instruction évoqué dans une ligne de code LineArm

- LabelArm : une classe utilisée pour représenter une étiquette dans le code ARM. Elle contient un champ 'name' qui est le nom de l'étiquette, un constructeur "LabelArm(String name)" ainsi q'une méthode 'display' pour l'afficher dans une sortie stream.

- La méthode abstraite display(PrintStream s) a pour but afficher un label

- Un constructeur de label à travers un nom d'étiquette LabelArm(String name)

- DAddrArm : c'est une classe abstraite qui représente un opérande qui pointe vers un emplacement mémoire. Elle hérite de la classe DValArm qui est également abstraite. Elle est utilisée pour les instructions qui ont besoin d'un accès en mémoire.

- AbstractLineArm : une classe abstraite qui permet de représenter une ligne de code générale pour le programme ARM.

- La méthode abstraite display() a pour but afficher une ligne du code de notre programme.

- ArmProgram : cette classe abstraite représente un programme ARM, il s'agit de l'ensemble des lignes de code de type AbstractLineArm qui définit un code assembleur. Elle permet de stocker et de gérer les instructions de l'assembleur ainsi que les données utilisées par le programme.

- La méthode add(AbstractLineArm line) qui permet d'ajouter une ligne de code.
- La méthode addInstruction(InstructionArm i) qui permet d'ajouter une instruction.
- La méthode addFirst(LineArm l) ajoute au début du code assembleur une ligne de code.
- La méthode display() est utilisée pour afficher tous les lignes du code de notre programme.

Cette classe contient également une structure de données de type dictionnaire qui permet de stocker les données sous forme de variables avec des valeurs associées.Ce dictionnaire représente la section data évoquée. Ce qui permet de manipuler facilement les données pendant l'exécution du programme.

- InstructionArm : Il s'agit d'une classe abstraite qui permet la représentation d'une instruction pour l'architecture ARM.

- La méthode displayOperands() est utilisée pour afficher les opérandes dont nécessitent cette instruction.

- La méthode display() est utilisée pour afficher le nom de l'instruction et ses opérande en vérifiant que ces derniers ne sont pas nuls.

La classe instructionArm est utilisée pour définir les différentes instructions disponibles pour l'architecture ARM, telles que les instructions d'addition, de soustraction, de décalage, etc.

Elle permet également de définir les différents types d'opérandes utilisés par ces instructions (registre, constante ou adresses mémoire). Les classes qui étendent cette classe définissent de manière spécifique les opérandes et leur nombre employés pour chaque instruction. Ces dernières sont définies dans le paquetage arm.pseudocode.instructions.

- ImmediateIntegerArm : cette sous-classe de la classe DValAr un nombre immédiat dans une instruction ARM.

- Le constructeur "IImmediateIntegerArm(int value)" qui permet de construire un immédiat de type entier en lui indiquant sa valeur.
- Elle hérite de la classe DValArm et surcharge la méthode "display()" pour afficher le nombre immédiat pour l'architecture ARM en utilisant la directive .word.

Cette classe permet de gérer les opérandes de type entier immédiat pour les instructions de l'architecture ARM, comme par exemple pour les instructions de comparaison ou pour les opérations arithmétiques.

Elle est utilisée pour spécifier les valeurs immédiates dans les instructions ARM, ce qui permet d'optimiser les performances en utilisant directement ces valeurs et ainsi éviter de passer par la mémoire.

- ImmediateStringArm : une sous-classe de la classe abstraite DValArm. Elle définit un opérande de type chaîne de caractères immédiate pour une instruction ARM.

- Le constructeur "ImmediateStringArm(String value)" qui permet de construire un immédiat de type chaîne de caractères en lui indiquant sa valeur.
- La méthode display() permet l'affichage de la chaîne en utilisant la directive .ascii.

- BinaryInstructionArm : une classe qui représenre es instructions à deux opérandes. Elle définit les méthodes pour récupérer les deux opérandes (getOperand1() et getOperand2()) et pour afficher les opérandes (displayOperands() pour ce cas précis).

Elle prend en compte les opérandes lors de sa création en s'assurant qu'ils ne sont pas nuls en utilisant la méthode validate.notNull. Elle surcharge aussi la méthode display() pour afficher le nom de l'instruction et les opérandes.

- UnaryInstructionArm : Dans cette sous-classe de la classe InstructionArm, on défini les instructions à un seul opérande.

- La méthode displayOperands() est utilisée pour afficher l'opérande unique.
- La méthode display() est utilisée pour afficher le nom de l'instruction et son opérande en vérifiant que ce dernier qui est passé en paramètre n'est pas nulle.
- le constructeur qui permet de définir une instruction à unique opérande "UnaryInstructionArm(OperandArm operand)"
- La méthode getOperand() retourne l'opérande unique évoqué .

- TernaryInstructionArm : cette sous-classe de la classe InstructionArm définit les instructions à trois opérandes.

- La méthode displayOperands() est utilisée pour afficher les trois opérandes.
- La méthode la méthode display() est utilisée pour afficher le nom de l'instruction et ses trois opérandes en vérifiant que ces derniers ne sont pas nuls.
- le constructeur qui permet de définir une instruction à trois opérandes "TernaryInstructionArm(OperandArm op1, OperandArm op2, OperandArm op3) "

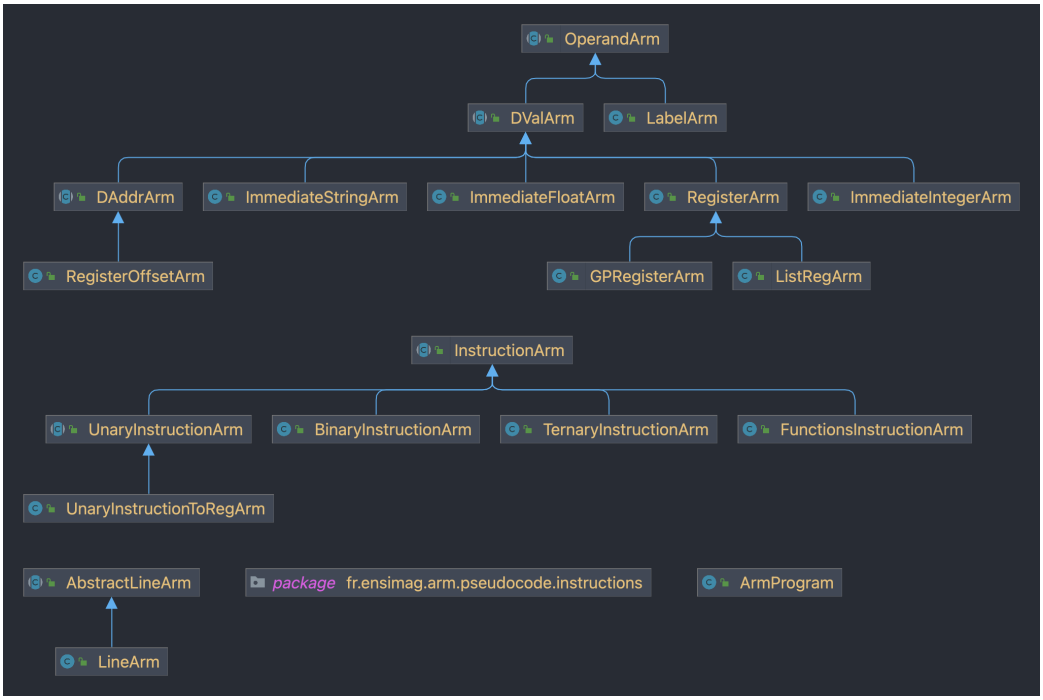
- RegisterOffsetArm: cette classe contient deux attributs :

1- "register" qui est un objet de la classe RegisterArm qui représente le registre .
2- "offset" qui est un entier qui représente le décalage

Les principales méthodes définies dans cette classe:

- La méthode getRegister() retourne le registre évoqué .
- La méthode getOffset() retourne l'entier de décalage.
- le constructeur qui permet de définir ce registre offset "RegisterOffsetArm(int offset, RegisterArm register)"

L'offset indique la position de décalage par rapport à l'emplacement mémoire associé au registre "register". Il permet de pointer vers une adresse mémoire spécifique en utilisant ce register.



Pour le paquetage arm.pseudocode.instructions, il contient les classes suivantes:

- ArmADD : une classe qui représente l'instruction ADD de l'architecture ARM. Elle hérite de la classe TernaryInstructionArm qui est une instruction avec 3 opérandes. L'instruction ADD prend en entrée 3 opérandes: un

registre (op1) qui va recevoir le résultat de l'addition, un deuxième registre (op2) et un troisième opérande (op3) qui peut être un nombre immédiat ou un autre registre.

- La méthode getName() retourne le nom de l'instruction qui est "add" dans ce cas.
- le constructeur permet de définir l'instruction d'addition en lui indiquant les 3 opérandes nécessaires "ArmADD(GPRegisterArm op1, GPRegisterArm op2, OperandArm op3)"

- ArmBeq : une sous-classe de UnaryInstructionArm qui constitue d'une instruction de saut conditionnel dans un programme ARM. Elle prend en entrée un unique opérande qui indique la destination du saut si la condition de comparaison précédente indique une égalité.

- La méthode getName() de cette classe renvoie "beq" qui est le nom de cette instruction dans la syntaxe ARM.
- le constructeur permet de définir l'instruction beq en lui indiquant l'unique opérande nécessairee "ArmBeq(OperandArm op)"

- ArmBgt : une classe qui hérite de UnaryInstructionArm, elle concerne les instructions de branchement en ARM. Elle permet d'éxecuter un saut conditionnel vers un bloc d'instructions définie par son étiquette si l'instruction de comparaison précédente indique qu'il y a une inégalité supérieure (r1 ¿= r2). Elle prend en entrée un unique opérande qui est utilisé pour spécifier l'adresse de saut.

- La méthode getName() de cette classe renvoie "bgt" qui est le nom de cette instruction dans la syntaxe ARM.
- le constructeur permet de définir l'instruction bgt en lui indiquant l'unique opérande nécessairee "ArmBgt(OperandArm op)"

- ArmBlt : une classe qui représente une instruction dans un programme ARM. Elle hérite de la classe UnaryInstructionArm qui elle-même hérite de la classe InstructionArm.
ArmBlt est utilisée pour spécifier un saut conditionnel vers un bloc d'instructions spécifié comme son unique paramètre si l'instruction de comparaison cmp précédente indique qu'il y a une inégalité inférieur entre ses deux opérandes (r1 ¡= r2).

- La méthode getName() de cette classe renvoie "blt" qui est le nom de cette instruction dans la syntaxe ARM.

- le constructeur permet de définir l'instruction blt en lui indiquant l'unique opérande nécessaire "ArmBlt(OperandArm op)". L'opérande passé en paramètre lors de la création de l'objet ArmBlt représente l'étiquette du bloc d'instruction vers laquelle le saut doit être effectué.

- ArmCMP : une sous-classe de BinaryInstructionArm à deux instructions qui représente l'instruction de comparaison CMP de l'architecture ARM. Elle prend en entrée deux opérandes :
1- Un registre général (GPRegisterArm)
2- Une valeur (DValArm)
Elle effectue une comparaison entre ces deux opérandes.

- La méthode getName() de cette classe renvoie "cmp" qui est le nom de cette instruction dans la syntaxe ARM.

- le constructeur permet de définir l'instruction cmp en lui indiquant les deux opérandes à comparer "ArmCMP(GPRegisterArm op1, DValArm op2)".

- ArmMUL : une classe qui fait partie des instructions permettant de réaliser des opérations arithmétiques. Cette sous-classe de la classe TernaryInstructionArm définit une instruction qui permet d'effectuer une opération de multiplication entre les opérandes op2 et op3 et stocke le résultat dans le registre op1.

- La méthode getName() de cette classe renvoie "mul" qui est le nom de cette instruction dans la syntaxe ARM.

- le constructeur permet de définir l'instruction mul en lui indiquant les trois opérandes obligatoires "ArmMUL(GPRegisterArm op1, GPRegisterArm op2, OperandArm op3)".

- ArmSUB : une classe qui fait partie des instructions permettant de réaliser des opérations arithmétiques. Cette classe permet de soustraire la valeur de l'opérande op3 à celle de l'opérande op2 et de stocker le résultat dans le registre de l'opérande op1.

- La méthode getName() de cette classe renvoie "sub" qui est le nom de cette instruction dans la syntaxe ARM.

- le constructeur permet de définir l'instruction sub en lui indiquant les trois opérandes obligatoires "ArmSUB(GPRegisterArm op1, GPRegisterArm op2, OperandArm op3)".

- LDR : une sous-classe de BinaryInstructionArm, elle a pour but de charger une constante 32-bit depuis la mémoire dans un registre cible spécifié. Elle prend en paramètres deux opérandes:
1- le premier est un GPRegisterArm qui est le registre cible
2- le deuxième est un LabelArm qui est l'étiquette de la constante à charger dans le registre cible.

- Elle surcharge la méthode displayOperands pour afficher l'opération de manière spécifique (avec " =" entre les opérandes).

- MOV : cette classe définit une instruction ARM mov qui permet de copier la valeur de l'opérande (op2) dans l'opérande (op1). Les opérandes 1 et 2 sont spécifiées en utilisant des opérandes de type GPRegisterArm et DValArm respectivement. Cette instruction permet le déplacement des données entre différents registres ou zones de mémoire.

- La méthode "MOV(GPRegisterArm op1, DValArm op2)" est le constructeur d'une instruction mov en lui indiquant les opérandes nécessaires avec les types définies.

- STR : Cette classe définit une instruction de l'architecture ARM qui permet le stockage de la valeur contenue dans un registre (op1) dans une adresse mémoire (op2). Elle hérite de la classe BinaryInstructionArm, ce qui implique qu'elle prend en compte deux opérandes.

- La méthode "STR(GPRegisterArm op1, OperandArm adr)" est le constructeur d'une instruction str en lui indiquant les opérandes nécessaires avec les types définies.

- SWI : une classe qui définit une instruction unaire qui représente un interrupteur. Elle prend en entrée un opérande unique DValArm qui est utilisé pour spécifier l'interruption à exécuter.

- La méthode "SWI(DValArm operand)" est le constructeur d'une instruction swi en lui indiquant l'unique opérande nécessaire de type DValArm.

Ce paquetage contient également des classes qui définissent des fonctions nécessaires et utiles pour le fonctionnement de la parties sans-objet. Il s'agit des algorithmes choisies pour permettre des calculs et des actions qui ne sont pas donnés par défaut par l'architecture ARM.
Parmi ces classes :

- RemArm : Cette classe permet de définir la fonction en assembleur qui effectue le modulo c'est-à-dire qui retourne le reste de la division entière de deux entiers.
En effet, on a par défaut :
1- r1 et r2 contiennent les valeurs du couple pour lesquels on calcule le modulo.
2- le regitre r0 est celui qui contient le resultat du modulo. Il est initialisé à 0 avant d'appeler la fonction.

- Cette classe contient la méthode "display(PrintStream s)" qui permet d'ajouter et d'afficher le bloc qui calcule le modulo "fct_rem_arm :",

Pour ce faire on a choisi l'algorithme des soustractions répetitifs :

Cet algorithme consiste à soustraire d'une manière répétitive le deuxième registre (r2) du premier registre (r1), en incrémentant à chaque fois un compteur (r0) pour compter le nombre de soustractions effectuées. D'où le but d'initialiser le registre r0 à O pour ne pas falsifier les résultats.

Un avantage derrière cet algorithme est qu'il est simple à mettre en œuvre parce qu'il nécessite une instruction arithmétique de base qui est l'instruction de soustraction.

- Cette classe contient également la redéfinition de la méthode equals(Object o) pour s'assurer si l'objet o est de type RemArm.

- Le constructeur ModuloArm() est également défini.

- QuoARM : Cette classe permet de définir la fonction en assembleur qui effectue la division entière c'est-à-dire qui retourne le quotient de la division entière de deux entiers.

En effet, on a par défaut :

1- r1 et r2 contiennent les valeurs du couple pour lesquels on calcule le modulo.

2- le regitre r0 est celui qui contient le resultat du modulo. Il est initialisé à 0 avant d'appeler la fonction.

3- le regitre r9 est celui utilisé pour les calculs au sein de la fonction.

- Cette classe contient la méthode "display(PrintStream s)" qui permet d'ajouter et d'afficher le bloc qui calcule le modulo "fct_quo_arm:",
- Cette classe contient également le constructeur QuoARM() est également défini.

- PrintIntegerArm : Cette classe permet de définir la fonction en assembleur qui effectue l'affichage d'un entier puisque cette fonctionnalié n'est pas possible par les instructions de base de l'architecture ARM

En effet, on a par défaut : le regitre r5 contient l'adresse de l'entier à afficher.

- Cette classe contient la méthode "display(PrintStream s)" qui permet d'ajouter et d'afficher les différents blocs qui permettent l'affichage d'un entier.
- Cette classe définit également le constructeur "constructeurPrintIntegerArm(OperandArm op)".

L'objectif est de convertir un entier en chaîne de caractères par des divisions répétitives de l'entier par 10 et en stockant chaque résultat obtenu. Cela permet de récupérer chacun des chiffres de l'entier dans l'ordre inverse.

Demanière plus détaillée, on retrouve parmi les blocs :

- _compteur: qui compte le nombre de digits sue contient l'entier pour savoir le nombre de division nécessaire;
- _fct: qui est la focntion principale
- _div10 : qui permet d'effectuer une division simple par 10
- _next: qui permet d'afficher le next digit

Les avantages de cet algorithme est sa simplicité. Mais on comprend qu'il est l'algorithme le plus efficace.

6 Méthode de validation :

Pour s'assurer de notre code assembleur géneré, On avait deux méthodes principale sui dépendent de notre avancement sur la partie extension:

- Avant avoir géré l'affichage des entiers, on a utilisé un simulateur et débogueur "CPUlator" qui fonctionne dans un navigateur web. Cet outil nous permet d'avoir une vue sur les registres et de savoir l'impact de chaque instruction sur les registres. Le lien de cet outil :

  https://cpulator.01xz.net

  Il suffit de choisir l'architecture ARMv7 pour démarrer.

- Après avoir réussi à afficher les entiers, on affiche les résultats des opérations à tester pour s'assurer du bon focntionnement des focntions.