Extention ARM

Benjelloun Otman
Birée Thomas
Boulouz Amine
El Goumiri Rida
Loginova Aleksandra

January 14, 2025

Spécification détaillée

L'architecture spécifique choisie est AArch64 (ARMv8-A). On commencera par implémenter la compilation vers ARMv7-A (32 bits). Passer à ARMv8-A ne devrait pas poser problème au vu de notre gestion des registres (c.f `ARMRegister.java` du paquetage `fr.ensimag.arm.pseudocode`)

- Architecture 64 bits (contrairement à l'architecture 32 bits de l'IMA)
- 31 registres généraux (contrairement aux 16 de l'IMA)
- Deux pointeurs de pile : SP et EL0_SP. On pense n'avoir besoin d'utiliser que SP, pas de code kernel level, de code privilégié, ou encore de multithreading. Context switch pour les deux à chaque fois au cas où. (Tout ça à vérifier). SI INTERRUPTIONS UTILISER EL0_SP.
- Niveaux d'exception : EL0 - EL4 (user, supervisor, hypervisor, virtualisation, secure modes). On n'aura normalement pas à gérer d'autre mode que EL0 (user).
- Rétrocompatibilité avec ARMv7 (32 bits). Algorithmes de génération de code les plus agnostiques possibles envers les architectures 32 bits (IMA, rétrocompatibilité) et 64 bits (ARMv8-A)
- Modes d'adressage similaires (immédiat, indexé par registre/décalage), mais plus, éventuellement non exploités.

Globalement, les différences entre l'assembleur IMA et l'assembleur ARMv8-A sont assez conséquentes, mais on peut gérer ces différences en se limitant aux fonctionnalités plus basiques des instructions ARM, en utilisant des algorithmes les plus agnostiques possibles vis-à-vis ces différences (quitte à sacrifier des performances, au moins dans un premier temps). Les différences plus notables (notamment la présence de deux pointeurs de piles et de niveaux d'exceptions) sont à gérer le moment venu, nous n'avons qu'une idée générale, possiblement éronnée, de comment s'y prendre.
Le gros du travail semble être dans la partie C (génération du code assembleur), mais la partie B peut être plus au moins concernée, pour la prise en compte d'exceptions par exemple.

Plateforme(s) d'éxecution du code ARM

Plusieurs options s'offrent à nous pour la vérification de l'éxecution du code ARM:

- Simulation en ligne de commande (QEMU et ARM Foundtion Model): Générer l'assembleur, utiliser ARM GCC Toolchain pour le linking, puis lancer le programme avec QEMU : `qemu-arm <path_to_prog`
- Machine virtuelle ARM (Encore QEMU, c'est bien). (Arch Linux ARM, Debian ARM, ou encore Ubuntu ARM)
- Raspberry Pi, même principe

Implémentation concrète (ARMv7-A)

- 16 Registres, dont tous les registres spéciaux, et donc moins de registres `General Purpose`. Détails:
  - `R0` : GP, argument and return values.
  - `R1 - R3` : GP, argument.
  - `R4 - R8` : GP, callee-saved (à préserver par les méthodes).
  - `R9 = SB` : Static Base (équivalent de `GB` en `IMA` dans notre cas).
  - `R10 = SL` : Stack Limit (utilisé pour vérifier le stack overflow).
  - `R11 = FP` : Frame Pointer (équivalent de `LB` en `IMA` dans notre cas).
  - `R12 = IP` : Intra-Procedure-Call scratch (Registre scratch pour lequel on n'attend aucune persistence).
  - `R13 = SP` : Stack Pointer.
  - `R14 = LR` : Link Register.
  - `R15 = PC` : Program Counter (Gestion des exceptions, probablement inutile dans notre cas).
- Transition vers ARMv8-A (au moins en mode 32 bits) facile (simplement adapter les registres, le reste est fait automatiquement).
- Utilisation de `SP` comme unique Stack Pointer, on ne gère pas l'exécution en user space / kernel R0.
- Création du paquetage `fr.ensimag.arm.pseudocode` où résident des classes analogues à celles du paquetage `fr.ensimag.ima.pseudocode`. Différences d'architecture gérées ici, et pendant la génération de code (utilisation des bonnes instructions, avec le bon format, avec nombre de registres GP plus limité, instructions ternaires, mêmes modes d'adressage)
- Ajout des méthodes `codeGenInstARM` et `codeGenPrintARM` et de l'argument `--arm` pour `decac`, si `--arm` est spécifié, les méthodes `codeGenInst` et `codeGenPrint` appelleront simplement leurs équivalents `ARM`. Le code est assez général et modulaire pour que cela soit (presque) le seul changement. (l'attribut `ARMProgram` est utilisé plutôt que `IMAProgram` dans `DecacCompiler`, avec les méthodes `addInstructionARM`, `addCommentARM` etc...).
- Certaines des classes présentes dans le paquetage `fr.ensimag.ima.pseudocode` sont dupliquées dans le paquetage `fr.ensimag.arm.pseudocode`, et cela est délibéré. Il nous est possible d'altérer la structure des deux paquetages pour faire en sorte que seules les classes qui différent soient présentes, et que celles communes n'aparaissent qu'une seule fois, mais cela n'a pas été jugé prioritaire, et n'est pour l'instant pas considéré.
- Parties A et B jugées définitivement non concernées à moins que l'on veuille implémenter du vrai 64 bits, ce qui changerais les limites permises pour les littéraux flottants et entiers acceptés, en utilisant toujours la norme IEEE-754. Cela nécessiterait donc un changement jugé pour l'instant assez mineur de la partie A.

Avancée actuelle : paquetage `fr.ensimag.arm.pseudocode` presque complet, avec le sous-paquetage `fr.ensimag.arm.pseudocode.instructions` entamé, il contient les instructions ARM (pas toutes, seulement celles dont on a besoin), et évoluera dynamiquement avec l'avancement du projet, dépendant des instructions dont on a besoin. Les méthodes `codeGen*ARM` ne sont pas encore entamées, mais au vu de la qualité du code, elles devraient être très similaires à celles d'IMA. Nous ne comptons évidemment pas utiliser toutes les fonctionnalités rajoutées de l'instruction set d'ARM.

Le langage déca ne faisant à priori aucun syscall, on déduit que la gestion du pointeur de pile `EL0_SP` n'est pas nécessaire. De même pour les niveaux d'excpetion `EL0 - EL4`, la gestion des interruptions software. La gestion de l'état du processeur n'est elle non plus pas nécessaire. L'Endianness ne devrait en l'occurence poser aucun problème.

IMPORTANT : les registres ARM ne sont pas "typés dynamiquement" comme ceux d'IMA, et donc il sera de notre responsabilité d'assure la cohérence des types. En soi, pour un compilateur Deca correctemtn implémenté, cela ne posera pas de problème, cependant, pendant le processus de développement, il faudra faire attention à cela. (Nous estimons qu'au vu de la qualité du code de génération de code pour l'IMA, nous ne rencontrerons aucun problème de typage à ce niveau-ci).