UFR IM²AG

UNIVERSITÉ Grenoble Alpes

PROG5 - Projet, année 2018-2019

Réalisation d'un éditeur de liens — Phase de réimplantation

L'objectif de ce projet est d'implémenter une sous partie d'un éditeur de liens. Plus précisément, le projet est centré sur la dernière phase, dite de *réimplantation*, exécutée par l'éditeur de liens. Le projet est structuré en étapes, avec la programmation de plusieurs outils intermédiaires, permettant de mieux comprendre les principales notions et de simplifier le découpage des tâches.

1 Introduction : édition de liens et réimplantation

La compilation de fichiers source écrits dans un langage de programmation est constituée de deux étapes : la traduction, qui produit un fichier objet (ou binaire) translatable pour chaque fichier source, et l'édition de liens, qui permet de manière très générale de regrouper et de charger à des addresses particulières le contenu des fichiers objet. Un fichier objet contient une traduction potentiellement incomplète du fichier source correspondant. Cette traduction est incomplète car certains symboles peuvent y avoir une valeur indéfinie. Ce qui est nommé *symbole*, dans un fichier objet, correspond, du point de vue du programme, à une adresse en mémoire. Au niveau du langage d'assemblage, ce sera donc une étiquette. À plus haut niveau, cela sera un nom de variable ou de fonction. Un fichier objet issu de la première étape de compilation est appelé *translatable* (ou encore *relogeable*) car il contient l'information permettant de le charger à n'importe quelle adresse en mémoire. Par définition, les symboles qu'il contient n'auront donc leur valeur définitive qu'une fois l'adresse de chargement en mémoire connue.

Le problème général de l'édition de liens consiste à produire, à partir d'un ensemble de fichiers objet translatables, un unique fichier binaire exécutable ou une bibliothèque de fonctions destinée à être incluse à d'autres programmes. Ce résultat est obtenu en deux étapes successives :

1. *Fusion* : Cette étape consiste à rassembler les différentes zones (ou sections) définies dans les fichiers objets donnés en entrée et dans les éventuelles bibliothèques à lier au programme. Il s'agit principalement de concaténer ces sections, mais il faut aussi tenir compte des effets produits par cette concaténation : changement de la valeur des symboles, renumérotation des symboles, etc. Une action importante réalisée au cours de cette étape consiste à mettre en correspondance les symboles utilisés dans l'un des fichiers mais définis dans un autre fichier. Le résultat de cette première étape est un fichier binaire translatable unique. Une fois la fusion réalisée, plusieurs situations sont possibles :
   - Il reste des symboles indéfinis (non résolus) dans le fichier résultat : l'étape de fusion doit être ré-itérée, le fichier résultat devant être fusionné avec au moins un autre fichier objet translatable ou une autre bibliothèque. Dans le cas contraire, l'édition de liens échoue.
   - Au moins un symbole est défini plusieurs fois : l'édition de liens échoue. [1]

1. En réalité, cela dépend du statut (faible/fort) des définitions, mais on ne traitera pas ces cas particuliers ici.

— Le fichier résultat ne comporte plus aucun symbole indéfini : la seconde étape de l'édition de liens peut prendre place.

2. *Implantation d'un fichier binaire translatable* : Dans cette étape, une valeur absolue et définitive est affectée à chaque symbole du fichier donné, en fonction de l'adresse d'implantation en mémoire prévue pour les différentes parties du programme. Cette opération suppose qu'il n'existe plus de symbole indéfini dans le fichier d'entrée. Le résultat de cette seconde étape est un fichier binaire exécutable et non translatable. Généralement, l'implantation ne se fait que lors de la génération de programmes exécutables ou lors de leur exécution (on parle alors de chargement). Les bibliothèques étant par nature destinées à être utilisées lors de la fusion, elles restent souvent sous forme translatable.

L'éditeur de liens présenté dans ce document fait partie de la suite d'outils de compilation GNU et produit du code destiné à être exécuté sur un processeur ARM. Il est capable d'effectuer de manière séparée les deux étapes de l'édition de liens : l'opération de fusion et l'opération d'implantation. L'objectif final de ce projet est de réaliser votre propre version du programme d'implantation.

Les bibliothèques mentionnées jusque là dans ce texte sont des bibliothèques dites *statiques*. Elles correspondent grossièrement à une collection d'objets archivés (format ar). Leur utilisation est la même que celle des objets eux-mêmes, au moment de la fusion. Il existe cependant un autre type de bibliothèques, les bibliothèques *dynamiques*, qui elles ont un fonctionnement différent. La résolution des symboles de ces bibliothèques est faite au moment du chargement du programme. Ces dernières bibliothèques (dynamiques) ne sont pas au programme de ce projet. Certaines fonctions classiques sont habituellement disponibles grâce à ces bibliothèques et un support du système d'exploitation, par exemple `printf`, `malloc`, ... Leur utilisation sera donc proscrite dans les objets que vous manipulerez avec votre programme d'implantation.

Exemple :

On considère deux fichiers `fich1.s` et `fich2.s` contenant du code source écrit en langage d'assemblage ARM. Pour produire les fichiers binaires translatables `fich1.o` et `fich2.o`, les commandes d'assemblage suivantes sont utilisées :

```
arm-eabi-as -o fich1.o fich1.s
arm-eabi-as -o fich2.o fich2.s
```

La fusion de ces deux fichiers est ensuite réalisée au moyen de la commande suivante :

```
arm-eabi-ld -r -o resultat.o fich1.o fich2.o
```

Cette commande produit le binaire translatable `resultat.o`.

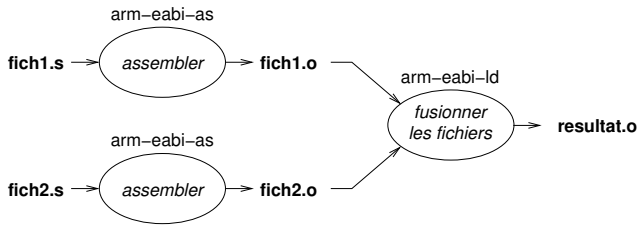

FIGURE 1 – Fusion de deux fichiers binaires translatables

À ce stade, on fait l'hypothèse que le fichier obtenu après fusion ne contient plus aucun symbole indéfini. L'étape d'implantation peut alors être réalisée au moyen de la commande suivante :

```
arm-eabi-ld --section-start .text=0x58 --section-start .data=0x1000 -o prog resultat.o
```

Dans cette commande, nous avons indiqué à `arm-eabi-ld` que la section `.text` sera chargée à l'adresse 0x58 et

que la section `.data` sera chargée à l'adresse 0x1000. Le resultat est le fichier `prog` qui n'est pas translatable : tous les symboles ont leur valeur définitive et il faudra le charger en mémoire aux adresses indiquées pour qu'il s'exécute correctement. L'objectif final de ce projet est de réaliser un programme exécutant cette étape d'implantation à la place de `arm-eabi-ld`.

Pour nous aider, nous disposons d'outils permettant d'examiner le contenu de fichiers objet ou de programmes exécutables : `arm-eabi-readelf` et `arm-eabi-objdump`. Ils nous permettront de mettre au point notre implanteur en examinant le résultat produit et, au besoin, en le comparant avec le résultat de `arm-eabi-ld` (figure 2).

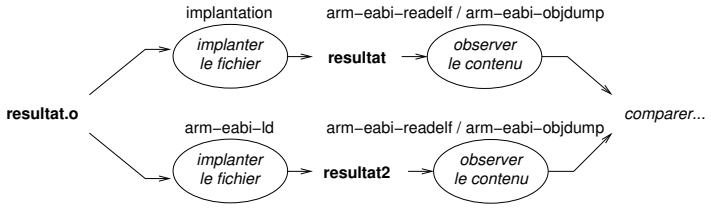

FIGURE 2 – Implantation d'un fichier binaire translatable

Les fichiers exécutables obtenus après implantation peuvent ensuite être exécutés. Se référer aux sections 3 et 4 pour les détails à ce sujet.

.

2 Principe général du projet

L'objectif final du projet est de développer un programme permettant de réaliser la *réimplantation* d'un fichier binaire translatable pour machine ARM (on parle aussi d'*implantation*). On utilise plutôt le terme *réimplantation*[2] car les symboles d'un fichier objet translatable ont une implantation par défaut, non définitive, que la phase d'implantation modifie. Un tel programme est généralement intégré comme module au sein d'un éditeur de liens complet mais cet aspect ne figure pas dans le cahier des charges du projet.

Pour simplifier le démarrage, le projet est décomposé en plusieurs étapes, décrites en section 4, constituant deux phases. La première phase est la phase de lecture d'un fichier objet. Elle contient des étapes dont les objectifs sont de développer de petits outils (similaires à des parties de `arm-eabi-readelf`) permettant d'examiner le contenu d'un fichier binaire au format ELF. Elles permettent de comprendre les notions de base de la spécification ELF, qui est un format de stockage d'un fichier objet ou d'un programme exécutable.[3] La seconde phase, et les étapes suivantes, concernent le travail de réimplantation proprement dit et nécessitent, pour la plupart, de comprendre les extensions du format ELF spécifiques au code binaire ARM.

Dans un premier temps, par souci de simplification, on fera l'hypothèse que les fichiers binaires manipulés ne contiennent pas de références vers des symboles fournis par des bibliothèques dynamiques (`printf`, `malloc`, ...). Cette hypothèse et la simplification associée impliquent que les fichiers binaires exécutables produits par l'outil final ne peuvent pas être exécutés directement par le simulateur `arm-eabi-run` utilisé habituellement dans le cadre des travaux pratiques d'ALM. À la place, un autre simulateur ARM vous est fourni, ainsi qu'une interface d'échange avec ce simulateur depuis un programme écrit en C. Des détails à ce sujet sont donnés en section 3.3.3. Par ailleurs, la toute dernière étape du projet concerne les modifications à apporter à l'implanteur afin de pouvoir exécuter les fichiers exécutables qu'il produit sur le simulateur `arm-eabi-run`.

2. L'expression correspondante en anglais est généralement *relocation*.

3. Le format ELF est utilisé par la plupart des systèmes Unix (Linux, FreeBSD, etc.) ainsi que d'autres systèmes (par exemple, les consoles de jeu Nintendo et Sony, ainsi que certains téléphones portables, notamment dans les versions récentes d'Android).

3 Matériel fourni

L'objectif principal de ce projet est de vous faire lire, comprendre et implémenter la spécification ELF pour les processeurs ARM, décrite dans les documentations standard associées. Ainsi, la principale resource dont vous disposerez sera constituée de ces documentations. Le code source qui vous est fourni est volontairement minimal et vise simplement à vous éviter de rencontrer des difficultés qui n'ont pas de rapport avec le thème du projet.

3.1 Documentation

Pour comprendre le cahier des charges du projet, il est nécessaire de lire (et relire) attentivement les documentations suivantes :

Spécification du format ELF (*Executable and Linkable Format (ELF) version 1.1*), qui constitue la documentation principale qui vous permettra de réaliser ce qui vous est demandé. Ce document donne la spécification générique du format ELF. S'y trouvent entre autres les informations suivantes :
- sections 1 et 2 : structure du format ELF ;
- section 3 : format des sections (type, flags, etc.) ;
- sections 4 et 5 : sections spécifiques—table des chaines de caractères et table des symboles ;
- section 6 : présentation des relocations (à recouper avec les informations spécifiques de l'ARM, cf. document suivant).

Compléments de spécifications ELF relatifs à l'architecture ARM (*ELF for the ARM architecture*), en particulier la partie relative aux types de réimplantations spécifiques au processeur ARM. Ce second document décrit les spécificités du format ELF ARM. La grande majorité des informations intéressantes se trouve dans la section 4 de ce document. Vous y trouverez en particulier (entre autres) :
- spécificités des symboles du format ELF ARM (4.5) ;
- types de relocations spécifiques à l'ARM (4.6.1.2).

Attention : dans cette documentation, toutes les spécificités sont détaillées y compris celle du Thumb (un jeu d'instructions particulier). Seul le jeu d'instructions ARM (ou ARM32) fait partie de ce sujet. Le Thumb (et autres variantes) ne fait donc pas partie du projet et vous pouvez donc ignorer toutes les parties de ce document qui le détaillent (exemples : Thumb16 relocations, Thumb32 relocations).

Par ailleurs, la manuel de référence du jeu d'instructions ARM, même s'il n'a pas un rôle central dans ce projet, pourra également être utile.

3.2 Simulateur ARM

L'exécutable d'un simulateur prenant en charge un sous ensemble du jeu d'instructions ARMv5 vous est fourni. Ce simulateur est différent de l'outil `arm-eabi-run` auquel vous êtes habitués. En particulier, il dispose d'une interface permettant à un autre programme de le piloter et il permet de tracer precisément les opérations effectuées sur les registres et la mémoire ainsi que l'état du processeur après chaque instruction. L'interface de pilotage du coté simulateur est accessible en effectuant une connexion réseau (TCP) au processus exécutant la simulation et en communiquant à l'aide du protocole `gdb` pour le débogage à distance. Il est possible de choisir le port sur lequel le simulateur attend une connexion via l'option `--gdb-port`. Les différentes fonctionnalités de trace sont activée via les options dont le nom débute par `--trace`. Toutes les options du simulateur sont décrites brievement à l'écran lorsque celui-ci est exécuté avec l'option `--help`.

3.3 Code

Afin de laisser une grande liberté dans les choix d'implémentation du projet, le volume de code fourni pour démarrer le projet est volontairement très limité. En particulier, aucune base de code n'est fournie pour la gestion du format ELF. Ce qui vous est fourni est constitué du code source d'un programme permettant de

gérer les options fournies en ligne de commande , d'exécuter une suite d'instructions ARM via le simulateur et d'une infrastructure de compilation fondée sur les outils `automake/autoconf`. Cette section décrit les modules de code mis à disposition pour faciliter l'implémentation de certains aspects. Il est à noter que ces aspects ne sont en rien spécifiques au cahier des charges du projet ; ces exemples pourront donc s'avérer utiles dans d'autres contextes.

3.3.1 Gestion d'arguments en ligne de commande

Le programme principal du code source fourni, contenu dans le fichier `ARM_runner_example.c`, reconnait un certain nombre d'options sur la ligne de commande. Ces options sont décrite par un petit texte affiché lors de l'exécution de `ARM_runner_example --help`. Dans le programme principal elles sont gérées à l'aide de la fonction `getopt_long` disponible sur la plupart des systèmes UNIX. En complément de la page de manuel de cette fonction (cf. `man 3 getopt_long`), vous pourrez vous inspirer de cet exemple pour gérer les options de vos propres programmes.

3.3.2 Fonctions de débogage

Les fichiers `debug.h` et `debug.c` fournissent un ensemble de fonctions permettant de faciliter le débogage. L'intérêt de cette bibliothèque est qu'elle permet d'activer sélectivement les traces au niveau de chaque fichier source. Pour activer les traces au sein d'un fichier source donné, il faut invoquer la procédure `add_debug_to` en lui passant en paramètre le nom du fichier concerné. Ceci est typiquement fait dans la fonction `main` d'un programme, en analysant les paramètres passés en ligne de commande comme dans le fichier `ARM_runner_example.c`.

Au sein des fichiers source susceptibles d'être débogués, plusieurs fonctions de traçage peuvent être utilisées pour générer des messages (qui s'affichent par défaut sur `stderr`) :

- `debug_raw` : s'utilise comme `printf` et se comporte de la même manière. La différence est que cette fonction ne fait rien lorsque le débogage est désactivé pour le fichier dans lequel elle se trouve.
- `debug` : similaire à `debug_raw`, mais préfixe chaque message par le nom de fichier et le numéro de ligne correspondants.
- `debug_raw_binary` : permet d'afficher une séquence d'octets de taille variable — pour chaque octet, le caractère ASCII correspondant est utilisé ou, à défaut (caractère non affichable) le caractère point est utilisé.

3.3.3 Interface avec le simulateur

Les fichiers `arm_simulator_interface.h` et `arm_simulator_interface.c` contiennent respectivement les prototypes et l'implémentation de fonctions permettant de piloter le simulateur depuis un programme C. Le fichier `ARM_runner_example.c` contient un exemple d'utilisation de ces fonctions. Ces fonctions sont :

- `arm_connect` : qui étant donné le nom de machine sur lequel le simulateur s'exécute et le numéro de port sur lequel il attend une connection (`gdb`), effectue la connexion ou affiche à l'écran un message d'erreur.
- `arm_write_memory` et `arm_write_register` : qui permettent d'écrire respectivement dans la mémoire et les registres d'un simulateur connecté.
- `arm_step` et `arm_run` : qui permettent d'exécuter respectivement une instruction ou toutes les instructions jusqu'à l'arrêt du simulateur.

L'arrêt du simulateur est provoqué par l'exécution de l'instruction `swi 0x123456` et provoque la déconnexion de l'interface. Cette interface ne sera utile pour effectuer des tests qu'en fin de projet, après avoir réalisé la gestion des réimplantations.

3.3.4 Infrastructure de compilation

La compilation du code fourni se fait à l'aide d'un `Makefile` généré automatiquement lors de la configuration du projet. Les instructions de configuration et de compilation sont décrites dans le fichier `INSTALL`

et, pour un démarrage rapide, peut se résumer à l'exécution des deux commandes :

```
./configure
make
```

Cette infrastructure de compilation a été générée à l'aide des outils `automake/autoconf` et est extensible de manière relativement simple : pour ajouter un programme, il faut éditer le fichier `Makefile.am`, y ajouter un nom de programme dans la liste affectée à `bin_PROGRAMS` et y ajouter une ligne `nom_SOURCES` contenant la liste des fichiers source du programme. Attention, si vous utilisez le code fourni sur une machine ayant une version d'`automake/autoconf` trop vieille, la compilation peut produire des avertissements concernant ces outils. Pour les supprimer :

```
make distclean
autoreconf
```

En cas de problème persistant, contactez les enseignants responsables du projet.

3.3.5 Exemples fournis

Quelques programmes en langage d'assemblage ARM sont fournis dans le répertoire `Examples_loader`. Le fichier `Makefile.am` de ce répertoire contient les bonnes options de compilation pour :

- générer les `.o` correspondants, qui seront les fichiers que vous aurez à implanter ;
- réaliser l'implantation avec `arm-eabi-ld` sans lier l'exécutable avec la bibliothèque standard, afin de vous permettre de comparer le résultat de votre implantation avec celui de `arm-eabi-ld` ;

La compilation est faite en boutisme *big endian* qui correspond au boutisme du simulateur. La compilation se fait avec l'option `-mno-thumb-interwork` qui évite que gcc ne génère un code particulier permettant l'interaction avec le mode *thumb* du processeur ARM. Sans cette option, certaines des instructions du code généré peuvent être légèrement différentes de celles dont vous avez l'habitude et le compilateur produit certaines réimplantations non envisagées dans ce projet.

4 Travail demandé

Le projet est organisé en une séquence d'étapes. Il est nécessaire, pour la plupart de ces étapes, d'avoir complètement terminé une étape i avant de passer à l'étape $i+1$. Une étape est considérée comme terminée seulement après une phase de tests approfondie.

La séquence d'étapes est résumée dans la liste ci-dessous. Chaque étape est ensuite décrite de façon plus précise. Les étapes 1 à 5 (première phase) portent sur la lecture du format ELF. Les étapes 6 à 11 (seconde phase) concernent le travail de réimplantation . Dans ce projet, nous nous limiterons au format ELF 32 bits.

1. Affichage de l'en-tête ;
2. Affichage de la table des sections et des détails relatifs à chaque section ;
3. Affichage du contenu d'une section ;
4. Affichage de la table des symboles et des détails relatifs à chaque symbole ;
5. Affichage des tables de réimplantation et des détails relatifs à chaque entrée ;
6. Renumérotation des sections ;
7. Correction des symboles ;
8. Réimplantations de type `R_ARM_ABS*` ;
9. Réimplantation de type `R_ARM_JUMP24` et `R_ARM_CALL` ;
10. Interfaçage avec le simulateur ARM ;
11. Exécution à l'aide du simulateur `arm-eabi-run`.

4.1 Phase 1

Dans cette phase, il vous est demandé d'écrire un ensemble de programmes permettant de lire le contenu d'un fichier au format ELF. La séquence d'étapes est présentée par ordre de difficulté croissante et certaines d'entre elles dépendent de la précédente. Pour tous les programmes de cette phase, le résultat à produire est un affichage à l'écran uniquement.

4.1.1 Étape 1 : Affichage de l'en-tête

L'objectif de cette étape est de construire un programme capable de lire et d'afficher (sur la sortie standard) les différents champs de l'en-tête d'un fichier ELF, tels que :

- la plateforme cible (architecture et système) ;
- la taille des mots (32/64 bits) ;
- le type de fichier ELF (fichier relogeable, fichier exécutable...) ;
- la spécification de la table des sections : position dans le fichier (offset), taille globale et nombre d'entrées ;
- l'index de l'entrée correspondant à la table des chaînes de noms de sections au sein de la table des sections ;
- la taille de l'en-tête ;
- etc.

Pour bien comprendre le travail demandé et vérifier les résultats produits, les affichages de l'utilitaire `readelf` (ou `arm-eabi-readelf`) peuvent être d'une grande utilité (voir notamment l'option `-h`).

4.1.2 Étape 2 : Affichage de la table des sections

L'objectif de cette étape est de construire un programme capable de lire et d'afficher la table des sections d'un fichier ELF. Pour chaque entrée de la table, il est demandé d'afficher les principales caractéristiques, telles que :

- le numéro de section (index dans la table) ;
- le nom de la section ;
- la taille de la section ;
- le type de la section (`PROGBITS`, `SYMTAB`, `STRTAB`, etc.) ;
- les principaux attributs de la section, en particulier :
  - les informations d'allocation (la section fait-elle partie de l'image mémoire du programme à exécuter ?) ;
  - les permissions (la section contient-elle du code exécutable ? des données modifiables lors de l'exécution du programme ?) ;
- la position (offset) de la section par rapport au début du fichier ;
- etc.

À cette étape, les affichages de l'utilitaire `readelf` (ou `arm-eabi-readelf`) pourront également être utiles pour les tests (voir notamment l'option `-S`).

4.1.3 Étape 3 : Affichage du contenu d'une section

L'objectif de cette étape est de construire un programme capable de lire et d'afficher le contenu de l'une des sections d'un fichier ELF. La section choisie pourra être désignée par son numéro ou par son nom. L'affichage correspond au contenu brut du fichier (sous forme héxadécimale).

À cette étape, les affichages de l'utilitaire `readelf` (ou `arm-eabi-readelf`) pourront également être utiles pour les tests (voir notamment l'option `-x`).

4.1.4 Étape 4 : Affichage de la table des symboles

L'objectif de cette étape est de construire un programme capable de lire et d'afficher la table des symboles d'un fichier ELF. Pour chaque entrée de la table, il est demandé d'afficher les principales caractéristiques, telles que :

— le nom du symbole ;
— la valeur du symbole ;
— le type du symbole (`NOTYPE`, `SECTION`, `FUNC`, etc.) ;
— la portée du symbole (`LOCAL`, `GLOBAL`, etc.) ;
— le numéro de la section concernée (index dans la table des sections) ;
— etc.

À cette étape, les affichages de l'utilitaire `readelf` (ou `arm-eabi-readelf`) pourront également être utiles pour les tests (voir notamment l'option `-s`).

4.1.5 Étape 5 : Affichage des tables de réimplantation

L'objectif de cette étape est de construire un programme capable de lire et d'afficher les tables de réimplantation d'un fichier ELF pour machine ARM. Pour chaque entrée, il est demandé d'afficher les informations suivantes :

— la cible de la réimplantation [4] ;
— le type de réimplantation à effectuer (noter que chaque architecture matérielle dispose d'une spécification distincte pour les types de réimplantation) ;
— l'index de l'entrée concernée dans la table des symboles (le symbole impliqué dans la réimplantation, par exemple, dans le cas d'un appel de fonction, le symbole impliqué est le nom de la fonction appelée).

À cette étape, les affichages de l'utilitaire `arm-eabi-readelf` pourront également être utiles pour les tests (voir notamment l'option `-r`).

4.2 Phase 2

Dans cette phase, il vous est demandé de modifier le contenu du fichier au format ELF donné afin d'effectuer l'implantation. Vos programmes de cette phase prendront en paramètre les adresses auxquelles les sections du programme (typiquement `.text` et `.data`) doivent être chargées. Durant les étapes de cette phase, le résultat de vos programmes pourra débuter par un simple affichage, mais devra évoluer vers la production d'un fichier de sortie au format ELF. En outre, dans les étapes finales il faudra :

— utiliser l'interface avec le simulateur pour exécuter le code chargé
— produire un fichier de sortie au format ELF pour comparer le résultat avec celui de `arm-eabi-ld` et l'exécuter avec `arm-eabi-run`

Dans toute cette phase, nous nous limiterons aux types de reimplantation produites par l'assembleur et le compilateur GNU pour une cible en ARM, à savoir des reimplantations codées par des tables de type `REL` (et pas `RELA`) et limitées aux types : `R_ARM_ABS32`, `R_ARM_ABS16`, `R_ARM_ABS8`, `R_ARM_JUMP24` et `R_ARM_CALL`.

4.2.1 Étape 6 : Renumérotation des sections

L'objectif de cette étape est de construire un programme capable de produire une modification de l'organisation en sections d'un fichier au format ELF. Plus précisément, lors de l'implantation, les indications contenues dans les tables de réimplantation sont utilisées pour corriger le contenu des sections associées. Une fois toutes les réimplantations résolues, les tables de réimplantation ne sont plus nécessaires et les sections les contenant disparaîssent alors du fichier de sortie. Ainsi, le fichier de sortie doit contenir toutes les sections

4. Le champ correspondant est nommé *offset* mais son contenu dépend du type de fichier ELF qui le contient. Pour un fichier translatable, il s'agit effectivement d'un offset à partir du début de la section concernée. Pour un fichier ELF exécutable ou une bibliothèque partagée, le champ correspond en fait à l'adresse mémoire de l'élément concerné.

présentes dans le fichier d'entrée, à l'exception des sections qui contiennent des tables de réimplantation. Cela implique donc des copies de données et une renumérotation des sections.

4.2.2 Étape 7 : Correction des symboles

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente, en modifiant la table de symboles (généralement unique) du fichier de sortie. Plus précisément, étant donné les adresses de chargement des différentes sections, il s'agit de donner une valeur (une adresse) absolue à chaque symbole translatable et de corriger son numéro de section de définition selon la renumérotation précédente.

4.2.3 Étape 8 : Réimplantations de type `R_ARM_ABS*`

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente, en modifiant les sections de code (instructions ou données) du fichier de sortie. Le principe consiste à appliquer les informations de translation à chaque section de code concernée ; autrement dit, il s'agit de compléter les adresses incomplètes référencées par ces sections. Afin d'échelonner les difficultés, on ne s'intéressera dans cette étape qu'à un sous-ensemble des réimplantations : celles dont le type commence par le préfixe `R_ARM_ABS` pour les tailles 32, 16 et 8. Se reporter à la documentation *ELF for the ARM Architecture* pour les détails.

4.2.4 Étape 9 : Réimplantations de type `R_ARM_JUMP24` et `R_ARM_CALL`

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente en ajoutant la prise en charge de deux autres types de réimplantation : `R_ARM_JUMP24` et `R_ARM_CALL`. Il est à noter que si vous utilisez un compilateur ARM ancien, ces deux types de réimplantation sont remplacées par le type `R_ARM_PC24` aujourd'hui obsolète. Attention, dans cette étape, la documentation ne précise pas clairement comment appliquer le résultat de la réimplantation dans la section concernée. Pour cela, reportez vous à l'encodage des instructions associées et à l'exemple fourni.

4.2.5 Étape 10 : Interfaçage avec le simulateur ARM

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente en chargeant dans le simulateur le contenu de toutes les sections allouables du fichier d'entrée, en positionnant le compteur ordinal au point d'entrée du programme et en lançant l'exécution du programme. Le programme d'entrée devra être choisi afin d'illustrer le fait que les réimplantations ont été appliquées correctement.

4.2.6 Étape 11 : Exécution avec `arm-eabi-run`

Dans cette dernière étape, vous devrez effectuer deux tâches afin de permettre l'exécution via `arm-eabi-run`. La première sera de vous documenter sur le fonctionnement de `arm-eabi-gcc` afin de lier le fichier objet contenant votre programme à la bibliothèque standard du C sans effectuer l'implantation (qui sera effectuée par votre propre programme). La seconde tâche sera de créer dans le fichier ELF de sortie produit par votre chargeur des segments de programme indiquant au simulateur les zones mémoire à allouer pour le chargement des sections du programme. Cette partie du format ELF n'est pas décrite dans la documentation fournie et vous devrez en trouver la spécification par vos propres moyens. Une fois ces deux tâches réalisées, votre programme de sortie pourra être exécuté par `arm-eabi-run`.

5 Tests

Chacune des étapes du projet devra être validée par des jeux de tests pertinents, avec une bonne couverture des différents cas de figure à considérer. Quelques fichiers tests et traces de références seront fournis par les enseignants au cours du projet mais la grande majorité des tests devront être conçus et développés par les étudiants. Ces tests devront être documentés et rendus avec le code final du projet. Un sous-ensemble

représentatif des tests devra être présenté au cours de la soutenance à la fin du projet. Une mise en œuvre automatisée de certains tests pourra être utile (gain de temps pendant le projet et au cours de la soutenance).

6 Documents à produire

En complément du code produit, chaque groupe devra fournir un ensemble de documents à la fin du projet, sous la forme de fichiers au format texte ou pdf. Ces documents ont pour objectif de donner une vue d'ensemble du projet et de la façon dont il a été mené. Leur contenu doit être aussi synthétique que possible. Liste des documents demandés :

- Bref mode d'emploi expliquant comment compiler et lancer chacun des programmes utilitaires développés dans le cadre du projet ;
- Descriptif de la structure du code développé : principales fonctions et fichiers correspondants (inutile de décrire le code fourni par les enseignants, sauf en cas de modifications importantes) ;
- Liste des fonctionnalités implémentées et manquantes ;
- Liste des éventuels bogues connus mais non résolus ;
- Liste et description des tests effectués [5] ;
- Journal décrivant la progression du travail et la répartition des tâches au sein du groupe (à remplir quotidiennement).

Le code produit devra être envoyé par mail au responsable du projet avant la soutenance sous forme d'archive compressée (`make dist` avec le squelette fourni).

7 Annexes

7.1 Détail des étapes de génération d'un fichier binaire exécutable

On considère deux fichiers `fich1a.c` et `fich1b.c`. e premier fichier dépend d'une fonction définie dans le second ainsi que de la fonction `printf` fournie par la bibliothèque C.

Étape 1 : création de fichiers en langage d'assemblage.

```
arm-eabi-gcc -mno-thumb-interwork -S fich1a.c
arm-eabi-gcc -mno-thumb-interwork -S fich1b.c
```

On obtient ainsi deux fichiers `fich1a.s` et `fich1b.s`. Cette étape n'est pas strictement nécessaire mais elle est utile pour comprendre le contenu des principales sections générées à partir d'un fichier C.

Étape 2 : création de fichiers objet.

```
arm-eabi-as -o fich1a.o fich1a.s
arm-eabi-as -o fich1b.o fich1b.s
```

On obtient ainsi deux fichiers objet translatables `fich1a.o` et `fich1b.o`.

Étape 3 : fusion de fichiers objet.

```
arm-eabi-ld -r -o prog.o fich1a.o fich1b.o
```

5. Pour chacun des tests, la description correspondante doit permettre de connaître :
— son objectif (ce qu'il cherche à vérifier) ;
— la marche à suivre pour le lancer ;
— la façon de conclure sur le résultat observé (réussite ou échec du test), si le test n'est pas automatisé.

On obtient ainsi un fichier objet translatable `prog.o`. L'option `-r` permet de demander la création d'un fichier translatable : la translation n'est pas effectuée.

Étape 4 : création d'un fichier exécutable
Pour parvenir à la création d'un fichier binaire exécutable, il reste plusieurs choses à faire :

- lier le fichier objet `prog.o` avec la bibliothèque C (ainsi qu'avec d'autres fichiers objets fournis par la chaîne de compilation, dont le code sert notamment à effectuer des initialisations avant l'appel de la fonction `main`) ;
- effectuer l'implantation du fichier binaire.

On peut réaliser ces différentes actions à l'aide de la commande suivante, qui produit le fichier exécutable `prog` :

```
arm-eabi-gcc -o prog prog.o
```

En fait, le lancement d'`arm-eabi-gcc` déclenche ici un appel à l'éditeur de lien (`arm-eabi-ld`) pour réaliser les opérations mentionnées ci-dessus [6].

Étape 5 : exécution du programme
Pour lancer l'exécution du programme sans débogueur, utiliser `arm-eabi-run` :

```
arm-eabi-run ./prog
```

6. On peut aussi appeler `arm-eabi-ld` directement mais la ligne de commande est assez complexe. Pour voir tous les paramètres à passer à `arm-eabi-ld` dans ce cas, on peut observer le résultat de la commande suivante : `arm-eabi-gcc -### -o prog prog.o` en regardant la dernière ligne affichée (la commande `collect2` est une enveloppe qui invoque `arm-eabi-ld`).