PROG5 - Projet, année 2016-2017

Réalisation d'un éditeur de liens — Phase de Fusion

L'objectif de ce projet est d'implémenter une sous partie d'un éditeur de liens. Plus précisément, le projet est centré sur la première phase, dite de *fusion*, 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 de fusion.

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. La gestion des bibliothèques, qu'elles soient statiques ou dynamiques, n'est 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 dans les objets que vous manipulerez avec votre programme de fusion sera possible, mais la fusion avec les bibliothèques contenant le code de ces fonctions devra être faite à l'aide des outils standard (cf. 7.1).

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`. L'objectif final de ce projet est de réaliser un programme exécutant cette étape de fusion à la place de `arm-eabi-ld`.

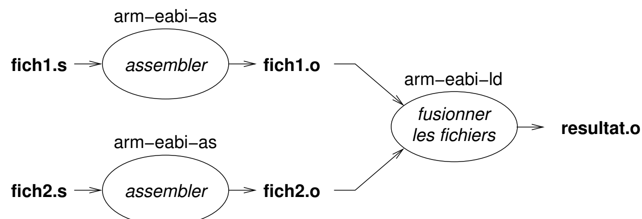

FIGURE 1 – Fusion de deux fichiers binaires translatables

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.

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 programme de fusion en examinant le résultat produit et, au besoin, en le comparant avec le résultat de `arm-eabi-ld` (figure 2).

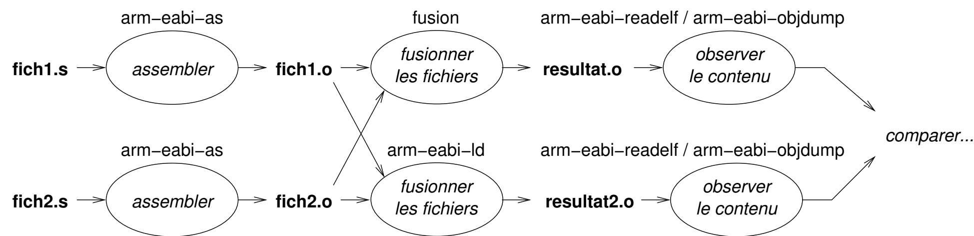

FIGURE 2 – Validation de la fusion de deux fichiers binaires translatables

Les fichiers exécutables obtenus après implantation peuvent ensuite être exécutés. Se référer aux sections suivantes 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 *fusion* de deux fichiers binaires translatables pour machine ARM. Le resultat sera un nouveau fichier binaire translatable qui pourra à son tour être fusionné à un autre fichier binaire translatable. 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. Nous nous limiterons au cas de la fusion de fichiers binaires translatables, la phase finale de fusion avec la bibliothèque standard du C, avant chargement et exécution, devra donc être réalisée à l'aide de l'éditeur de liens existant : `arm-eabi-ld`. Quant au chargement, il peut également être réalisé, comme expliqué précédemment, par `arm-eabi-ld` si les adresses de chargement sont connues (cela est le cas de certains systèmes embarqués) ou directement par le système d'exploitation avant l'exécution (c'est ce que fait `arm-eabi-run`). Un exemple de déroulement des différentes étapes impliquées dans le projet, de la compilation jusqu'à l'exécution, est donné en annexe.

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. [2] La seconde phase, et les étapes suivantes, concernent le travail de fusion proprement dit et nécessitent, pour la plupart, de comprendre les extensions du format ELF spécifiques au code binaire ARM.

2. 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 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 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 Exemples fournis

Un programme en C ayant la même structure que l'exemple de l'annexe est fourni dans le répertoire `Examples_fusion`. 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 fusionner ;
- réaliser la fusion avec `arm-eabi-ld` sans lier l'exécutable avec la bibliothèque standard, afin de vous permettre de comparer le résultat de votre fusion avec celui de `arm-eabi-ld` ;
- lier le résultat de la fusion avec la bibliothèque standard pour créer un programme exécutable avec `arm-eabi-run`.

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 9 (seconde phase) concernent le travail de fusion . Dans ce projet, nous nous limiterons au format ELF 32 bits. Nous supposerons également que nous fusionnons toujours deux fichiers binaires translatables. En effet, la fusion de plus de deux fichiers binaires translatables peut se faire de manière incrémentale : en fusionnant d'abord les deux premiers fichiers, puis en fusionnant le résultat avec les autres fichiers.

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. Fusion et renumérotation des sections ;
7. Fusion, renumérotation et correction des symboles ;
8. Fusion et correction des tables de réimplantations ;
9. Production d'un fichier résultat au format ELF.

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 [3] ;
- 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).

3. 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.

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 combiner le contenu de deux fichiers au format ELF donnés afin d'effectuer la fusion. Vos programmes de cette phase prendront en paramètres les noms des deux fichiers. Durant les étapes de cette phase, le résultat de vos programmes pourra être, au début, un simple affichage, mais devra évoluer vers la production d'un fichier de sortie au format ELF. Ce fichier résultat pourra être comparé au résultat de `arm-eabi-ld` et exécuté avec `arm-eabi-run` après avoir complété l'édition de liens via `arm-eabi-ld`.

4.2.1 Étape 6 : Fusion et renumérotation des sections

L'objectif de cette étape est de construire un programme capable de concaténer les sections de code correspondantes de deux fichiers binaires translatables et de mémoriser le possible changement de numéro des sections du second fichier ainsi que l'*offset* auquel la concaténation a eu lieu.

Plus précisément, lors de la fusion, les sections de type `PROGBITS` ayant le même nom dans les deux fichiers donnés en entrée doivent être fusionnées en une seule dans le fichier résultat. En outre, le fichier résultat devra également contenir les sections de type `PROGBITS` n'apparaissant que dans un seul des fichiers donnés en entrée. Ceci implique que les numéros des sections du fichier résultat peuvent être différents de ceux des sections des fichiers donnés en entrée. Comme les sections du fichier résultat forment un surensemble des sections des fichiers donnés en entrée, nous pouvons choisir de numéroter ses sections de manière compatible avec celle du premier fichier donné en entrée. Dans ce cas seuls les numéros des sections du second fichier donné en entrée changeront.

La fusion de deux sections se fait par simple concaténation. Pour simplifier les choses, il est possible de toujours concaténer les sections du second fichier donné en entrée à la suite des sections du premier fichier donné en entrée. Dans ce cas, seules les valeurs des symboles du second fichier donné en entrée changeront potentiellement, car ces valeurs sont des *offsets* relatifs au début de la section contenant le symbole. Il faut donc mémoriser les offsets de concaténation des sections du second fichier donné en entrée afin de pouvoir corriger les valeurs de ses symboles (cf. étape suivante).

4.2.2 Étape 7 : Fusion, renumérotation et correction des symboles

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente, afin de construire la table de symboles (généralement unique) du fichier de sortie. Plus précisément, étant donné les tables de symboles des deux fichiers donnés en entrée, il s'agit de produire par fusion une unique table respectant les contraintes suivantes :

- tous les symboles locaux des tables des symboles en entrée devront se retrouver dans la table en sortie
- les symboles globaux des tables de symboles en entrée devront être fusionnés de la manière suivante :
  - si deux symboles de même nom sont définis dans les tables en entrée, l'édition de liens échoue de manière définitive
  - si un symbole défini dans l'une des deux tables apparaît comme non défini dans l'autre, seule la définition devra apparaître dans la table de sortie
  - si deux symboles apparaissent comme non définis dans les deux tables en entrée, une seule entrée pour ce symbole devra être présente dans la table de sortie
  - si un symbole n'apparaît que dans l'une des deux tables en entrée, il devra apparaître dans la table de sortie

Bien entendu, la sortie étant constituée d'une unique table contenant un sous ensemble de l'union des deux tables en entrée, le numéro d'un symbole donné de la table de sortie ne sera pas forcément le même que le(s) numéro(s) du(des) symoble(s) correspondant(s) des tables en entrée. Il faudra conserver cette renumérotation afin de pouvoir réaliser les étapes suivantes. En outre, la valeur des symboles de la table de sortie devra être

la valeur du symbole correspondant dans les tables en entrée corrigée à l'aide de l'offset de concaténation de sa section de definition calculé à l'étape 4.2.1.

Enfin, il ne faut pas oublier que dans le fichier de sortie, une seule table de chaînes correspondra à la table de symboles générée. Le plus simple est de contruire cette table au fur et à mesure de l'ajout de symboles dans la table fusionnée, cela facilite la détermination de l'offset correspondant à chaque nom de symbole.

4.2.3 Étape 8 : Fusion et correction des tables de réimplantations

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente, en fusionnant les tables de réimplantation des fichiers en entrée et en corrigeant leurs entrées. La fusion des tables de réimplantation peut se faire par simple concaténation des tables de même nom des fichiers d'entrée. Ensuite, pour chaque entrée de la table résultante, il faut corriger :

- l'offset de la réimplantation, à l'aide de l'offset de concaténation de sa section d'origine calculé à l'étape 4.2.1.
- le numéro de symbole, à l'aide de la renumérotation calculée à l'étape 4.2.2.
- dans le cas d'un symbole de type `SECTION`, l'*addend* présent directement dans le contenu de la section associée à la table de réimplantation, en y ajoutant l'offset de concaténation de la section de definition du symbole :
  - tel quel dans le cas d'une réimplantation de type `R_ARM_ABS*`.
  - divisé par 4 dans le cas d'une réimplantation de type `R_ARM_JUMP24` et `R_ARM_CALL`.

Remarque : il est à noter que si vous utilisez un compilateur ARM ancien, les deux derniers types de réimplantation sont remplacées par le type `R_ARM_PC24` aujourd'hui obsolète.

4.2.4 Étape 9 : Production d'un fichier résultat au format ELF

L'objectif de cette étape est d'étendre le programme développé pour l'étape précédente en produisant un fichier objet relogeable au format ELF en sortie. La principale difficulté ici est de construire les tables du fichier de sortie : la table des sections (qui est déduite de la fusion des sections réalisée à l'étape 4.2.1) et la table de chaînes associée (construite de manière similaire à celle associée aux symboles). La seconde difficulté est l'écriture de l'ensemble des tables et du contenu des sections dans le fichier de sortie en respectant le format ELF. La vérification du résultat de cette étape, outre l'examen du fichier résultat, se fera en terminant l'édition de liens et en exécutant le résultat à l'aide des outils ARM.

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 [4] ;
— Journal décrivant la progression du travail et la répartition des tâches au sein du groupe (à remplir quotidiennement).

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
```

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
```

4. 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.

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 [5].

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

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

5. 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`).