Projet d'Informatique

PHELMA 2e année

Année universitaire 2015–2016

Émulateur ARM

Francois.Portet@imag.fr

Résumé

Au cours de votre expérience en informatique, il vous est sûrement arrivé d'utiliser un débogueur pour trouver une erreur bien cachée au fond de votre programme. Cet outil bien pratique permet d'*émuler* l'exécution d'un programme et d'explorer ou modifier dynamiquement sa mémoire. C'est un outil tellement indispensable pour le développement et l'analyse de logiciels, qu'on en trouve pour tous les langages informatiques.

Ce projet a pour but de réaliser un débogueur de programmes écrits en langage assembleur sous certaines contraintes usuellement rencontrées dans un projet professionnel. Ce programme, que l'on appelle *émulateur*, sera écrit en langage C. Durant sa réalisation vous vous familiariserez avec le microprocesseur *ARM Cortex-M* et le langage assembleur qui lui est associé. Vous aborderez les notions d'émulateur, d'analyse lexicale, de gestion des erreurs, de fichiers objets et bien d'autres qui vous permettront d'enrichir considérablement votre culture générale et votre savoir-faire en informatique.

Table des matières

Chapitre 1

Introduction

L'objectif de ce projet informatique est de réaliser, en langage C, un émulateur de microprocesseur *ARM Cortex-M* permettant d'exécuter et de mettre au point des programmes écrits dans le langage binaire cible. Le rôle d'un tel émulateur est de lire un programme donné en entrée et d'exécuter chacune des instructions avec le comportement attendu de la machine cible. Les émulateurs permettent notamment de prototyper et déboguer des programmes sans la contrainte de posséder le matériel cible (en l'occurrence, au cas où vous ne l'auriez pas compris, une machine avec un microprocesseur *ARM Cortex-M*).

Plus précisément, l'émulateur que vous devrez réaliser prendra en entrée un fichier objet binaire au format *E*LF et permettra :

— de le charger en mémoire (virtuelle) ;
— de l'exécuter, entièrement ou pas à pas ;
— de modifier et/ou afficher son code assembleur ou la mémoire qu'il utilise pour le mettre au point ;

Votre émulateur sera évalué dans l'environnement GNU/Linux de l'école [1]. Pour ce projet, nous considérerons en fait un microprocesseur simplifié, n'acceptant qu'un jeu réduit des instructions de l'architecture complète *ARM Cortex-M*. L'exécution de ces instructions sera émulée par des appels de fonctions du langage C dans la machine dite *hôte* [2].

Vous vous référerez à la table des matières pour une organisation générale du sujet.

L'intérêt opérationnel de ce projet est multiple : il permet tout d'abord de travailler sur un projet de taille raisonnable sous tous ses aspects techniques (analyse d'un problème, conception puis implantation d'une solution, et enfin validation du résultat), mais il permet aussi d'aborder la notion de gestion de projet (découpage du travail et respect d'un *planning*). Ce projet vous permettra également d'acquérir une certaine maîtrise du langage C, tel qu'effectivement utilisé ailleurs [3], tant pour la programmation scientifique que pour le développement industriel. Nous en profiterons pour vous présenter quelques notions importantes (*e.g.* machine à état) qui vous manqueraient encore. Vous aurez aussi à intégrer l'utilisation d'outils de développement propres à automatiser les tâches élémentaires de développement. Enfin, ce projet illustre et met en pratique des connaissances relatives aux microprocesseurs ou aux systèmes (*cf.* cours d'architecture, d'ordinateurs et microprocesseurs ou de systèmes d'exploitation),

1. Toute excuse du genre *"Mais je ne comprends pas, ça fonctionnait pourtant chez moi"* ne saurait être prise en compte pour l'évaluation de votre travail. La note minimale, à savoir zéro, vous serait alors attribuée sans autre forme de procès.

2. La machine dite *hôte*, malgré la glorieuse ambiguïté de la langue française au sujet de ce mot, est réputée être celle sur laquelle le code de la machine *ARM Cortex-M* est émulé.

3. Il n'est donc pas impossible que vous ressentiez comme une légère accélération par rapport à ce que vous avez vécu en première année...

Chapitre 2

Description du ARM Cortex-M3

Le *ARM Cortex-M* est un microprocesseur de la famille ARM qui est principalement composée de microprocesseurs RISC 32 bits (ARMv1 à ARMv7) mais également 64 bits (ARMv8). RISC signifie que le microprocesseur possède un jeu d'instructions réduit (*Reduced Instruction Set Computer*) mais qu'en contrepartie, il est capable de terminer l'exécution d'une instruction à chaque cycle d'horloge. Les processeurs ARM sont devenus dominants sur le marché de l'informatique embarquée au point de les retrouver dans la plupart des tablettes et téléphones mobiles. Ce succès s'explique par les performances de ces microprocesseurs mais également par leur technique de commercialisation qui est assez originale dans le domaine. En effet, ARM [1] Ltd. conçoit les architectures mais ne fabrique pas les puces. Elle vend des licences aux constructeurs. Ceci explique que la plupart des grands fabricants proposent des puces d'architecture ARM : Qualcomm, Nvidia, STMicroelectronics, Texas Instruments, Freescale, Broadcom, etc.

Dans ce projet, nous nous intéresserons à un des microprocesseurs les plus récents, le Cortex-M pour l'embarqué et notamment à l'architecture ARMv7-M qui est celle mise en œuvre dans les microprocesseurs de type Cortex-M3. Les aspects importants de cette architecture pour le projet seront succinctement décrits dans ce chapitre après des rappels généraux sur l'exécution d'un programme.

2.1 Architecture du ARM Cortex-M3

Les RISC sont basés sur un modèle en pipeline pour exécuter les instructions. Cette structure permet d'exécuter chaque instruction en plusieurs cycles, mais de terminer l'exécution d'une instruction à chaque cycle. Dans le cas du *ARM Cortex-M*3, cette structure en pipeline est illustrée sur la Figure 2.1. L'extraction (*Instruction Fetch - Fe*) va récupérer en mémoire l'instruction à exécuter. Le décodage (*Instruction Decode - De*) interprète l'instruction, résout les adresses des registres et les lit. C'est aussi à cette étape que certains branchements sont calculés. L'exécution (*Execute- Ex*) utilise l'unité arithmétique et logique pour exécuter l'opération, accéder à la mémoire et mettre à jour la valeur de certains registres avec le résultat de l'opération.

2.2 Exécution d'un programme sur un microprocesseur

Pour la plupart d'entre vous, la transformation du code écrit en texte (programme C) vers un fichier exécutable (suite à une compilation et édition de liens) reste encore une partie très obscure de votre cursus informatique. Pour l'instant, il suffit de rappeler que tout programme saisi dans un langage textuel doit être traduit dans le langage machine de l'ordinateur sur lequel il doit être exécuté (c.-à-d., la machine *hôte*) [2]. Dans notre cas, il s'agit du code du processeur que vous verrons plus en détails dans la section 2.6 [3]. Cependant, pour que ce programme s'exécute sur le processeur, il faut qu'il soit placé en mémoire (on dit *chargé* en mémoire) afin que celui-ci puisse être 'lu' par le

1. ARM signifie *Advanced Risc Machine*, acronyme qui signifiait à l'origine *Acorn Risc Machine*.
2. Vous aurez l'opportunité, dans ce projet, d'aller ouvrir des fichiers dans le langage machine, ce qui sera, n'en doutons pas, une expérience inoubliable voire même une révélation...
3. Bien entendu un programme compilé sur/pour un système d'exploitation et un processeur particuliers n'est pas exécutable sur un OS ou processeur différent.

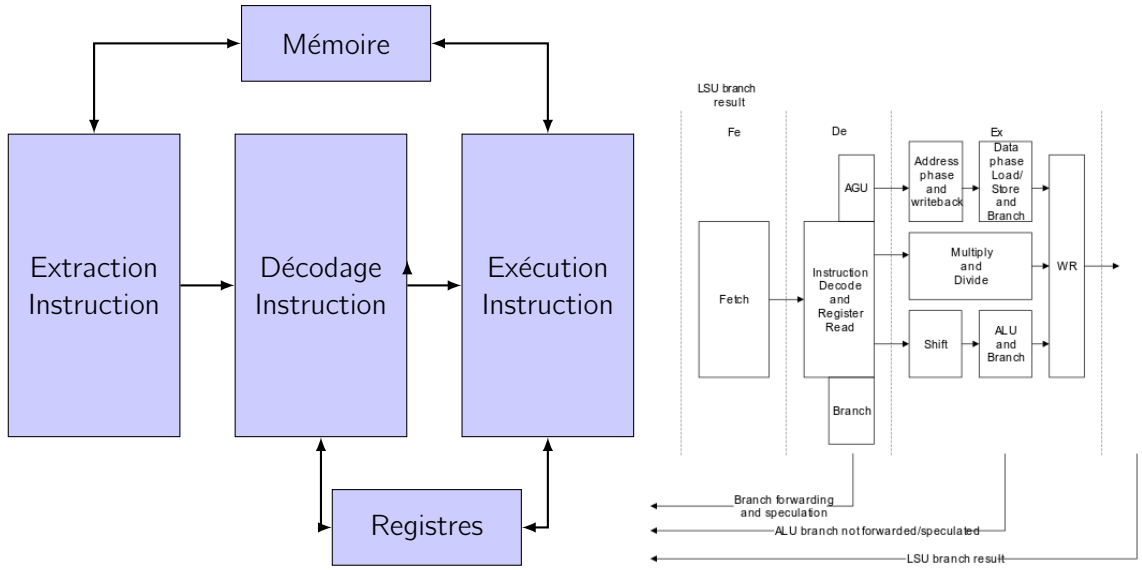


Figure 2.1 – Architecture interne simplifiée et plus détaillée du pipeline à trois étages du *ARM Cortex-M3*

processeur [4].

Le Schéma de la Figure 2.2 décrit les grandes étapes de l'exécution d'un programme.

1. Lorsque le programme est 'lancé', le fichier exécutable est 'chargé' en mémoire vive par le système d'exploitation [5]. Pendant ce chargement, l'OS, trouve un espace mémoire suffisant pour stocker le programme et ses donnés auxquelles il réservera l'accès exclusif au programme qui s'exécute (c.-à-d., les programmes s'exécutant sur l'OS en même temps n'y auront pas accès).
2. Le programme est exécuté proprement dit. C'est à dire que chacune des instructions est lue en mémoire et exécutée par la CPU (Central Process Unit) séquentiellement. Pour des raisons de rapidité et modularité, les opérations (arithmétique, logique, accès mémoire) sont effectuées sur des registres qui sont des zones mémoires à accès ultra-rapide contenues dans la CPU du microprocesseur. Autrement dit, les opérations ne sont jamais directement effectuées sur la mémoire vive mais toujours à travers des registres [6]. Ces registres sont mis à jour à travers des instructions explicites lisant ou écrivant de/vers la mémoire de/vers les registres. Par ailleurs, pour connaître l'emplacement de la prochaine instruction à exécuter, le microprocesseur utilise un pointeur d'instruction (Programme Counter) qui est un registre mis automatiquement à l'adresse mémoire de la prochaine instruction à exécuter. Enfin, le programme peut également avoir à interagir avec le monde extérieur à travers des 'appels systèmes' (interactions avec l'OS). Le simple fait de vouloir afficher des caractères à l'écran nécessite de faire appel à l'OS qui gère les périphériques d'E/S tels que l'écran ou le clavier [7].
3. Une fois le programme terminé (arrivé en fin d'exécution, interrompu par l'OS, etc) l'OS libère

4. Pour un nombre conséquent de bonnes raisons que vous découvrirez en cours d'OS, le processeur ne lit pas le programme à partir du disque dur

5. Vous noterez que ce système d'exploitation est lui-même exécuté par le microprocesseur, mais nous ne discuterons pas cet aspect que vous aurez le plaisir de découvrir en cours d'OS

6. C'est la fameuse *load/store architecture* que l'on retrouve dans quasiment tous les processeurs RISC.

7. Vous conviendrez aisément qu'exécuter le programme *Mario Kart* sans écran ni manette devient vite frustrant...Certains programmes font donc abondamment usage des appels systèmes.

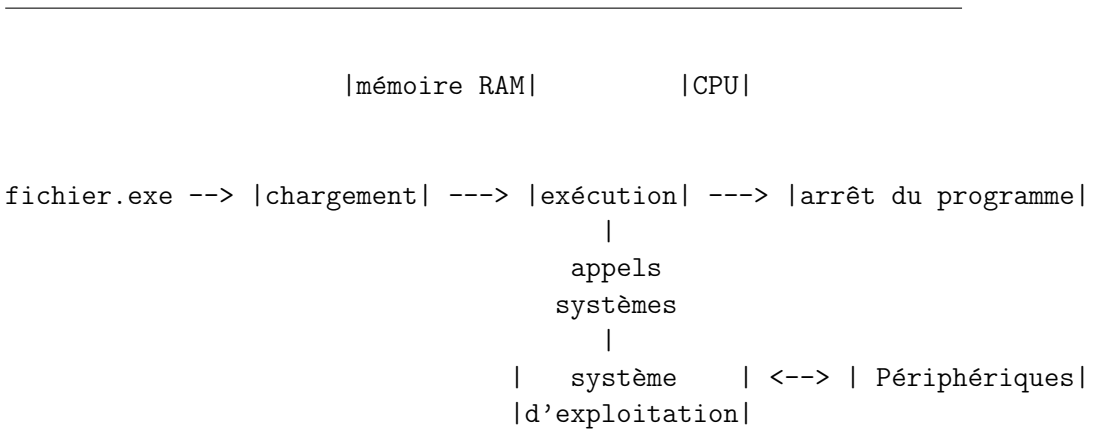


Figure 2.2 – Vue simplifiée de l'exécution d'un programme sur un microprocesseur

la mémoire vive et toutes autres ressources allouées au programme.

Tous ces éléments (code binaire, mémoire, registres) impliqués dans l'exécution d'un programme sont l'objet du reste de ce chapitre. Ce sont, entre autre, ces éléments qu'il va falloir représenter dans le projet.

2.3 Format des fichiers binaires utilisés dans ce projet (ELF)

Il ne vous aura pas échappé qu'une image n'est pas un son. De la même manière, un fichier MP3 n'est pas non plus un fichier exécutable. Il faut donc qu'il existe des conventions pour décrire des sons, des images, des fichiers exécutables, ou quoi que ce soit d'autre : des vidéos, des objets 3D, des fichiers de paramètres. . .Vous conviendrez aisément qu'il n'aurait aucun sens à ce que des sons soient décrits de la même manière qu'autre chose que des sons. Il faut donc que chaque type de fichier ait ses propres conventions pour décrire son contenu. Nous nous intéressons donc ici aux fichiers décrivant des programmes exécutables.

Nous voulons être capables d'émuler le comportement d'une machine *ARM Cortex-M* sur une autre machine[8]. Il faut donc trouver une manière de décrire *ce* qui doit être exécuté et *à* quel endroit. De la même manière qu'il existe un format[9] pour décrire les fréquences d'un son, il doit donc en exister un autre pour décrire ce qu'un programme doit exécuter. Un *format* décrit les modalités particulières répondant à la fonction de ce qui ne serait sinon rien d'autre qu'une bête suite d'octets stockés sur un disque. *Cf.* les notions de codeur/décodeur en communications numériques.

Les formats de représentation des programmes informatiques sont multiples. Ils dépendent principalement des systèmes sur lesquels le code machine qu'ils contiennent est censé être exécuté. Dans le monde *Windows*, on parle généralement aujourd'hui de format PE (*Portable Executable*), alors que dans le monde Unix des premiers temps on utilisait le format COFF (*Common Object File Format*[10]). Et aujourd'hui, le monde Unix utilise le format ELF (*Executable and Linkable Format*).

Dans ce projet, nous utiliserons le format ELF pour représenter le code *ARM Cortex-M* devant être émulé. Autant que faire se peut, nous voudrons vous masquer les détails de cette représentation particulière du code exécutable. Par contre, il nous faut vous expliciter la versatilité de ce format. En effet, le format ELF permet de décrire :

8. Ici, une machine de type amd64, mais peu importe.
9. *i.e* Un ensemble de conventions.
10. Le format PE de Windows est une extension du format COFF.

— du code partagé par plusieurs programmes (des *bibliothèques*[11] dites partagées[12]) ;
— du code dit *objet* : la plus simple expression d'un code machine, auquel il manque beaucoup de choses pour s'exécuter sur un système natif ;
— du code dit *exécutable* : du code pouvant être exécuté immédiatement dans un système.

Dans ce projet, nous allons vouloir exécuter du code *objet ARM Cortex-M* dans un émulateur. Se pose alors la question de savoir comment une vraie machine voit l'exécution d'un code machine, et surtout comment lui faire croire qu'il s'agit d'un authentique code *exécutable* alors qu'il ne s'agit en réalité que d'un code *objet*[13].

2.4 Mémoire et map mémoire

Comme dit précédemment, lorsqu'un fichier exécutable est lancé, il est tout d'abord chargé en mémoire. Dans le cas du microprocesseur qui nous intéresse, celui-ci possède un bus de 32 bits. Cette mémoire est donc au maximum de 4 Go (2^{32} bits) adressable par octets. C'est dans cette mémoire qu'on charge la suite des instructions du microprocesseur contenues dans un programme binaire exécutable. Pour exécuter un tel programme, le microprocesseur vient chercher séquentiellement les instructions dans cette mémoire, en se repérant grâce à un compteur programme (PC) contenant l'adresse en mémoire de la prochaine instruction à exécuter. Les données nécessaires à l'exécution d'un programme y sont également placées.

Même si en pratique la mémoire physique utilisée par les RISC 32 est bien inférieure à 4Go, pour des raisons qui vous deviendront limpides en cours de Systèmes d'exploitation, il est bon qu'un programme croie qu'il dispose de toute la mémoire de la machine sur laquelle il s'exécute[14]. Quoi qu'il en soit, nous considérerons qu'un programme *voit* effectivement 4Go pour lui seul. Il est donc libre d'organiser la vue de sa mémoire comme bon lui semble. Dans le cas qui nous concernent, c'est-à-dire l'architecture ARMv7-M, la mémoire virtuelle n'est pas supportée. Cependant, vous aurez quand même le bonheur de découvrir l'adressage virtuel grâce à l'émulateur...

La vue qu'a un programme en train de s'exécuter (*i.e.* en mémoire) vis-à-vis de lui-même est appelée son *map* mémoire[15]. Ce *map* mémoire doit nécessairement contenir la zone dans laquelle sont contenues les instructions à exécuter, ainsi que les données sur lesquelles agir. C'est même en réalité les seules zones utiles que contient la description d'un fichier *objet*. Ces zones seront par la suite appelées *section* lorsqu'elles feront référence à leur emplacement dans le fichier *objet*, et *segment* lorsqu'elle référenceront un espace en *mémoire*. Le *segment* d'instructions exécutables d'un programme est appelé `.text`, et son segment de données est appelé `.data`.

Le paysage est loin d'être complètement décrit. En effet, pour reprendre un exemple décrit précédemment, lorsque'il vous prend la fantaisie d'appeler `sin` pour un calcul vous n'écrivez pas le

11. Appelées DLL pour *Dynamic Link Libraries* sous *Windows*, mais aussi utilisées pour d'autres buts d'ailleurs.
12. Lorsque deux programmes veulent calculer un cosinus, il paraît légitime de ne pas dupliquer en mémoire le code pour effectuer ce calcul, d'où l'intérêt d'une bibliothèque de mathématiques *partagée*.
13. La complexité des paramètres à prendre en compte pour l'exécution d'un authentique code *exécutable* est telle que vous nous auriez jeté des pierres pour nous punir d'avoir eu cette ambition pour vous. De plus, et c'est aussi encore là que ce projet devient intéressant, nous nous proposons de vous faire toucher du doigt comment ces choses fines s'articulent en pratique. Cette démarche constructive ne saurait trouver d'équivalent intellectuel, et le voyage commence donc avec des fichiers dits *objet*.
14. C'est même pire que ça : même si physiquement une machine dotée d'une largeur de bus d'adresse de 32 bits ne dispose que de 2 Go de mémoire physique (des barrettes de RAM), *chaque* programme croira qu'il dispose de 2^{32}octets=4Go de mémoire pour lui seul.
15. En français : sa *carte* mémoire. Cette même expression est aussi utilisée pour décrire l'ensemble des adresses auxquelles une *machine* a *physiquement* accès – mais notre contexte ne permet pas l'ambiguïté puisque nous parlons de la vue *virtuelle* qu'un *programme* a de lui-même.

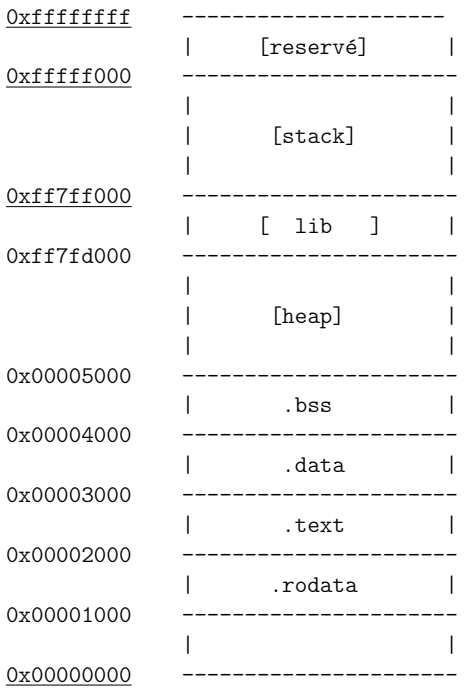


Figure 2.3 – Un *map* mémoire typique de 4Go tel que considéré dans ce projet. Les adresses soulignées sont invariables. La suite du sujet explique comment fixer les autres. Notez qu'il est possible que certains segments (typiquement `.rodata`, `.bss`, `.data`) n'existent pas dans certains programmes.

code de `sin`. Vous vous contentez d'inclure `math.h` pour que tout se passe bien. Mais le code de `sin` est ailleurs. Nous ferons l'hypothèse que ce code se trouve dans une zone d'instructions machine appelée `[lib]`. Ce *segment* en mémoire sera réputé contenir la plupart des fonctions usuelles que vous aviez l'habitude d'utiliser.

Continuons la description des lieux. Si par exemple, votre code contient un appel à une fonction d'affichage (p.ex., `printf`), fonction située quelque-part dans la zone `[lib]`, il se trouve qu'elle ne fait en réalité que faire appel à une autre fonction permettant d'*écrire* sur un dispositif *matériel* (en l'occurrence : l'écran). En tant que simple *utilisateur* de la machine, vous n'avez en réalité pas le droit de pratiquer des opérations directement sur le *matériel*. Pour ce faire, vous êtes obligés de procéder à ce que l'on appelle un *appel système*. Car, sur une machine bien policée, seul le *système* est autorisé à accéder au matériel, pour le bien de tous et de chacun.

Afin de terminer la description du paysage vu par un programme, il nous reste à décrire deux autres zones en mémoire. La plus simple concerne les données qui ne font qu'être lues et qui ne seront jamais modifiées. Ces données résident dans un segment appelé `.rodata`[16]. Un exemple de telles données pourrait être la chaîne de format `"Projet Info"` que votre programme ne manquera de contenir...

Le dernier segment, et un des plus importants, est celui de la pile. La pile contient les données temporairement utiles. Il contient aussi, tout aussi temporairement, les valeurs nécessaires pour que les appels de fonction à fonction se passent correctement : comment passer les paramètres ? comment récupérer la valeur de retour d'une fonction ? Ce segment sera appelé `[stack]` dans la suite. Il sert notamment et nécessairement pour les appels récursifs de fonction.

La visite du paysage est maintenant terminée. Néanmoins, pour être certain d'être complet, nous nous devons de mentionner la zone mémoire appelée le *tas*. C'est dans cette zone que sont retournés

16. `ro` comme *read only*.

les pointeurs correspondant aux blocs mémoire alloués par la fonction `malloc` et libérés par `free`. Mais nous n'utiliserons pas le tas car sa gestion, que nous vous aurions fait supporter, se révèle à la fois trop complexe et inutile pour notre propos. Ce segment s'appellerait `[heap]`.

La mémoire du Cortex-M3 peut être vue comme un tableau d'octets, c'est à dire que l'on ne peut pas accéder directement à un bit particulier en RAM mais seulement à un ensemble de 8 bits (octet). L'adresse d'un octet en mémoire correspond alors au rang qu'il occupe dans le tableau des 4 Go qui la constitue. Pour stocker en mémoire des valeurs sur plusieurs octets, par exemple un mot sur 4 octets, deux systèmes existent (figure 2.4) :

- les systèmes de type *big endian* écrivent l'octet de poids le plus fort à l'adresse la plus basse. Les processeurs MIPS sont *big endian*, ainsi par exemple que les processeurs PowerPC, DEC ou SUN.
- les systèmes de type *little endian* écrivent l'octet de poids le plus faible à l'adresse la plus basse. Les processeurs ATHLON ou PENTIUM notamment sont *little endian*.

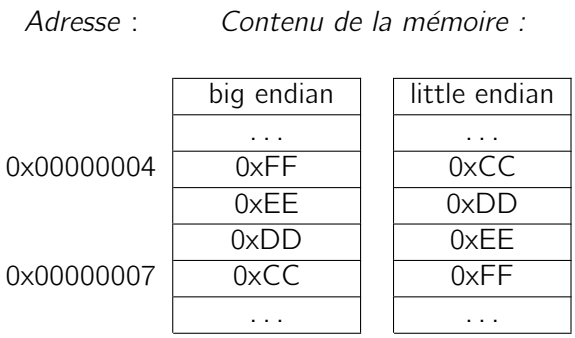


Figure 2.4 – Mode d'écriture en mémoire de la valeur hexadécimale *0xFFEEDDCC* pour un système *big endian* ou *little endian*.

L'architecture ARMv7-M est assez particulière sur ce point ; elle permet de sélectionner l'*endianess* de la zone de données. L'accès aux instructions se faisant toujours en little endian quelque de soit le modèle d'*endianess* sélectionné. La mémoire du ARMv7-M est dite *alignée*. En effet, les instructions étant soit sur 16 bits soit sur 32bits, le processeur ne lira des instructions que sur des adresses mémoires multiples de 2 octets (16 bits).

2.5 Les registres

La plupart des processeurs RISC sont de type *load/store*, c'est-à-dire que l'information en mémoire n'est accessible qu'à travers des opérations de lecture/écriture. Il est donc impossible d'effectuer le calcul d'un sinus directement sur des données mémoires. Pour cela les microprocesseurs utilise des registres (petit emplacement mémoire câblé dans la CPU) qui permet de stocker des valeur temporaires sur lesquelles s'effectue le gros des calculs. Le résultat peut ensuite être stocké en mémoire si nécessaire. Ainsi donc les registres sont des emplacements mémoire spécialisés utilisés par les instructions et se caractérisant principalement par un temps d'accès rapide. Leur importance est telle qu'ils méritent qu'on les détaille.

2.5.1 Les registres d'usage général

La machine ARM dispose de 16 registres de 32 bits chacun, dénotés `R0` à `R15`. 13 d'entre eux sont d'usage général (General Purpose Registers, *GPR*) tandis que les 3 derniers sont réservés à des opérations particulières. La figure 2.5 résume les conventions et restrictions d'usage que nous retiendrons pour ce projet.

Registre	Usage
`r0-r3`	Arguments d'une sous-routine ainsi que les résultats d'un appel
`r4-r11`	Variables locales
`r12`	Registre temporaire, préservés par les sous-routines
`r13`	*Stack Pointer* : pointeur de pile
`r14`	*Link Register* : utilisé par certaines instructions pour sauver l'adresse de retour d'un saut
`r15`	*program counter* : contient l'adresse de la prochaine instruction à executer

Figure 2.5 – Conventions d'usage des registres *ARM Cortex-M*.

Pour plus de clarté, les trois registres spéciaux sont décrits ci-dessous

SP *Stack pointer* (`R13`) : qui contient l'adresse où sera stocké le prochain élément de la pile. À l'initialisation du programme, SP pointe sur le haut de la pile.

LR *link register* (`R14`) est utilisé pour stocker une valeur de retour quand le programme entre dans une fonction à travers l'appel d'une instruction *Branch with Link*. LR est initialisé à $0xFFFFFFFF$ à l'initialisation. Notez que LR peut être utilisé comme un registre général lorsqu'il n'est pas sollicité.

PC *Program Counter* est le compteur programme 32 bits, qui contient l'adresse mémoire de la **prochaine** instruction. Il est mis à jour après l'exécution de chaque instruction. Il n'est en principe pas accessible directement en écriture.

2.5.2 Le registre d'état

Le registre d'état `APSR` (pour *Application Program Status Register*), contient des drapeaux mis à jours après l'exécution de certaines instructions. Il est notamment utilisé en cas d'erreurs (débordements, ...). La figure 2.6 décrit les indicateurs que nous utiliserons dans le projet.

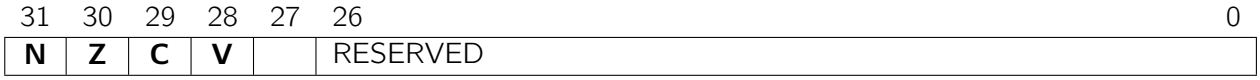


Figure 2.6 – Indicateurs du registre `APSR`

- **N** : Négatif : mis à 1 si le résultat d'une opération est un nombre négatif (en complément à 2) à 0 si le résultat est positif ou nul.
- **Z** : Zéro. Il contient 1 si le résultat d'une opération arithmétique ou logique est nul et 0 sinon. Une valeur zéro indique souvent un résultat égal pour une instruction de comparaison.
- **C** : *Carry* mis à un si l'exécution d'une instruction résulte en une retenue (par exemple, après une addition ou une soustraction non signée).
- **V** : *oVerflow*. Il indique un débordement après une opération dont les opérandes sont considérées comme des valeurs signées. Pour des opérations sur des valeurs non signées il est non modifié.

2.6 Les Instructions

Bien entendu, comme tout microprocesseur qui se respecte, l'*ARM Cortex-M* possède une large gamme d'instructions. Dans la famille des processeurs ARM, le jeu d'instruction originel purement ARM (sur 32 bits ou 64bits) a progressivement été étendu avec d'autres jeux d'instructions. L'une des originalités des instructions ARM est pouvoir être conditionnelles. C'est à dire que le code binaire de l'instruction contient les conditions de son exécution. L'ARMv7-M utilise le jeu d'instructions *Thumb-2* qui étend le jeu d'instructions *Thumb*.

Thumb est un jeu d'instructions, principalement codées sur 16bits[17], qui permet d'obtenir un code plus compact qu'en utilisant les instructions ARM classiques. Cela permet ainsi d'obtenir un chargement et une exécution beaucoup plus rapide. Cependant, seule les instructions de branchement sont conditionnelles, le nombre de registres accessibles est réduit à 8 et le nombre d'instructions devient évidement plus restreint. Par exemple, elles ne permettent que deux opérandes au maximum. Pour compenser cette restriction, *Thumb-2* ajoute des instructions 32 bits à *Thumb* afin de donner plus de possibilité de codage. *Thumb-2* est donc un jeu d'instruction de largeur variable. L'ambition étant ici de garder l'aspect compact du code *Thumb* tout en proposant un panel d'instructions similaires au ARM 32bits. L'exécution conditionnelle est généralisée à travers les *If-Then (IT) blocks* qui permettent d'exécuter jusqu'à 4 instructions liées par une même condition.

Dans ce projet nous ne prendrons en compte qu'un nombre restreint d'instructions simples. Les spécifications des instructions étudiées dans ce projet sont données dans l'annexe C. Elles sont directement issues de la documentation du ARMv7-M [3].

Nous donnons ici un exemple pour expliciter la spécification d'une instruction : l'opération compare (`CMP`), dont la spécification, telle que donnée dans le manuel, est reportée ci-dessous Figure 2.7. Cette opération consiste à soustraire à un registre une valeur immédiate pour ensuite mettre les drapeaux du registre APSR à jour. Cette opération respecte la syntaxe suivante :

```
CMP<c><q> <Rn>, #<const>
```

`<c>` représente une condition optionnelle d'exécution (voir section 2.6.1). Si cette valeur est omise, l'instruction est toujours exécutée.
`<q>` est un qualificatif optionnel qui peut prendre 2 valeurs :
`.N` (*Narrow*) qui signifie que l'assembleur doit encoder l'instruction sur 16 bits
`.W` (*Wide*) qui signifie que l'assembleur doit encoder l'instruction sur 32 bits
`<Rn>` est le registre dont la valeur va être comparée et
`# <const>` est la valeur immédiate non signée à comparer.

Cette instruction peut être codée sous deux formes : **T1** (Thumb) sur 16 bits et **T2** (Thumb2) sur 32 bits.

La syntaxe de T1 est :
T1 : `CMP<c> <Rn>,#<imm8>`

où `#<imm8>` est la valeur immédiate non signée à comparer (entre 0 et 255). Ainsi, `CMP R3,#12` ou `CMP R4,#250` sont des exemples valides correspondant à l'encodage T1.

Tandis que celle de T2 est :
T2 : `CMP<c>.W <Rn>,#<const>`
où `#<const>` est une constante sur 32 bits qui peut être représentée par une valeur sur 8 bits décalée

17. À part certaines instructions de branchement sur 32 bits

ou répliquée. Ainsi, `CMP.W R3,#0x00FF0000` ou `CMP.W R7,#0xFF00FF00` sont des exemples valides correspondant à l'encodage T2.

La version T1 permet d'encoder une comparaison lorsque la valeur immédiate est comprise entre 0 et 255 tandis que T2 est utilisée dans d'autre cas (notamment lorsque la valeur est signée). Notez qu'il est préférable d'éviter de comparer une valeur avec le registre PC car ce cas a un effet non déterminé (certains compilateurs n'assemblent pas une telle instruction).

Pour le reste, pas de commentaire : il s'agit juste un petit exercice pratique d'anglais...

CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb ISA.

`CMP<c> <Rn>,#<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn			imm8							

```
n = UInt(Rdn);  imm32 = ZeroExtend(imm8, 32);
```

Encoding T2 ARMv7-M

`CMP<c>.W <Rn>,#<const>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	1	Rn				0	imm3			1	1	1	1	imm8							

```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Assembler syntax

`CMP<c><q>  <Rn>, #<const>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A6-7.

`<Rn>` Specifies the register that contains the operand. This register is allowed to be the SP.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. The range of allowed values is 0-255 for encoding T1. See *Modified immediate constants in Thumb instructions* on page A5-15 for the range of allowed values for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

Figure 2.7 – Instruction CMP

Exemple de codage pour l'instruction `CMP` :

`cmp r4, #250`	`2CFA`
`CMP.W R3,#0x00FF0000`	`F5B30F7F`

À vous de retrouver ceci à partir de la doc ! Un bon petit exercice pour bien comprendre...

cond	Mnemonic	Meaning integer	meaning float	Condition flags
0000	EQ	Equal	Equal	$Z == 1$
0001	NE	Not equal	Not equal, or unordered	$Z == 0$
0010	CS (HS)	Carry set	Greater than, equal, or unordered	$C == 1$
0011	CC (LO)	Carry clear	Less than	$C == 0$
0100	MI	Minus, negative	Less than	$N == 1$
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	$N == 0$
0110	VS	Overflow	Unordered	$V == 1$
0111	VC	No overflow	Not unordered	$V == 0$
1000	HI	Unsigned higher	Greater than, or unordered	$C == 1$ and $Z == 0$
1001	LS	Unsigned lower or same	Less than or equal	$C == 0$ or $Z == 1$
1010	GE	Signed greater than or equal	Greater than or equal	$N == V$
1011	LT	Signed less than	Less than, or unordered	$N! = V$
1100	GT	Signed greater than	Greater than	$Z == 0$ and $N == V$
1101	LE	Signed less than or equal	Less than, equal, or unordered	$Z == 1$ or $N! = V$
1110	AL	Always (unconditional)	Always (unconditional)	Any

Table 2.1 – Liste des différents codes de conditions possibles

2.6.1 Les instructions conditionnelles

Une instruction conditionnelle a un effet sur le programme seulement si les drapeaux N, Z, C et V du registre APSR satisfont une condition spécifiée dans l'instruction. Si la condition n'est pas satisfaite, alors l'instruction agit comme un NOP (No Operation). Par exemple, dans le code suivant :

```
cmp r0,#8
BEQ fonction
```

Une comparaison `CMP` entre le registre r0 et la valeur 8 est d'abord effectuée. Ceci a pour effet de mettre à jour les drapeaux du registre APSR. Ensuite un appel à la routine `BL` *fonction* est effectué uniquement si le drapeau Z est a un, c'est le test `EQ`[18]. Le tableau 2.1, donne la liste des conditions possibles.

Ce mécanisme qui mêle instruction et condition d'exécution permet d'obtenir un code très compact. Cependant, mis à part les instructions de branchement, la plupart des instructions Thumb sont non-conditionnelles. Pour pallier ce manque, Thumb2 introduit l'instruction `IT` (*IF-THEN-else*) qui peut exécuter jusqu'à 4 instructions selon la même condition. Si le code de condition donné à l'instruction `IT` est vérifié alors l'instruction suivante est exécutée. L'intérêt de cette instruction est de pouvoir englober jusqu'à trois autres instructions dans ce que l'on appelle le *IT block*. Les conditions additionnelles de type `t` (*then*) ou `e` (*else*) sont agrégées à l'instruction pour contrôler l'exécution des instructions suivantes. Par exemple, `ite` se lit *if-then-else*, `itte` se lit *if-then-then-else* et `itee` se lit *if-then-else-else*. Le code présenté ci-dessous permet d'incrémenter r0 s'il contient une valeur inférieur à 10 ou de le réinitialiser à 0 si sa valeur est supérieure ou égale à 10.

```
cmp     r0, #10
itee    lo        @ si r0 est inférieur à 10
addlo   r0, #1    @ Alors (then) r0 = r0 + 1
movhs   r0, #0    @ Sinon (else) r0 = 0
blhs    fonction  @ Sinon on va à fonction
```

18. Le test d'égalité entre deux valeurs est effectué par soustraction. Deux valeurs sont donc égales si le résultat de leur soustraction vaut zéro, d'où la valeur Z à un.

Notez que les instructions couvertes par le `IT` doivent également répliquer la condition (ici `LO`) ou l'opposée de cette condition (ici `HS`). L'assembleur vérifiera la cohérence de ces conditions avec la condition du bloc `IT`. Toute instruction couverte par un *then* (ici `add`) doit avoir la même condition que le `IT` (ici, `lo`), alors qu'un *else* (ici, `mov` et `bl`) doit avoir la condition opposée (ici `hs`).

L'utilisation du bloc `IT` contient plusieurs cas particuliers. Tout d'abord la condition `al` n'attend pas de *else*. De plus, tout branchement impliqué dans un bloc `IT` doit être la dernière instruction de celui-ci comme dans l'exemple ci-dessus. Si le cette contrainte n'est pas respectée, le code devient imprévisible. Par ailleurs, aucun branchement ne peut avoir pour cible des instructions à l'intérieur d'un bloc `IT`. La documentation en annexe C est utile à lire pour tout connaître de l'instruction `IT`.

```
  cmp     r0, #10
  itee    lo        @ si r0 est inférieur à 10
toto:               @ étiquette autorisée mais saut dans ce bloc INTERDIT
  addlo   r0, #1    @ Alors (then) r0 = r0 + 1
  blhs    fonction  @ IMPREVISIBLE DANS UN BLOCK IT
  movhs   r0, #0    @ Sinon (else) r0 = 0
```

Dans le projet, aucune des instructions 16 bits n'a d'effet sur le registre d'état excepté l'instruction `cmp` [19]. Il en résulte que le registre APSR peut être modifié dans un block IT et que donc les conditions d'exécution des instructions altérée. Dans l'exemple suivant :

```
cmp   r0, r1          @ On compare r0 et r1 et le registre d'état est mis à jour
ittt  gt              @ On n'exécute les trois instructions suivantes que si gt est vérifié
  add    r2, r0, r1   @ r2 = r0+r1 -> le registre APSR n'est pas mis à jour
  cmpgt  R2, R3       @ si r0>r1 alors on compare r2 et r3 (APSR mis a jour)
  movgt  R4, R5       @ si r2>r3 alors r4 = r5
```

seul cmp met à jour le registre APSR. `add r2, r0, r1` qui normalement affecte le registre APSR ne l'affecte pas dans le bloc. Par conséquent, si l'instruction IT impose les conditions d'exécution à tous le bloc, la vérification de cette condition est locale à chaque instruction (c.-à-d., que le registre APSR doit être vérifié pour chacune des instructions add, cmp et mov du bloc)

19. *With the exception of CMP, CMN, and TST, the 16-bit instructions that normally affect the condition code flags, do not affect them when used inside an IT block*

Chapitre 3

Le langage d'assemblage du ARM Cortex-M

Pour programme un *ARM Cortex-M*, on utilise un langage informatique qui sera traduit par le compilateur dans le langage machine cible, c'est à dire dans le jeu d'instructions du processeur. Dans ce projet, on utilisera le langage assembleur, qui est un langage de bas niveau beaucoup plus 'proche' de la machine que le C. Grosso modo, chaque instruction assembleur correspond à une instruction machine (nous verrons des exceptions). Cependant, étant donné que ARM ne fabrique pas de processeurs, les spécifications de chaque processeur de type *ARM Cortex-M*varient d'un constructeur à un autre. De même, les spécifications des langages assembleurs pouvant gérées du code compatible *ARM Cortex-M* dépendent du compilateur utilisé. Cependant, ARM Ltd. a sortie un langage appelé *the ARM Unified Assembler Language* (UAL) qui fournie un ensemble de forme canoniques pour les instructions. Les exemples d'instruction fournis dans l'annexe [3] respectent la convention UAL.

Dans ce projet, nous allons donc utiliser la syntaxe UAL telle qu'implantée dans l'assembleur GNU ainsi que la syntaxe générique de cet assembleur pour les éléments non spécifiés par l'UAL (notamment les étiquettes). La syntaxe qui est présentée ici est volontairement moins permissive que celle de l'assembleur *GNU*. On se contente ici de présenter la syntaxe de manière intuitive.

Un programme se présente comme une liste d'unités, une unité tenant sur une seule ligne dont la syntaxe est la suivante :

```
    [étiquette] [mnémonique instruction] [opérandes] [@ commentaire]
ou
    [étiquette] [directive] [@ commentaire]
```

Chaque élément entre `[]` est optionnel sous certaines contraintes. Une ligne peut ne comporter qu'un champ étiquette, une instruction peut ne pas avoir d'étiquette associée, ni d'opérande, ni de commentaire. Les champs doivent être séparés par des séparateurs qui sont des combinaisons d'espaces et/ou de tabulations. Il est possible (et même recommandé pour aérer le texte) de rajouter des lignes blanches.

3.1 Les commentaires

C'est un texte optionnel qui est supprimé par l'assembleur. Cela dit ce n'est pas parce qu'ils n'apparaissent pas dans le fichier objet que les commentaires sont inutiles. . .Nous vous recommandons fortement de mettre des commentaires dans vos fichiers de test. Un commentaire commence sur une ligne par le caractère `@` et se termine par la fin de ligne.

Exemple

```
@ Ceci est un commentaire. Il se termine à la fin de la ligne
CMP r1,r2 @ Ceci est aussi un commentaire, qui suit une instruction CMP
```

3.2 Le champ étiquette

Dans un programme assembleur, il est possible de placer des étiquettes avant une instruction ou une directive. Une étiquette ainsi définie peut-être utilisée ailleurs dans le code assembleur pour designer l'instruction ou la directive qui a été étiquetée. Dans le code binaire issu de l'assemblage, les étiquettes prendront la valeur de l'adresse mémoire (relative) de l'instruction ou espace mémoire qu'elles désignent. Dans le code objet, elles sont regroupées dans la table des symboles.

Les étiquettes peuvent servir d'opérande à une instruction ou à une directive de l'assembleur. Une étiquette est une suite de caractères alphanumériques. Cette chaîne est suivie par le caractère « :». Le nom de l'étiquette est la chaîne de caractères alphanumériques située à gauche du caractère « :». Plusieurs étiquettes peuvent être associées à la même opération ou à la même directive.

Une étiquette ne peut être définie qu'une seule fois dans une unité de compilation. Sa valeur lors de l'exécution est égale à son adresse d'implantation dans la mémoire après le chargement. Elle dépend donc de la section dans laquelle elle est définie et de sa position dans cette section (cf. section 3.3.1).

Exemple

```
etiq1:  _etiq2:
etiq3:  CMP r1,r2 @  les trois étiquettes repèrent la même instruction CMP
```

3.3 Les directives

Les directives sont des commandes à destination de l'assembleur et non des instructions du microprocesseur. Elle sont destinées à être interprétées par l'assembleur [1].

Une directive commence toujours par un point («.»). Il y a trois familles de directives : les directives de sectionnement du programme, les directives de définition de données et les directives de gestion de symboles.

3.3.1 Directives de sectionnement

Selon le type de processeurs il peut exister plusieurs zones mémoires pour stocker les instructions et les données. Dans le cas du *ARM Cortex-M*, il est possible de d'utiliser de la RAM et de la mémoire FLASH pour stocker différents types de donnée. Cependant même dans les processeurs ne contenant qu'une seule zone mémoire cette séparation existe (notamment pour gérer les droits d'accès). Ainsi, des directives existent en langage assembleur pour spécifier les sections de code et de données.

- la section `.text` contient le code du programme (instructions) .
- la section `.data` est utilisée pour définir les données du programme.
- la section `.bss` est utilisée pour définir les zones de données non initialisées du programme (qui réservent juste de l'espace mémoire). Ces données ne prennent ainsi pas de place dans le fichier binaire du programme. Elles seront effectivement allouées au moment du chargement du processus. Elles seront initialisées à zéro.

Les directives de sectionnement s'écrivent par leur nom de section : `.text`, `.data` ou `.bss`. Elles indiquent à l'assembleur d'assembler les lignes suivantes dans les sections correspondantes.

1. on peut faire le parallèle entre les commandes du préprocesseur, par exemple `#define N 5` et les instructions du langage C `int i = N*5;`. Le premier modifiera le code C tandis que le deuxième sera effectivement traduit en suite d'instructions machine

Directive	Description
`.data`	Ce qui suit doit aller dans le segment DATA
`.text`	Ce qui suit doit aller dans le segment TEXT
`.bss`	Ce qui suit doit aller dans le segment BSS
`.thumb`	Ce qui suit doit être encodé par des instructions Thumb. À mettre en préambule de tout code assembleur du projet.
`.syntax unified`	!!!OBLIGATOIRE!!!, spécifie à l'assembleur d'utiliser la syntaxe UAL
`.thumb_func`	le symbole qui suit correspond à une fonction
`.global string`	la chaîne string est un symbole global
`.word w1, ..., wn`	Met les *n* valeurs sur 32 bits dans des mots successifs
`.byte b1, ..., bn`	Met les *n* valeurs sur 8 bits dans des octets successifs
`.ascii s1, ..., sn`	stocke les *n* chaînes de caractères à la suite en mémoire
`.space` *n*	Réserve *n* octets en mémoire. Les octets sont initialisés à zéro.

Table 3.1 – Directives de l'assembleur considérées dans le projet.

3.3.2 Les directives de définition de données

On distingue les données initialisées des données non initialisées.

Déclaration des données non initialisées Pouvoir réserver un espace sans connaître la valeur qui y sera stockée est une capacité importante de tout langage. Le langage assembleur fournit la directive suivante.

[étiquette] `.space` **taille** La directive `.space` permet de réserver un nombre d'octets égal à *taille* à l'adresse *étiquette*. Les octets sont initialisés à zéro.

```
toto: .space 13
```

La directive `.space` se trouve généralement dans une section de données `.bss`.

Déclaration de données initialisées L'assembleur permet de déclarer plusieurs types de données initialisées : des octets, des mots (32 bits), des chaînes de caractères, etc. Dans ce projet, on ne s'intéressera qu'aux directives de déclaration suivantes :

[étiquette] `.byte` **valeur** *valeur* peut être soit un entier signé sur 8 bits, soit une constante symbolique dont la valeur est comprise entre -128 et 127, soit une valeur hexadécimale dont la valeur est comprise entre 0x0 et 0xff. Par exemple, les lignes ci-dessous permettent de réserver deux octets avec les valeurs initiales -4 et 0xff. Le premier octet sera créé à une certaine adresse de la mémoire, que l'on pourra ensuite manipuler avec l'étiquette $Tabb$. Le second octet sera lui à l'adresse $Tabb + 1$.

```
Tabb:   .byte -4, 0xff
```

[étiquette] `.word` **valeur** *valeur* peut être soit un entier signé sur 32 bits, soit une constante symbolique dont la valeur est représentable sur 32 bits, soit une valeur hexadécimale dont la valeur est comprise entre 0x0 et 0xffffffff. Par exemple, la ligne suivante permet de réserver un mot de 32 bits avec la valeur initiale 32767 à une adresse de la mémoire, adresse que l'on manipulera ensuite avec l'étiquette Tabw.

```
Tabw:   .word 0x00007fff
```

[étiquette] `.ascii` **chaîne de caractères** *chaîne* est une suite de caractères entre guillemets. Cette chaîne est stockée en mémoire (attention pas d'ajout automatique du caractère \0). Par exemple, la ligne suivante permet de stocker la chaîne "Projet Info\n" dans le programme. L'étiquette `String1` référencera l'adresse du premier caractère de la chaîne. La taille totale de la chaîne sera de 13 caractères[2] (11 caractères alphanumériques + retour à la ligne + caractère de fin de chaîne).

```
String1:   .ascii "Projet Info\n\0"
```

3.3.3 Directives de gestion de symbole

L'écriture d'une étiquette est utile pour 'pointer' des zones mémoires. Selon l'utilisation qui en est faite, ces étiquettes vont être présentes ou non dans le programme sous la forme d'un symbole (représentant, une variable, une fonction, un espace mémoire...). Ces étiquette ont une portée *locale* à l'unité de compilation (ici un fichier assembleur avec toutes les inclusions ayant été effectuées). Cependant, il peut être utile de déclarer un symbole comme '*global*' afin d'étendre sa portée à l'ensemble du programme (composé généralement de plusieurs unités de compilation). Ainsi, dans l'exemple ci-dessous l'étiquette `entete` pourra être accessible à partir d'un autre fichier assembleur.

```
.global entete
.data
entete: .ascii "Projet Info\0"
```

Notez que `.global entete` a pour effet de déclarer un symbole mais non de le définir (on ne sait pas ce que c'est à ce moment du programme). Ainsi, bien que la directive `.global` est utilisée dans la section `text`, c'est bien à un symbole de la section data qu'elle s'applique (à l'endroit où le symbole est défini).

Un des éléments de programme qu'il est utile de partager est la fonction (élément de base des programmes C mais pas de l'assembleur) pour éviter d'avoir à la réécrire dans chaque unité de programme. Pour déclarer un symbole comme étant associé à une fonction on utilise la directive UAL `.thumb_func` qui permet spécifier que le symbole défini suivant est une fonction. L'exemple ci dessous illustre son usage :

```
.global fun    @ déclaration de fun comme global
.thumb_func   @ le prochain symbole défini sera une fonction
.global entete @ juste une déclaration et non une définition
               @ ce symbole n'est donc pas affecté par thumb_func
fun:           @ définition de fun, ce symbole est donc affecté par thumb_func
  push {lr}    @ début du code de la fonction
  ...
```

2. attention, les caractères accentués et certains caractères spéciaux sont codés sur plus d'un octet en UTF-8

```
main:
  bl fun      @ endroit où on 'appelle' fun
  ...
```

Dans cet exemple fun est déclaré comme global mais n'est défini que plus tard. C'est donc devant cette définition que la directive `.thumb_func` doit être placée pour qualifier fun comme étant une fonction. Comme fun est déclaré en global, il peut être accessible en dehors du code écrit dans le fichier.

3.4 Les instructions machines

Elles ont la forme générale ci-dessous, illustrée par l'instruction SUB. Les champs entre {} sont optionnels tandis que les champs entre <> sont obligatoire mais à choisir entre différentes valeurs.

```
mnémonique instruction        opérandes
SUB{S}<c><q>                  {<Rd>,} <Rn>, #<const>
SUB{S}<c><q>                  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

Le mnémonique correspond à une instruction. Celui-ci est souvent auto-explicatif. Un grand ensemble d'instructions possède un champ condition <c> (voir la table 2.1 pour l'ensemble des conditions possibles) qui rend l'exécution de l'instruction dépendente des drapeaux de l'APSR. Si une condition est donnée, le processeur détermine (avant même le fetch, donc sans perdre un cycle) si l'instruction doit être traitée ou ignorée. C'est un grand avantage de cette famille de processeur car les exécutions conditionnelles sont souvent sources de ralentissement dans la plupart des processeurs.

Le champ <q> permet de spécifié la taille voulue de l'instruction (`.N` (*Narrow*) pour du 16 bits ou `.W` (*Wide* pour du 32 bits). Si le champ n'est pas spécifié, dans la plupart des cas, l'assembleur essayera de trouver le code le plus compact. Si le champ est spécifié mais qu'il n'est pas possible de le respecter (p.ex., ne tient pas dans une instruction 16 bits) alors le compilateur émet une erreur.

Le drapeau `S` permet d'ordonner au processeur de modifier le registre APSR selon le résultat de l'opération. Très pratique si l'on veut gérer un overflow ou une retenue après une opération arithmétique.

On peut donc voir que les mnémoniques peuvent prendre plusieurs formes pour une seule et même instruction. Pour ajouter un peu de sel, il se trouve que les opérandes peuvent également prendre des formes assez complexes

Les opérandes vont permettre d'identifier les données sur lesquelles les instructions agissent. Dans notre exemple

```
{<Rd>,} <Rn>, #<const>
ou
{<Rd>,} <Rn>, <Rm> {,<shift>}
```

le registre `Rd` spécifie la destination (cad, l'endroit où stocker le résultat de la soustraction), le registre `Rn` est le premier opérande de la soustraction mais également la destination quand `Rd` n'est pas donné. L'opérande `#<const>` est une valeur immédiate encodée sur 3, 8 ou 12 bits. Du coup, comment faire des soustractions avec un nombre supérieur à $2^{12} - 1$? Pour cela le fascinant mécanisme mis au point par ARM permet de représenter certaines valeurs sur 32 bits par une constante sur 12 bits. Cette constante est appelée *Modified immediate constant* et n'est pas simplement un code binaire

Mnemonic	description	valeur n	champ type (2 bits)	champ immediate (5bits)
omis	équivalent à LSL #0	0	0x0	0x0
LSL #n	logical shift left	$0 \leq n \leq 31$	0x0	n
LSR #n	logical shift right	$1 \leq n \leq 32$	0x1	n is $n < 32$, 0 si $n == 32$ lllll
ASR #n	arithmetic shift right	$1 \leq n \leq 32$	0x2	n is $n < 32$, 0 si $n == 32$
ROR #n	rotate right	$1 \leq n \leq 31$	0x2	n
RRX #n	*non utilisé dans le projet*	0	0x2	0

Table 3.2 – Liste des décalages applicables aux registres

classique sur 12 bits. En fait, cette valeur permet de représenter un nombre 32 bits soit par un nombre sur 8 bits décalable à gauche, soit par un motif sur 8 bits replicable. Ainsi les nombres suivants sont encodables par un *Modified immediate constant*.

```
0xAB000000
0x0AB00000
0xAB00AB00
0x00AB00AB
0xABABABAB
```

Toutes les autres valeurs 32bits qui ne sont pas 'ARM friendly' sont résolues par l'assembleur, par exemple, en stockant la valeur immédiate 32 bits en mémoire et en la récupérant dans un registre pour l'opération (mécanisme du *literal pool* un morceau de mémoire dans le code qui contient des constantes).

Dans la deuxième syntaxe d'opérandes le registre `Rm` est le deuxième opérande de la soustraction. C'est à dire que l'opération effectuée sera `Rd = Rn - Rm`. Le quatrième opérande (optionnel) représente une transformation opérée sur la valeur contenue dans le registre `Rm` avant l'opération (mais qui n'affecte pas `Rm` de manière définitive). Le tableau 3.2 donne la liste des différents décalages applicables.

Pour finir, on peut donner les différentes formes suivantes de SUB valides :

```
SUB r1,#1              @ décremente r1
SUB r0,r1,#1           @ r0 = r1 - 1
SUB r0,r1,#0x500050    @ la constante est valide
SUB r0,r1,r2           @ r0 = r1 - r2
SUB r1,r2              @ r1 = r1 - r2
SUB r1,r1,r2, LSL #4   @ r1 = r1 - (r2 * 16)
SUB r1,r1,r2, ASR #2   @ r1 = r1 - signe(r2/4)
```

3.4.1 Les modes d'adressage

Comme nous l'avons vu, les instructions du microprocesseur peuvent avoir plusieurs types d'opérande. On appelle mode d'adressage d'un opérande la méthode qu'utilise le processeur pour déterminer où se trouve la valeur de l'opérande. Dans ce projet nous nous intéresserons aux mode d'adressage suivants :

3.4.2 Adressage registre direct

Dans ce mode, la valeur de l'opérande est contenu dans un registre et l'opérande est désigné par le nom du registre en question.

Exemple :

```
SUB r2, r3, r4   @ les valeurs des opérandes sont dans les registres 3 et 4
                 @ le résultat est placé dans le registre 2
```

3.4.3 Adressage immédiat

La valeur de l'opérande est directement fournie dans l'instruction.

Exemple :

```
X:
    SUB r2, r3, #200   @ valeur immédiate sur 8 bits
    SUB r2, r3, #0x3f  @ idem avec une valeur immédiate hexadécimale
    SUB r2, r3, X      @ soustrait à r2 à la valeur (et non le contenu) de X (adresse mémoire)
```

Selon l'instruction, la valeur immédiate doit être codable sur 2,3,4,5,7,8,12 ou 16 bits. Comme chaque instruction possède plusieurs encodages, c'est l'assembleur qui déduit de la valeur immédiate quel est l'encodage le plus adéquat.

3.4.4 Adressage indirect avec base et déplacement

Dans ce mode, intervient un registre appelé *registre de base* qui contient une adresse mémoire, et une constante signée codée sur deux octets appelée *déplacement*. La syntaxe associée par l'assembleur à ce mode est `offset(base)`.

Pour calculer l'adresse de l'opérande, le processeur ajoute au contenu du registre de base `base` la valeur sur 5, 8 ou 12 bits du déplacement `offset`. En plus de pouvoir spécifier ce décalage, on peut également modifier le registre de base après ou avant l'accès mémoire (à la manière de `int a = tab[i++]` ou `int a = tab[++i]` qui n'ont pas le même effet).

Exemple :

```
STR r2, [r3,#-200]   @ r2 = memory[(r3) - 200]
STR r2, [r3,#-200]!  @ r2 = memory[(r3) - 200], r3= r3-200
STR r2, [r3],#-200   @ r2 = memory[(r3)], r3= r3-200
```

Dans dans première ligne de cet exemple, le mot qui se trouve 200 bits avant l'adresse 'pointée par' r3 est copié dans le registre r2. C'est le mode **Offset** simple.
Dans la deuxième ligne, le mot qui se trouve 200 bits avant l'adresse 'pointée par' r3 est copié dans le registre r2 puis 200 est soustrait à r3. C'est le mode **Pre-indexed**.
Enfin, dans la troisième ligne, le mot qui se trouve à l'adresse 'pointée par' r3 est copié dans le registre r2 puis 200 est soustrait à r3. C'est le mode **Post-indexed**.

3.4.5 Adressage relatif

Ce mode d'adressage est utilisé par les instructions de branchement. L'adresse du saut est déterminée à partir d'une étiquette `label` dont le décalage `offset` par rapport à PC est calculé. La valeur de cet `offset` peut varier de -16777216 (0xFF000000) à 16777214 (0x00FFFFFE). Cette valeur doit absolument être paire. Lors du décodage, comme une instruction occupe au moins 2 octets pour obtenir

le saut à réaliser en mémoire, l'`offset` est d'abord décalée de 1 bits vers la gauche puis ajouté au compteur PC courant. Par exemple, un offset codé 0xFD dans l'instruction correspond en réalité à un offset de 0x1FA ! La valeur est ensuite ajoutée au compteur programme pour déterminer l'adresse de saut.

Exemple :

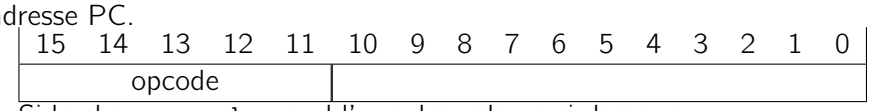
```
X:
   CMP r2,r3
   B X     @ ici, -4 est stocké dans l'instruction
           @ c'est donc la valeur -8 qui sera ajoutée au PC
```

3.5 Instructions étudiées dans le projet

Les instructions du *ARM Cortex-M* qui devront être traitées dans le projet sont toutes données en annexe C. Toutes les instructions ne seront pas traitées (en particulier les entrées-sorties, la gestion des valeurs flottantes...).

Comme dit précédemment le *ARM Cortex-M* n'accepte que le jeu d'instruction Thumb2 qui est composée d'instructions sur 16 bits (demi-mot) ou 32 bits (mot). Il en résulte que les instructions sont toujours codées sur des adresses alignées sur un demi-mot, c'est-à-dire divisibles par 2. Cette restriction d'alignement favorise la vitesse de transfert des données.

La distinction entre ces deux types est donnée par les 5 bits de poids forts du demi-mot lu à l'adresse PC.



Si le champ`opcode` prend l'une des valeurs ci-dessous :

```
0b11101
0b11110
0b11111
```

Alors il s'agit d'une instruction Thumb 32 bits (et donc le prochain demi-mot doit être lu pour composer l'instruction) sinon, il s'agit d'une instruction Thumb 16bits qui peut être directement décodée. Pour le détail binaire, tout est dans la doc ! Ce qui suit donne les grandes lignes de ces instructions.

3.5.1 Affectation de données, comparaison et décalage

Ce sont les opérations permettant de copier des données ou des constantes de/vers des registres. Le code ci-dessous n'implique que ce type d'opération.

```
CMP r1,r2          @ r1 == r2 ? -> affecte le registre APSR
MOV r7, #1234      @ r7 = 1234
MOVT r8,#0xAAAA    @ r8 = 0xAAAAxxxx -> charge les 16 bits
                   @ dans les deux octets de poids fort de r8
MOV R0, R1, LSR #2 @ R0 = R1 / 4
MOVEQ R0, R1       @ R0 = R1 si Z == 1
MOVS R0, R1        @ R0 = R1, change les drapeaux de l'ASPR
LSLS r0,r1,1       @ r0 = r1 * 2, affecte ASPR
```

3.5.2 Opérations arithmétiques

Ce sont les opérations permettant de réaliser des addition, soustraction, multiplication etc. Elles peuvent modifier les drapeaux de l'APSR. Le code ci-dessous n'implique que ce type d'opération.

```
MOV r0,#10      @ r0=10
SUB r1,r0,#1    @ r1 = r0 -1 = 9
ADD r1,-1       @ r1 = r1 - 1 = 8 decrementation
MUL r2,r0,r1    @ r2 = r0 * r1, seul les 32 de poids faibles sont stockés dans r2
```

3.5.3 Opérations Logiques

Ce sont les opérations permettant de réaliser des ET, OU EXCLUSIF etc. Elles peuvent modifier les drapeaux de l'APSR. Le code ci-dessous n'implique que ce type d'opération.

```
ANDS r0,r1,#0xFF000000  @ r0 = r1 & 0xFF00000, copie l'octet de poids fort
ORR.W r0,r1,r2 LSL #2   @ ou bit à bit entre r1 et R2 décalé de 2 à gauche.
                        @ le résultat est stocké dans r0
```

3.5.4 Accès mémoire

Ce sont toutes les instructions qui vont lire ou écrire dans la mémoire. La valeur lue ou écrite est dans un registre. Le code ci-dessous n'implique que ce type d'opération.

```
.text
toto:
LDR r0,[pc,#8]    @ r0 = valeur à l'adresse PC+8
STR r0,[pc,#-12]  @ stocke la valeur dans r0 à
                  @ l'adresse PC-12

ldr r2,toto       @ charge dans r2 la valeur à l'adresse
                  @ de toto soit le code binaire de la première
                  @ instruction LDR !! ne fonction que dans
                  @ la section text

ldr r3,=titi      @ charge dans r3 l'adresse de titi
str r0,[r3]       @ stocke r0 dans le mot 'pointé par' titi
                  @ va écraser la valeur 0xAB

.data
titi: .word 0xAB
```

Comme illustré dans l'exemple les instructions LDR et STR ont principalement 2 formes :

```
LDR Rd, etiquette   @ met le mot à l'adresse étiquette dans Rd
LDR Rd, =etiquette  @ met l'adresse etiquette (la valeur de etiquette) dans Rd
```

Dans le premier cas, etiquette doit se trouver dans la même section que l'instruction (ici .text). Autrement le compilateur ne peut générer une adresse PC-relative.

Dans le deuxième cas, l'assembleur génère plusieurs instructions. En effet, comme les adresses sont sur 32 bits et que les instructions sont encodées sur 16 ou 32 bits, Une instruction 32 bits ne peut pas contenir un opcode, un numéro de registre et une adresse de 32 bits[3]...Les assembleurs font donc les remplacements suivants :

1. l'adresse de etiquette sera copiée dans le code (.text) dans ce que l'on nomme le *literal pool*
2. une instruction LDR Rd, [PC+Offset] mettra l'adresse de étiquette dans Rd. Offset représentera l'écart entre l'adresse de l'instruction et l'adresse de la constante dans le *literal pool*.

Dans le cas où etiquette se trouve dans une autre section (.data ou .bss) ou une autre unité de compilation, alors la constante devra être relogée (cf. section 4).

L'accès mémoire concerne également une tâche importante qui est la gestion de la pile. L'exemple ci-dessous illustre les opérations courantes :

```
.text

push {r5,r12}     @ sauve les registre 5 et 12 dans la pile
add sp,sp,#-8     @ mise à jour du stack pointeur

pop {r6,r7}       @ extrait les deux mots les plus récents et
                  @ les copie dans r6 et r7
                  @ effet équivalent à r6=r5 et r7 = r12
add sp,sp,#8      @ mise à jour du stack pointeur

.thumb_func
mafonction:       @ sous-routine
push {lr}         @ sauve l'adresse de retour
...
pop {pc}          @ mise de pc à l'adresse de retour
                  @ pc = lr (car lr précédemment sauvé dans la pile)
```

Comme illustré ici, la pile permet de sauver l'état des registres avant un appel à une fonction. La pile, permet par ailleurs de compenser le nombre restreint de registres ou dans certains cas de créer des variables locales.

3.5.5 Branchement

Ce sont toutes les instructions qui vont permettre de 'sauter' à un autre endroit du programme, généralement selon une condition. Les adresses de sauts peuvent être absolues ou relatives. Les fonctions sont appelées à travers des branchements. Le code ci-dessous n'implique que ce type d'opération.

```
CMP r1,r2    @ compare les registres
NOP          @ ne fait rien
etiq0:
  BEQ etiq1    @ si égaux saute à etiq1
  mov r1,r2    @ r1=r2

etiq1:
```

3. Sauf avec des entier ARM-friendly...

```
  BL mafonction  @ saut dans une sous routine (cf. plus haut)
  B etiq0
```

L'appel à BL à pour effet de sauver la valeur de PC dans le registre LR. C'est pourquoi le registre LR doit être sauvé dans la pile au début de chaque fonction (pour éviter d'écraser sa valeur et donc de perdre son chemin de retour).

Le calcul des adresses est toujours effectué relativement à la valeur de PC. Si l'on s'appuie sur l'exemple dont le code objet assemblé est ci-dessous :

```
   0: 4291       cmp r1, r2

<etiq0>:
   2: d001       beq.n 8 <etiq1>
   4: bf00       nop
   6: 4611       mov r1, r2

<etiq1>:
   8: f000 f80c  bl 24 <mafonction>
   c: e7f9       b.n 2 <etiq0>
```

L'offset du premier beq.n vaut 1 << 1 c'est à dire 2. La valeur de PC vaut toujours l'adresse de l'instruction *N* plus 4. Donc PC vaut 6. L'adresse de saut se calcule donc par PC+offset = 2+6 = 8 qui est bien l'adresse de etiq1.

L'offset du dernier b.n vaut 0xF9 soit -7, l'offset vaut donc $-7 << 1 = -14$. L'adresse de PC pour cette instruction vaut $0xC + 0x4 = 0x10$ (16). @cible vaut donc PC + offset = 16 -14 = 2 qui est effectivement l'adresse pointée par etiq0.

3.5.6 Contrôle

Ce sont toutes les instructions qui vont affecter directement le microprocesseur. Typiquement, c'est dans cette famille que l'on trouve les interruptions. Nous en parlerons très peu dans ce projet.

Chapitre 4

Relocation

4.1 Principe de la relocation

4.1.1 Nécessité d'un code relogable

Le rôle de l'assembleur est, à partir du fichier source en langage assembleur, de générer le code binaire de l'ensemble des instructions et des données composant le programme. C'est ce code binaire qui sera chargé en mémoire puis exécuté lorsque le programme sera lancé.

Certaines instructions peuvent être codées directement, indépendamment du reste du programme et de l'adresse à laquelle elles seront chargées en mémoire. C'est notamment le cas des opérandes des instructions ne mettant en jeu que des registres ou des valeurs immédiates. Par contre, quand les opérandes des instructions sont des étiquettes, comme dans le cas d'un saut ou d'un accès à la zone `.data`, le code binaire de ces instructions ne peut pas être produit à l'assemblage. En effet, les adresses définitives référencées par les étiquettes ainsi que les adresses des sections ne seront connues que lors du chargement du programme en mémoire.

Les exemples suivants illustrent les problèmes rencontrés dans ce cas.

Exemple 1:

```
.text
  MOV r3,#123        @ met 123 dans le registre 3
  LDR r0,=X          @ met l'adresse de X dans r0
  STR r3,[r0]        @ écrit le contenu de r3 à l'adresse X

.data
X: .word 0         @ réservation d'un mot initialisé à 0 dans data
```

Dans cet exemple, l'instruction MOV ne pose aucun problème. Par contre le LDR dépend de l'adresse à laquelle correspond l'étiquette `'X'`. Or cette adresse ne peut être connue avant que l'adresse d'implantation des sections `.text` et `.data` ne soient elles mêmes connues. Or celles-ci ne seront connues qu'au chargement du programme. Mais le programme ne pourra jamais être chargé s'il n'est pas assemblé... Le problème est donc : Comment coder l'instruction LDR de l'exemple si l'offset ne peut pas être calculé ?

Exemple 2:

```
.data
X: .byte 0xAB   @ réservation d'un octet initialisé à 0xAB
Y: .byte Z      @ réservation d'un octet initialisé à l'adresse
                @ référencée par l'étiquette Z
Z: .word 0      @ réservation d'un mot initialisé à 0
```

Dans cet exemple, la valeur qui suit l'étiquette `Y` est en fait l'adresse référencée par l'étiquette `Z`. Comme dans le cas précédent, il est nécessaire de connaître l'adresse d'implantation de la section `.data` pour déterminer les valeurs de `X, Y, Z`.

Exemple 3:

```
  BL mafonction    @ Appel de la procédure "mafonction"
  NOP              @ On ne fait rien
  B end            @ Retour ici après la procédure, on se branche
                   @ sur la fin du programme
  NOP              @ rien

.global mafonction @ la fonction est globale et peut donc
.thumb_func        @ être utilisée en dehors de cette unité
mafonction :       @ début de la procédure
  MOV r1,#0
  mov pc,lr        @ fin de procédure fonction, retour à l'appelant

end:
```

Ici, deux étiquettes sont utilisées par des instructions de branchement. Ces deux cas ne sont cependant pas équivalents :

- Le codage du branchement nécessite de calculer le décalage entre l'instruction `B` et l'étiquette `end`. Mais puisque cette étiquette se situe dans la même section que le branchement lui-même, ce décalage peut être calculé lors de l'assemblage. Mais ce n'est pas toujours le cas, car cette étiquette aurait très bien pu être déclarée dans un autre fichier assembleur. Ce décalage n'aurait été dans ce cas calculable qu'au moment de l'édition de liens.
- Le problème est différent pour l'instruction `BL`. En effet, le symbole `mafonction` est *global*, ce qui signifie que sa véritable adresse ne sera connue qu'à l'édition des liens. Donc, même si la position de l'étiquette `mafonction` est ici connue *par rapport* à l'instruction `BL`, il manque la position effective en mémoire de cette instruction pour pouvoir déterminer son codage complet.

En résumé, **le code d'un programme n'est déterminé de manière absolue qu'à partir du moment où les adresses mémoires auxquelles seront implantées les différentes sections (.text, .data ...) sont connues.**

4.1.2 Codage en position, flat binary file

Une solution est d'assembler le programme en définissant à l'avance, l'adresse mémoire de la section `.text`. Par exemple si `.text` est implanté en 0x5000, la section `.data` sera ensuite placée à la suite et ainsi de suite pour toutes les autres sections. De cette manière, il devient possible de coder l'ensemble des cas des exemples 1, 2 et 3.

Le programme peut ainsi être chargé directement en mémoire **UNIQUEMENT** *à l'adresse prévue* et exécuté en l'état. Par contre le programme n'est plus valide dès que les adresses utilisées ne sont plus disponibles. C'est notamment le cas si la machine ne dispose pas de suffisamment d'espace mémoire ou si les plages d'adresses sont déjà utilisées par d'autres programmes.

4.1.3 Code relogeable

Une autre solution consiste à créer des fichiers objets dits *relogeables* qui contiennent des informations permettant de **modifier le code binaire au moment du chargement du programme**, juste

avant l'exécution, pour l'adapter dynamiquement aux plages d'adresse qui lui auront été allouées. On appelle cette opération la *relocation*.

Le principe des codes relogeables repose sur le fait que si les adresses effectives des instructions et étiquettes ne sont pas connues au moment de l'assemblage, leur position *relative* au début de la section où elles se trouvent l'est ! Ainsi dans les exemples ci-dessous :

— l'étiquette `X` de l'exemple 1 correspond en fait à l'adresse de la section `.data`
— l'étiquette `Y` (resp. `Z`) de l'exemple 2 correspond à l'adresse de la section `.data` + 2 octets (resp. 4 octets).
— l'étiquette `mafonction` (resp. `end`) de l'exemple 3 correspond à l'adresse de la section `.text` + 12 octets (resp. 18 octets) et l'instruction `BL` est implantée en début de section.

Lors de l'assemblage, il suffit donc de générer une version *'relative'* du code binaire et d'inclure dans le fichier objet toutes les informations qui permettent d'adapter ce code lors de son positionnement en mémoire. Un fichier relogeable contient donc au moins :

— le code des sections assemblées avec des informations liées au positions relatives des instructions et données par rapport au début de chaque section.
— une *table de relocation* indiquant, pour chaque section, les positions des adresses relatives à mettre à jour au moment du chargement et le mode de relocation (définissant les calculs à effectuer pour déterminer les bonnes adresses effectives).
— une *table des symboles* contenant les labels des étiquettes (chaînes de caractères) associées à leur adresse relative (pour les symboles définis localement).

Le chargement, avant l'exécution du programme est alors précédé d'une phase de relocation :

1. le fichier objet est lu pour déterminer le nombre d'octets nécessaires (taille des sections) pour copier les instructions et données dans la mémoire.
2. l'éditeur de liens (ou sur certains systèmes le chargeur dynamique) détermine à quelles adresses vont être placées les différentes sections composant le programme
3. La relocation proprement dite a alors lieu. À partir des adresses d'implantation déterminées par le système et des informations de la table de relocation, elle convertit les *adresses relatives* des étiquettes en *adresses absolues*. Selon les modes de relocation à utiliser, cette conversion est plus ou moins simple.
4. le code ainsi modifié est prêt a être exécuté.

Le principal intérêt des codes relogeables est d'être portable d'une machine à une autre puisqu'ils ne sont pas spécifiques à une configuration mémoire. Ils sont également moins encombrants, en particulier dans le cas de programme multi-fichiers ou utilisant des bibliothèques car les procédures ne sont codées qu'une seule fois.

Compilation séparée

Les fichiers relogeables permettent la compilation séparée, c'est-à-dire la création d'un fichier objet qui fait partie d'un programme plus vaste utilisant plusieurs fichiers objets qui seront rassemblés par l'édition de liens.

Au moment de l'assemblage, il est possible que certains symboles soient indéfinis (par exemple, procédures définies dans un autre fichier) et c'est l'éditeur de liens qui ira chercher leur définition. Pour chaque symbole indéfini, l'assembleur va donc laisser un "trou" à l'endroit où ce symbole est référencé et noter dans le fichier binaire à quel symbole correspond ce trou, ainsi que l'action à exécuter pour l'adresse finale au moment où il aura connaissance de l'adresse de ce symbole.

4.2 Informations nécessaires à la relocation

4.2.1 Table de relocation

Les informations nécessaires à la relocation sont définies dans les tables de relocation qui seront incluses par l'assembleur dans le fichier objet. Une table est associée à chaque section contenant des symboles à reloger. La table associée à la section `.text` (resp. `.data`) est appelée `.rel.text` (resp. `.rel.data`). Chaque ligne d'une table de relocation est définie par les lignes suivantes :

- `offset` : la position de l'entrée à modifier, en nombre d'octets par rapport au début de la section à laquelle la table est associée.
- `type` : le mode de relocation
- `value` : le symbole par rapport auquel il faudra faire la relocation. Dans le cas de symboles locaux, `value` est *l'index* de la zone contenant le symbole et dans le cas de symboles globaux (éventuellement non définis à l'assemblage) c'est l'index du symbole en question.

Dans l'exemple 2, l'entrée à reloger est un mot à l'adresse `Y` et le symbole à reloger est `Z` qui se trouve aussi dans la section `.data`. La table associée est donc :

```
[.rel.data]
Offset          Type                Value
00000004        R_ARM_ABS8          .data
```

Dans l'exemple 3, l'entrée à reloger est l'instruction `BL`, la première de la section `.text` et le symbole recherché est l'étiquette `mafonction` qui se trouve également dans la section `.text`. La table associée est donc :

```
[.rel.text]
Offset          Type                Value
00000000        R_ARM_THM_CALL      .text
```

4.2.2 Champ addend

Une dernière donnée pour la relocation d'une entrée est le `addend`, c'est-à-dire la valeur binaire présente dans l'espace qui va être modifié par la relocation. Cette valeur, contenue dans le fichier objet, va être récupérée par l'éditeur de lien ou le chargeur dynamique et utilisée avec les informations de la table de relocation pour déterminer le nouveau code de l'instruction ou de la donnée. Elle sera finalement écrasée par le nouveau code au moment du chargement en mémoire.

Selon le mode de relocation, le addend correspond à un nombre variable de bits de poids faible de l'entrée à reloger.

Dans le cas de symboles définis mais dont on ne connaît pas l'adresse absolue, `addend` est en générale l'adresse relative du symbole par rapport au début de section. Dans le cas des symboles indéfinis (référence à un symbole défini dans un autre fichier), l'assembleur, n'ayant pas d'adresse relative laisse généralement le addend à zéro.

4.2.3 Modes de relocation du ARMv7-M

Le *mode de relocation* détermine le calcul spécifique à effectuer pour reloger une entrée (instruction ou donnée). Beaucoup de modes existent mais nous nous restreindrons à ceux définis ci-dessous :

R_ARM_ABS8 Ce mode est utilisé pour reloger une donnée de taille 1 octet. Seuls 8 bits de l'entrée sont modifiés.

R_ARM_ABS32 Ce mode est utilisé pour reloger une donnée de taille 32 bits. Les 32 bits de l'entrée sont modifiés.

R_ARM_THM_CALL Ce mode est utilisé pour reloger une instruction de type `BL` qui saute dans une fonction. Les 5 bits de poids fort (*opcode*) ne seront jamais modifiés par la relocation.

Les notations utilisées pour la description des calculs de relocations sont les suivantes :

— **P**lace désigne l'adresse de l'entrée à reloger, c'est-à-dire l'adresse allouée par l'éditeur ou le chargeur à la section plus la valeur `offset` associée à l'entrée, c'est à dire l'adresse "finale" de l'élément à reloger.

— **A**ddend désigne la valeur à **a**jouter pour calculer la valeur du champ à reloger (c'est la valeur du champ avant relogement). Selon le mode de relocation il s'agit d'un nombre variables de bits de poids faible de l'entrée (en fait l'extension signée de `addend`)

— **S**ymbole désigne l'adresse "finale" du symbole par rapport auquel on reloge l'instruction. Si le symbole est de type `STT_SECTION` alors **S** prend la valeur de l'adresse du segment auquel le symbole appartient. Sinon, l'adresse est calculée à l'aide du champs `st_value` qui est ajouté à l'adresse de début de section à laquelle il appartient.

— **T** vaut 1 si S est de type `STT_FUNC` et que le symbole pointe sur une instruction Thumb (ce qui est toujours le cas pour l'ARMv7-M). T vaut 0 sinon.

Avec ceci, les calculs de relocation sont :

Mode	Adresse du 1er bit à modifier	Nb de bits à modifier	Valeur à écrire
`R_ARM_ABS8`	P	8	S+A
`R_ARM_ABS32`	P	32	(S + A) | T
`R_ARM_THM_CALL`	P+5 bits	16	((S + A) | T) – P

4.3 Format `elf`

Plusieurs formats de fichier relogeable existent pour différents systèmes d'exploitation. Dans ce projet nous utiliserons le format ELF qui est celui utilisés par les systèmes UNIX. Pour la description de ce format veuillez vous référer à l'annexe A.

Chapitre 5

Spécifications de l'émulateur, travail à réaliser

Un émulateur est un logiciel capable de reproduire le comportement d'un objet. Dans notre cas il s'agit de reproduire le comportement d'un programme executé sur un *ARM Cortex-M*. *Reproduire le comportement* signifie plus précisément que l'on va définir pour notre émulateur une mémoire et des registres identiques à ceux du microprocesseur, puis on doit réaliser l'évolution de l'état de cette mémoire et de ces registres selon les instructions du programme. On doit obtenir les mêmes résultats que ceux que l'on obtiendrait avec une exécution sur la machine cible *ARM Cortex-M* réelle mais pas forcément de la même façon : c'est le principe du *faire semblant* (par opposition au *faire comme*).

La Figure 5.1 présente le diagramme des différents composants d'un émulateur et de leurs interactions. Le programme assembleur, sous forme d'un fichier ELF, contient non seulement les instructions mais aussi les données du programme. Celui-ci est chargé dans les segments de mémoire du microprocesseur. Les registres contiennent les données représentant les arguments des instructions arithmétiques et logiques. On peut noter la présence du **PC** (*Program Counter* ou *Instruction pointer*) qui est l'indice donnant l'adresse de la prochaine instruction à exécuter. Ce PC est mis à jour par le module de décodage/exécution qui doit interpréter les instructions pour connaître la prochaine à exécuter. Par exemple, une simple opération d'addition (`ADD r9,r10,r11`) impliquera un simple incrément du PC (la prochaine instruction est celle suivant l'addition) mais un branchement tel que `BNE ailleurs` demandera l'exécution complète de l'instruction pour savoir si oui ou non il faut sauter à l'adresse `ailleurs` ou exécuter l'instruction suivante.

Pour débugger un programme, il faut pouvoir l'arrêter au milieu d'une exécution et pouvoir analyser son état (valeur de registre, valeur du PC, etc.). Pour cela, on utilise des points d'arrêt qui permettent de stopper une exécution à une adresse précise ou une exécution pas-à-pas. L'ajout de points d'arrêt et le mode d'exécution est décidé par l'utilisateur, à travers l'interpréteur de commande. Cet interpréteur est l'interface avec l'utilisateur qui lui permet de contrôler l'émulateur.

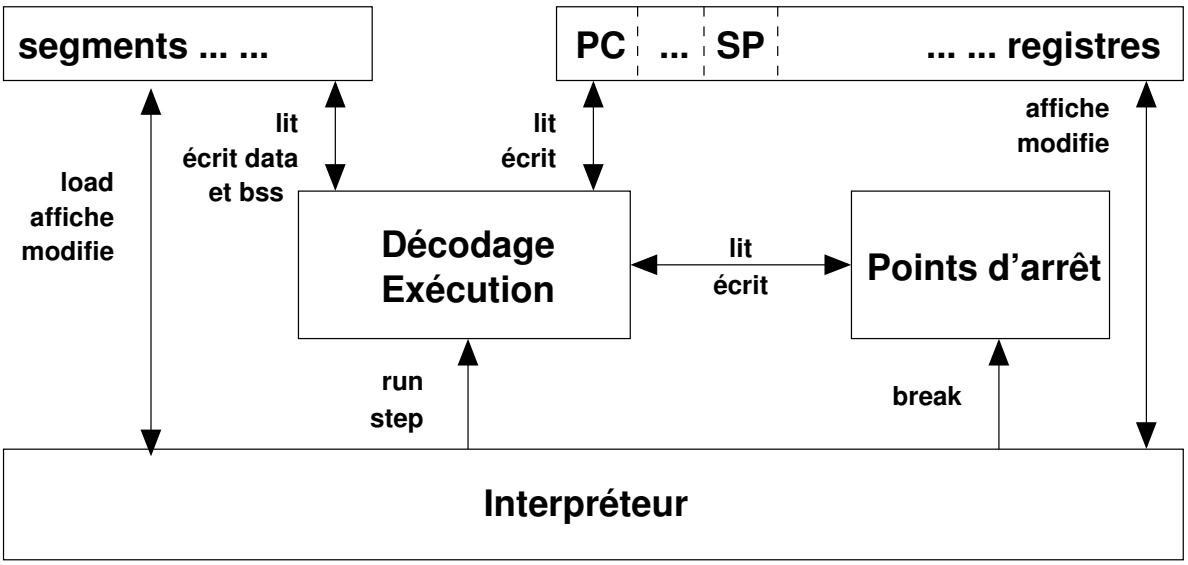


Figure 5.1 – Architecture d'un émulateur de microprocesseur

5.0.1 Modes d'utilisation de l'émulateur

L'émulateur sera un programme C qui sera appelé en tapant sous Linux la commande :

```
emul-arm
```

ou

```
emul-arm script_filename
```

où `script_filename` est le nom d'un fichier de commandes à exécuter. En effet, tout comme le *shell*, l'émulateur pourra fonctionner sous deux modes.

- Le mode dit *interactif* dans lequel chaque commande de l'utilisateur est interprétée directement, exécutée puis l'invite de commande est rendu en attente d'une nouvelle commande. Dans notre cas, ce mode est utilisé lorsque l'exécutable est lancé sans paramètre.
- Le mode dit *non-interactif* dans lequel l'interpréteur va lire les commandes une à une dans un fichier[1]. Ce fichier est couramment appelé un *script*[2]. Dans notre cas, ce mode est utilisé lorsque l'exécutable est lancé avec le chemin d'un fichier en paramètre.

Pour faire un parallèle avec les cours de programmation de première année, taper chaque commande `gcc -c fichier.c` correspond au mode *interactif* alors qu'utiliser un Makefile correspond au mode *non-interactif*[3]. Le mode *non-interactif* sera particulièrement utilisé pour automatiser des tests de l'émulateur ainsi que des fichiers objet.

Format des fichiers de script

Nous partirons du principe que les fichiers de script sont composés d'une seule commande par ligne et que les commentaires sont précédés d'un dièse #. L'exemple suivant illustre le contenu d'un fichier de script pour l'interpréteur :

```
#fichier de test de l'émulateur
load fichier.o    # charge en mémoire le fichier objet
disp reg all      # affiche le contenu de tous les registres
exit
#sort du programme
```

5.1 Machine simulée

5.1.1 Adresse des segments de mémoire

Pour charger le programme en mémoire, il faudra (au moins dans un premier temps) respecter les contraintes ci dessous :

- Le premier segment, `.text` (ou `.rodata` si présente) sera placé à une adresse fixée par défaut par le programme. Cette adresse doit également être paramétrable via l'interpréteur.
- Les adresses d'implantation des autres segments dépendent de la taille du premier segment et ne peuvent donc être déterminées *a priori*. La seule contrainte est que les adresses mémoires des segments `.text, .data, .bss, .rodata`, etc soient toujours multiples de 0x1000 (soit

1. C'est en fait plus générique car il s'agit plus généralement d'un *flux* d'entrée.
2. Un *script* désigne tous les programmes textuels qui n'ont pas besoin d'être compilés avant d'être exécuté. On dit qu'ils sont *interprétés*. . .Par exemple, les fichiers de programmes de base de MATLAB sont des scripts.
3. Ceci n'est pas tout à fait exact car c'est l'utilitaire `make` qui interprète le Makefile, mais vous nous excuserez cette approximation car cette comparaison n'est pas dénuée de vertu pédagogique.

4096 octets ou 4ko). On parle d'alignement de pages. Par exemple, si le segment `.text` est implanté en 0x2000 et fait une taille de 4352 octets alors le segment `.data` devra commencer en 0x3000. Ainsi si le segment `.data` a une taille de 200 octet, la section `.bss` devrait commencer en 0x4000 [4].

— Comme indiqué en section 2.4, la fin de la pile sera implantée 1ko avant la fin de la mémoire et devra avoir une taille parametrable.

Notion d'adresse virtuelle, adresse réelle

Il faut bien comprendre que les valeurs des adresses des segments précités sont *virtuelles*. En effet, nous allons émuler la mémoire, c'est-à-dire, faire croire, par exemple, que le segment `.text` débutera en 0x1000 alors que les données réelles qui lui sont associées seront véritablement stockées dans une zone mémoire allouée sur la machine *hôte* dont l'adresse est inconnue au démarrage du programme et pour laquelle nous n'avons pas prise [5]. Ainsi, si l'on veut accéder à l'octet en 0x100c d'adresse virtuelle, votre programme devra trouver à quelle adresse réelle il se trouve (qui peut très bien être en 0x7fff14b7a484 !). Votre programme devra donc contenir un mécanisme pour traduire les adresses virtuelles en adresses réelles et vice-versa.

5.1.2 Pile d'exécution

Une pile est une structure de données qui permet de sauvegarder les données locales et le passage de paramètres lors d'appels de fonctions. Cette structure en pile permet notamment les appels récursifs de fonctions en toute sécurité. En effet, prenez l'exemple si dessous :

```
.thumb
.syntax unified
.text
debut :
  mov r0,#15     @ on met 15 dans le 1er argument
  mov r1,#3      @ on met 3 dans le 2e argument
  bl puiss       @ on veut 15^3
  mov r2,r0      @ on stocke le résultat dans r2
  B fin          @ on va à la fin du programme

puiss:           @ procédure récursive de calcul de puissance
  cmp r1,#0      @ test de la valeur de la puissance
  BNE else       @ si la puissance ne vaut pas zéro
  mov r0,#1      @ si la puissance vaut zéro on init la valeur
  B sortie       @ de retour (r0) à 1 et on s'en va

else:            @ sinon
  SUB r1,r1,#1   @ on décrémente la puissance
  bl puiss       @ on relance avec la puissance moins n-1
  mul r0,r0,r0   @ on calcul à la puissance n
sortie:
  mov pc,lr      @ on sort de la fonction

fin:
  nop
```

Dans ce code où l'on utilise le calcul de puissance par procédure récursive se pose le problème suivant :

4. En fait, les adresses d'implantation suivent des lois plus complexes pour permettre l'optimisation de la gestion mémoire. La règle imposée dans le sujet permet de faciliter la mise en œuvre. Vous aurez le loisir d'en apprendre plus sur l'occupation mémoire d'un programme en cours d'OS.

5. Hé oui, c'est le malloc qui décide où l'on peut réserver de la mémoire sur la machine hôte, on ne peut donc pas lui imposer de trouver de la place en 0x1000 (zone à laquelle vous n'avez de toutes façons pas accès)

— l'appel à BL stocke automatiquement l'adresse de retour dans le registre `lr`. Or si plusieurs BL se suivent (comme dans l'exemple), l'adresse originale de retour est écrasée (et donc perdue).

Par ailleurs, certaines fonctions imbriquées peuvent utiliser les mêmes registres temporaires et donc écraser des valeurs entres les appels !

Pour résoudre ces problèmes, on utilise couramment un espace mémoire pour stocker les variables intermédiaires entre deux appels afin de pouvoir les rétablir lorsque l'on retourne dans la fonction appelante. Cet espace, c'est la pile. Typiquement, on sauve la valeur d'adresse de retour, les paramètres ainsi que les variables locales (i.e., les registres utilisés par la fonction). On sauve ces informations en les 'empilant' dans l'espace mémoire de la pile avant l'appel d'une fonction puis en les 'dépilant' au retour de celle-ci.

Sur la plupart des architectures, la convention est d'initialiser la pile en 'haut' du segment qui lui est attribué. Ainsi, si la pile occupe l'espace `[0x0f7ff000,0x0ffff000[` la pile devra donc être initialisée à `0x0fffeffc`. En effet, la pile est utilisée pour stocker de l'information de la taille d'un mot, soit 4 octets.

5.2 Description des commandes de l'interpréteur

Dans cette section, on donne la liste et les spécifications des commandes de l'interface utilisateur de l'émulateur.

L'interface utilisateur à réaliser est un interpréteur de commandes (comme le Shell Linux, si ce n'est que ce ne sont pas des commandes Shell qui sont interprétées). Le fonctionnement de cet interpréteur est le suivant, dans une boucle infinie :

- Le programme se met en attente d'une commande ;
- L'utilisateur tape une commande et termine par "Entrée" $\hookleftarrow$;
- Le programme décode la commande et l'exécute, puis se met en attente de la commande suivante.

Toute les commandes écrivant un résultat utiliseront la sortie standard du programme (`stdout`). À l'inverse, vous prendrez garde à ce que toutes les messages de débogage qu'écrit votre programme soient envoyés sur le flux l'erreur standard du programme (`stderr`). Cela est nécessaire pour pouvoir automatiser les tests de votre programme.

On s'attachera à ce que le programme contrôle le bon format des paramètres des commandes. En particulier, et à titre d'exemple, les adresses mémoires passées en paramètres doivent être des entiers compris entre 0 et la valeur maximale du programme. Ainsi, pour toutes les commandes faisant intervenir un accès en mémoire, tout dépassement doit être signalé par un message d'erreur et la main doit être rendue à l'utilisateur. De même, on contrôlera le bon format des paramètres relatifs aux registres.

Concernant la syntaxe des paramètres, il est à noter les conventions suivantes :

- Les crochets { } indiquent que l'argument (ou le groupe d'arguments) à l'intérieur est optionnel.
- L'étoile $*$ collée à un argument indique que cet argument peut être répété un nombre quelconque de fois.
- Le symbole $+$ quand à lui signifie que l'argument doit être donné au moins une fois.
- Le caractère $\|$ séparant des arguments indique que l'un des arguments au choix peut être l'argument de la commande.
- Un argument entre les caractères $<$ et $>$ indique que la description de cet argument est donnée dans les lignes suivantes.

La grammaire en BNF de l'interpréteur est donnée en Annexe B. Veuillez vous y référer pour la mise en œuvre de l'interpréteur.

5.2.1 Commandes relatives à la gestion de l'environnement de l'émulateur

Charger un programme

- Nom de la commande : load
- Syntaxe : `load <nom_du_fichier> {<adresse>}`
- Paramètres :

  `nom_du_fichier` : chemin du fichier objet à charger.
  `adresse` : une valeur hexadécimale non signée sur 32 bits représentant l'adresse d'implantation du premier segment mémoire. Cette valeur est arrondie au premier multiple de 1ko supérieur.

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Le fichier dont le nom est passé en paramètre doit être un fichier ELF relogeable. Les fichiers ELF et la notion de relocation sont décrits en détail à la section 4. Si le fichier ELF contient des sections de relocations `.rel.text` et/ou `.rel.data`, la relocation sera gérée automatiquement à la fin du chargement en mettant à jour les adresses relatives dans le codage des instructions et données.

Quitter le programme

- Nom de la commande : exit
- Syntaxe : `exit`
- Description : Cette commande provoque la sortie de la boucle infinie et la fin du programme.

Afficher la mémoire, les registres

- Nom de la commande : disp (display)
- Syntaxe :

```
affiche les données mémoire  :  disp mem <plage>+
affiche la carte mémoire     :  disp mem map
affiche des registres        :  disp reg <registre>+
```

- Paramètres :

  `plage` : deux entiers non-signés séparé par un " :"
  `registre` : une chaîne parmi {`r0 ...r15, sp, lr, pc, apsr, all`} où 'all' signifie tous les registres.

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description :
  Cette commande affiche sur la sortie standard le contenu de la mémoire dont les adresses sont spécifiées en paramètres, la carte mémoire du programme ou les valeurs des registres.
- Format de sortie
  L'affichage des données mémoire se fera par ligne. Chaque ligne doit débuter par l'adresse virtuelle affichée en hexadécimale sur 4 octets suivie de l'affichage de 16 octets maximum en hexadécimal. Chaque octet sera séparé d'un espace.
  L'affichage des registres se fera à raison de 4 registres par ligne. Chaque valeur de registre affichée en hexadécimal sur 4 octets sera précédée de son mnémonique. Les valeurs seront séparées par une tabulation.

L'affichage de la carte mémoire se fera suivant l'exemple ci-dessous. Chaque ligne donnera les caractéristiques d'un segment comprenant : le nom du segment, les permissions associées, son adresse implantation virtuelle (en hexadécimale sur 4 octets) et sa taille (en octet).

- Exemple :
  Affichage de la mémoire de la zone `0x0000bee0` à `0x0000beef`.

```
>> disp mem 0xbee0:0xbeef
0x0000bee0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

  Affichage de registres spécifiques

```
>> disp reg r1 r4 r12 r13 apsr
   r0 :  00000000   r1 :  00000000  r12 :  00000000   sp :  00000000
 apsr :  00000000
```

  Affichage de tous les registres

```
>> disp reg all
   r0 :  00000000   r1 :  00000000   r2 :  00000000   r3 :  00000000
   r4 :  00000000   r5 :  00000000   r6 :  00000000   r7 :  00000000
   r8 :  00000000   r9 :  00000000  r10 :  00000000  r11 :  00000000
  r12 :  00000000   sp :  00000000   lr :  00000000   pc :  00000000
 apsr :  00000000
```

  Affichage du map mémoire.

```
>> disp mem map
Virtual memory map (8 segments)
.text       r-x   Vaddr: 0x00005000   Size: 12 bytes
.data       rw-   Vaddr: 0x00006000   Size: 8 bytes
.bss        rw-   Vaddr: 0x00007000   Size: 0 bytes
[stack]     rw-   Vaddr: 0x0f7ff000   Size: 8388608 bytes
```

Afficher le code assembleur

- Nom de la commande : disasm (disassemble)
- Syntaxe : `disasm <plage>+`
- Paramètres :

  `plage` : <adresse> :<adresse> (deux entiers non-signés séparé par un " :")
  `plage` : <adresse>+<décalage non-signé> (deux entiers non-signés séparé par un "+")

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : La commande affichera une instruction par ligne avec comme premier élément l'adresse virtuelle de l'instruction `adresse` en hexadécimal puis le code binaire de l'instruction en hexadécimal puis le code assembleur en chaîne de caractères. Si la zone d'adresse couvre une zone plus grande que le segment `.text`, les zones mémoire en dehors du segment `.text` ne sont pas affichées. Si une des adresses se trouve au milieu d'une instruction, cette derniere n'est pas affichée. Dans la mesure du possible, les instructions, les sauts ou les branchements à une adresse repérée par une étiquette devront afficher l'étiquette correspondante.
- Exemple :
  désassemblage de 2 instructions à partir de l'adresse `0x5000`

```
>> disasm 0x5000+8
```

```
5000 :: 2900       cmp r1, #0
5002 :: d102       bne.n 1a <else>
5004 :: f04f 0001  mov.w r0, #1
```
Désassemblage en dehors de la zone `.text` (qui commence ici en 0x5000). Notez que l'affichaga s'arrete en 0x5009 et non 0x500C car l'instruction suivante étant sur 4 octet elle dépasse la limite memoire de 0x500C.
```
>> disasm 0x4FF8:0x500c
5000 :: 2900       cmp r1, #0
5002 :: d102       bne.n 1a <else>
5004 :: f04f 0001  mov.w r0, #1
5008 :: e005       b.n 26 <sortie>
```

Modifier une valeur en mémoire ou un registre

- Nom de la commande : set
  ```
  modifie la mémoire   :  set mem <type> <adresse> <valeur>
  modifie un registre  :  set reg <registre> <valeur>
  ```
- Paramètres :

  `type` : `byte` ou `word` (i.e., 8 bits ou 32 bits)
  `adresse` : une valeur hexadécimale non signée sur 32 bits
  `registre` : une chaîne parmi {`$0 ...$31, $zero ...$ra, hi, lo, pc`}.
  `valeur` : une valeur entière

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Cette commande écrit la quantité `valeur` dans la mémoire à l'adresse `adresse` ou dans le registre `registre`. Elle n'affiche rien sur la sortie standard.
- Exemple :
  Affecte la valeur 0xAAFFBBDD à 0x5000.
  ```
  >> set mem word 0x5000 0xAAFFBBDD
  ```
  Affecte -15 au registre r1.
  ```
  >> set reg r1 -15
  ```

5.2.2 Commandes relatives au test du programme en cours

Évaluer les valeurs en mémoire ou dans les registres

- Nom de la commande : assert
- Syntaxe :
  ```
  test la valeur d'un registre          :  assert reg <registre> <valeur>
  test la valeur d'un mot en mémoire    :  assert word <adresse> <valeur>
  test la valeur d'un octet en mémoire  :  assert byte <adresse> <valeur>
  ```
- Paramètres :

  `adresse` : valeur entière non signée hexadécimale
  `valeur` : valeur entière signée

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : La commande `assert` vérifie les valeurs de mémoire et registre de l'émulateur. Cette commande est surtout utile pour automatiser les tests du programme assembleur.

- Exemple :
  Teste si le registre r0 contient 0xFFFFFFFE
  `>> assert word r0 -2`
  Teste si l'octet à l'adresse 0x500c contient 12
  `>> assert byte 0x500c 0xC`

Interrompre l'exécution d'un script

- Nom de la commande : debug
- Syntaxe : debug
- Valeur de retour : néant
- Description : Lorsqu'un script est exécuté, si la commande `debug` est rencontrée, l'exécution s'interrompt et la main est rendue à l'utilisateur qui peut interagir avec l'interpréteur.

Reprendre l'exécution d'un script

- Nom de la commande : resume
- Syntaxe : resume
- Valeur de retour : néant
- Description : Si l'interpréteur est en mode DEBUG alors la commande `resume` permet de reprendre l'exécution du script.

5.2.3 Commandes relatives à l'exécution du programme

Exécuter à partir d'une adresse

- Nom de la commande : run
- Syntaxe : `run {<adresse>}`
- Paramètres :

  `adresse` : valeur entière non signée hexadécimale
- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Cette commande charge PC avec l'adresse fournie en paramètre et lance le microprocesseur. Si le paramètre est omis, l'exécution commencera à la valeur courante de PC.

Exécution pas à pas

- Nom de la commande : step

  `Exécute la prochaine instruction dans la procédure : step`
  `Exécute la prochaine instruction : step into`
- Description : Cette commande provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre PC puis rend la main à l'utilisateur. Dans le cas de `step`, si l'on rencontre un appel à une procédure, cette dernière s'exécute complètement jusqu'à l'instruction de retour incluse (par exemple `BX lr`). Par contre, dans le cas de `step into`, l'exécution s'interrompt juste après l'appel (i.e., dans la procédure appelée). La main est alors rendu à l'utilisateur sur l'instruction suivant l'appel.

Gérer les points d'arrêt

- Nom de la commande : break (breakpoint)

```
ajout de points d'arrêt        :  break add <adresse>+
suppression de points d'arrêt  :  break del <adresse>+|all
affichage des points d'arrêt   :  break list
```

- Paramètres :

  `adresse` : une valeur hexadécimale non signée sur 32 bits

- Valeur de retour : retourne 0 en cas de succès sinon une valeur non nulle est retournée.
- Description : Cette commande met un point d'arrêt à (aux) l'adresse(s) fournie(s) en paramètre. Lorsque le registre PC sera égal à cette valeur, l'exécution sera interrompue avant l'exécution de l'instruction et l'utilisateur reprendra la main. Que ce soit pour l'ajout ou la suppression, break n'affiche rien sur la sortie standard. Si un point déjà présent est ajouté ou si un point non présent est supprimé, la commande est sans effet. Si une adresse est non atteignable (i.e., en dehors du segment `.text`), un message d'erreur est affiché. La commande `break` doit également permettre d'afficher l'ensemble des points d'arrêts enregistrés.
- Exemple :
  On ajoute deux points d'arrêt
  `>> break add 0x5000 0x5010`
  Affichage des points d'arrêt
  `>> break list`
  `#0 0x00005000`
  `#1 0x00005010`
  Suppression de points d'arrêt et affichage
  `>> break del 0x00006000`
  `>> break del 0x5000`
  `>> break list`
  `#0 0x00005010`

Chapitre 6

À propos de la mise en œuvre

6.1 Méthode de développement

La conduite d'un projet de programmation de taille "importante" dans un contexte de travail en équipe (qui n'offre pas que des avantages !) et multitâches (autres cours en parallèles) nécessite une méthodologie qui vous permettra de parvenir efficacement à un code qui, par exemple : répond au problème, soit robuste (absence de plantage), fiable (absence d'erreur, ou gestion appropriée des erreurs), clair et lisible, maintenable (facilité de reprise et d'évolution du code), etc.

Par ailleurs, le développement de programmes nécessite de nos jours de plus en plus de réactivité aux modifications de toutes sortes (p.ex. : demandes des clients, changement d'équipe, bugs, évolutions de technologie) ce qui a conduit à des méthodes de conception s'écartant des schémas classiques de conception/implémentation pour adopter un processus plus souple, plus facile à modifier en cours de développement.

La mise au point de méthodologies de développement est l'une des activités du "Génie Logiciel", une branche de l'informatique. Une telle méthodologie définit par exemple le cycle de vie qu'est censé suivre le logiciel, les rôles des intervenants, les documents intermédiaires à produire, etc.

À l'occasion du projet, vous expérimenterez non pas une méthodologie complète – ce serait trop lourd – mais deux méthodes, ou moyens méthodologiques, qui sont promus par plusieurs méthodologies : le *développement incrémental* et le *développement piloté par les tests*.

6.1.1 Notion de cycle de développement ; développement incrémental

La notion de cycle de vie du logiciel correspond à la façon dont le code est construit au fur et à mesure du temps, depuis la rédaction des spécifications jusqu'à la livraison du logiciel, en passant par l'analyse, l'implantation, les tests, etc. Plusieurs cycles de vie sont possibles.

Dans le cadre du projet, vous adopterez un *cycle incrémental* - appelé également *cycle en spirale*. Il s'agit de découper les fonctionnalités du logiciel de telle sorte qu'il soit construit progressivement, en plusieurs étapes. À l'issue de chaque étape, le logiciel ne couvre pas toutes les fonctionnalités attendues, mais doit être fonctionnel - et testable - pour la sous-partie des fonctionnalités réalisées jusqu'ici.

Les avantages du développement incrémental dans le cadre du projet sont nombreux. Par exemple, les risques sont réduits : au lieu de ne pouvoir tester votre logiciel qu'à la fin du projet et de risquer que rien ne marche, vous aurez à chaque étape quelque chose de partiel, certes, mais qui fonctionne. Incidemment, cela tend à garantir une plus grande implication dans l'activité de développement, puisqu'on voit la chose - le logiciel - se construire au fur et à mesure.

À l'inverse, le développement incrémental peut nécessiter une plus grande dextérité : durant un incrément, il peut être nécessaire de développer une "fausse" partie du logiciel pour "faire comme si" cette partie existait... Puis être capable de la remplacer intégralement lors de l'incrément suivant. Une autre des difficultés du développement incrémental tient à la définition des incréments et de l'ordre de leur réalisation – c'est-à-dire du découpage temporel de l'activité de développement. Fort heureusement pour vous, le découpage est déjà défini pour le projet !

Vous réaliserez votre projet en quatre incréments. Chaque incrément vous occupera de une à trois semaines (typiquement : une séance de TD, une séance de TP et une séance en temps libre).

Par ailleurs, on ne résiste pas à vous donner les conseils/rappels suivants inspirés de la programmation structurée et incrémentale :

— Séparer le projet en modules indépendants de petites tailles en fonction des fonctionnalités désirées du programme.
— Choisir des solutions simples pour les réaliser (ce qui ne veut pas dire les SEULES solutions que vous connaissez).
— Concevoir les tests des modules AVANT leur écriture (concevoir les tests avant permet de bien réfléchir sur le comportement attendu).
— Intégrer la génération de traces pour faciliter le débogage.
— Commenter le code pendant l'écriture du code (après, c'est trop tard).
— Bien définir les rôles de chaque membre de l'équipe (p.ex. : écriture des tests, des structures de données, des rapports, etc.).
— Discuter du projet avant chaque phase de travail.
— ! Se mettre d'accord sur les standards de programmation [1] ! (p.ex. : organisation des dossiers, include, makefile, éditeurs, commentaires, nom des variables, etc.)
— ...

À toute fin utile vous pouvez consulter le site de la communauté de l'*eXtreme Programming* [2] qui fourmille de conseils intéressants.

6.1.2 Développement piloté par les tests

Pour développer un logiciel, il est possible de travailler très précisément ses spécifications, de s'en imprégner, de beaucoup réfléchir, de développer... Et de ne tester le logiciel qu'à la fin, lorsqu'il est fini. Cette démarche est parfois appropriée mais, dans le cadre du projet, vous renversez le processus pour vous appuyer sur la méthode dite de "développement piloté par les tests".

Pour chaque incrément, vous commencerez donc par écrire un jeu de tests avant même d'écrire la première ligne de code. Le jeu de test sera constitué de fichiers de script qui seront :

— des fichiers textes **.sml** comprenant le code des commandes de l'émulateur à exécuter (par exemple : charger un fichier elf, lancer le programme, vérifier une valeur en mémoire) ainsi que de variables qui, en particulier, indique si le test est censé échouer ou réussir.

Ce n'est qu'après avoir écrit votre jeu de tests que vous réfléchirez à la structure de votre programme et que vous le développerez.

Au début de l'incrément, aucun des tests du jeu de tests ne doit "passer". À l'issue de l'incrément, tous les tests que vous avez écrits doivent "passer" : votre programme doit produire le résultat attendu pour chacun des tests.

Voici quelques-uns des avantages du *développement piloté par les tests*.

— Cette méthode garantit que le programme sera accompagné d'un jeu de tests - alors que, trop souvent, les développeurs "oublient" d'écrire des tests.
— écrire les jeux de tests est un très bon moyen pour assimiler les spécifications du programme et s'assurer qu'on les a comprises. Le jeu de tests doit faire apparaître les situations simples, mais aussi faire ressortir les cas particuliers et plus complexes. Ce faisant, le développement lui même peut être guidé par une vue d'ensemble de ce que doit faire le programme, dans laquelle, dès le début, on a en tête non seulement les cas simples, mais aussi les situations plus délicates.

1. https://computing.llnl.gov/linux/slurm/coding_style.pdf
2. http://www.extremeprogramming.org

— L'existence d'un jeu de tests permet de très vite détecter les *problèmes de régression*, c'est à dire les situations ou, du fait de modifications introduites dans le code, une partie du programme qui fonctionnait jusqu'ici se met à poser problème. Or, la méthode incrémentale que vous adopterez tend à ce que le code soit souvent modifié pour tenir compte des nécessités d'un nouvel incrément...et donc à ce que des problèmes de régression surgissent.

Plus concrètement, vous vous astreindrez à faire tourner votre programme très souvent – le plus souvent possible, en fait – sur l'ensemble de vos tests ; cela vous permettra de détecter rapidement les problèmes et de vous assurer que votre programme, petit à petit, se construit dans la bonne direction.

6.1.3 À propos de l'écriture des jeux de tests

L'écriture d'un jeu de tests pertinent – c'est-à-dire à même de raisonnablement "prouver" que votre logiciel fait ce qu'il doit faire – n'est pas chose évidente.

Voici quelques conseils pour l'écriture des jeux de tests.

1. un bon jeu de tests comprend nécessairement de nombreux tests - par exemple entre 20 et 100 pour chaque incrément.
2. commencer par écrire des tests très courts et très simples. Par exemple, pour le premier incrément, lancer une commande inconnue et vérifier que le programme renvoie bien une erreur du bon type. La simplicité de ces tests facilitera la recherche du problème si un test échoue.
3. penser également à écrire des tests complexes : les problèmes peuvent surgir du fait de l'enchaînement d'instructions !
4. se remuer les méninges face aux spécifications ; essayer d'imaginer les cas qui risquent de poser problème à votre programme ; pour chacun d'eux, écrire un test !
5. **important** : un jeu de tests se doit de tester ce qui doit *fonctionner*... mais aussi ce qui doit *échouer* et la façon dont votre programme gère les erreurs ! Pour ce faire, un jeu de tests doit *aussi* inclure des tests qui font échouer le programme. Par exemple, un premier test pourrait être de charger un elf corrompu afin de s'assurer que votre programme renvoie une erreur.

6.1.4 Organisation interne d'un incrément

On a déjà indiqué que la réalisation de chacun des quatre incréments commencera par l'écriture du jeu de tests, et devra se terminera par un test du programme dans lequel tous les tests du jeu de tests "passent". Mais comment s'organiser à l'intérieur de l'incrément ? Voici quelques conseils...

1. réfléchir à la structure de votre programme. Définir en commun les principales structures de données et les signatures et "contrats" (ou "rôle") des principales fonctions.
2. Autant que faire se peut, essayer d'organiser l'incrément... de façon incrémentale ! La notion de développement incrémental peut en effet s'appliquer récursivement : à l'intérieur d'un incrément, il est souhaitable de définir des étapes ou "sous-incréments" qui vous assurent que votre programme est "partiellement fonctionnel" le plus souvent possible.
3. Réfléchir à la répartition du travail. En particulier, éviter que la répartition ne bloque l'un d'entre vous si l'autre prend du retard.
4. Compiler très régulièrement. Il est particulièrement fâcheux de coder pendant plusieurs heures pour ne s'apercevoir qu'à la fin que la compilation génère des centaines d'erreurs de syntaxe. Un développeur expérimenté fait tourner le compilateur (presque) tout le temps en tâche de fond !

5. Exécuter souvent le jeu de tests, afin de mesurer l'avancement et de faire sortir le plus tôt possible les éventuelles erreurs.

6.2 Notions d'interpréteur, de syntaxe et de machine à états finis

Nous décrivons ici brièvement la manière dont l'utilisateur de l'émulateur va pouvoir interagir avec le programme simulé.

6.2.1 Interpréteur

Lorsque vous entrez des commandes dans un terminal, vous utilisez en réalité son interpréteur : en effet, il faut tout d'abord que le terminal ait une idée de ce que vous voulez lui faire faire en *interprétant* la chaîne de caractères entrée par l'utilisateur. Nous prendrons un exemple simple : imaginez que vous entriez la commande `"ls -l *.c"` (destinée, donc, à lister tous les fichiers de code source C présents dans le répertoire courant). L'interpréteur va tout d'abord découper la chaîne de caractères en trois chaînes : `"ls"`, `"-l"` et `"*.c"`. Chacune de ces chaînes sera appelée par la suite un *token*[3]. L'interpréteur va ensuite identifier le *type* du premier *token*. Dans notre cas, il va déterminer qu'il s'agit d'un exécutable à lancer. Le terminal va ensuite lancer cette commande en lui passant en paramètres les deux autres chaînes `"-l"` et `"*.c"`. À charge ensuite à l'exécutable `ls` de remplir sa fonction, en interprétant lui-même les paramètres que l'interpréteur du terminal lui aura transmis. La section 6.2.5 donne des indices pour réaliser l'extraction des tokens en langage C.

Le mécanisme de base d'un interpréteur est simple : c'est un bout de code qui passe son temps à afficher une *invite de commande*[4], à attendre que l'utilisateur daigne entrer une ligne de commande, la lire, puis, donc l'interpréter et, si elle est valide, l'exécuter. Et on recommence ainsi indéfiniment – mais on prévoit généralement une commande `exit` pour sortir de cette boucle infinie !

Nous utiliserons un interpréteur de commandes pour guider l'interaction de l'utilisateur avec l'émulateur. Cet interpréteur devra accepter une dizaine de commandes. Mais avant de vous les présenter, il nous faut d'abord vous expliquer comment *décrire* une commande.

6.2.2 Forme de Backus-Naur (BNF)

Une commande ne va accepter que certains *tokens* : des options ou des paramètres clairement définis, et dont l'enchaînement est lui aussi défini précisément. L'ensemble des enchaînements de *tokens acceptés* par une commande est appelé sa *syntaxe*.

Il serait vain de vouloir lister exhaustivement tous les enchaînements, potentiellement en nombre infini, acceptés par une syntaxe. On va plutôt chercher à *décrire* les enchaînements acceptés par la syntaxe d'une commande. Cette description d'une syntaxe s'effectue en utilisant la forme de Backus-Naur (BNF étant l'acronyme utilisé, depuis l'anglais). En conséquence, le langage BNF, qui sert à décrire une syntaxe et donc un autre langage, est parfois qualifié de méta-langage.

Le langage BNF est composé de différents éléments : les méta-symboles (qui sont des symboles propres au langage BNF), les *terminaux* (ce que l'utilisateur écrit) et les *non-terminaux* (les catégories du langage que l'on est en train de décrire). Ces éléments sont composés entre eux pour décrire des *règles de production* (qui décrivent comment les non-terminaux sont formés). L'ensemble des règles de production forme la syntaxe du langage que l'on est en train de décrire avec le langage BNF.

Les méta-symboles du langage BNF sont :

3. En bon français, un jeton.
4. Typiquement, dans un terminal : `bash-3.2$`. En anglais, on parle de *prompt*.

— : :=, qui signifie “est défini par” ;
— < et >, qui englobent un non-terminal ;
— . . ., qui signifie “jusqu’à” ;
— |, qui signifie “ou” ;
— ’, dont une paire englobe un terminal ;
— +, qui signifie “un ou plus” ;
— *, qui signifie “zéro ou plus”.

Un règle de production consiste à définir un non-terminal à l’aide de terminaux ou d’autres non-terminaux.

Par exemple, imaginons le langage suivant pour un interpréteur, composé de seulement deux commandes :

— `exit`, qui ne prend pas d’argument ;
— `show`, qui prend en argument un ou plusieurs intervalles d’adresses.

Les terminaux seront les chaînes de caractères `exit` et `show`, le marqueur hexadécimal (’0x’), les chiffres hexadécimaux, et le séparateur d’adresses (’-’). Les non-terminaux seront les commandes et les intervalles d’adresses, ainsi que les adresses exprimées en hexadécimal. En effet, nous définissons un intervalle d’adresses comme étant composé d’une adresse, suivie d’un séparateur d’adresses, et d’une seconde adresse. D’autre part, nous définissons une adresse comme étant composée du marqueur hexadécimal suivi d’un ou plusieurs chiffres hexadécimaux. La syntaxe BNF correspondant à cet exemple est donné Fig. 6.1. Notre petite syntaxe contient donc six règles de production, dont la première n’a pour objet que de lister les commandes permises.

```
<wait>  ::= "exit" epsilon
            "show" <inter>+ epsilon
            epsilon

<inter> ::= <blank>+ <adr1> ’-’ <adr2>

<adr1>  ::= 0x<hex>+

<adr2>  ::= 0x<hex>+

<hex>   ::= ’0’...’9’ | ’a’...’f’ | ’A’...’F’

<blank> ::= ’ ’|’\t’
```

Figure 6.1 – Syntaxe BNF de notre petit exemple.

Le langage BNF est un outil fondamental en théorie des langages. Il nous permet de dire que, par exemple, les commandes de la session d’exemple représentée dans la Fig. 6.2 sont valides et doivent être exécutées par l’interpréteur.

Pour aussi puissant que puisse être cet outil, il reste purement théorique et descriptif, et ne dit rien de la manière d’implanter effectivement une syntaxe en machine. Pour cela, nous devons utiliser un automate. Il existe plusieurs sortes d’automates en informatique, mais celui qui suffira amplement pour implanter le genre de syntaxe dont nous aurons besoin est appelé une machine à états finis. Il n’en reste pas moins que nous vous donnons la syntaxe complète de l’interpréteur sous forme de syntaxe BNF en annexe B, et qu’il vous appartiendra de construire et d’implanter l’automate correspondant.

```
interp $ show 0xdeadbeef-0xffffffff
interp $ show      0xdeadbeef-0xffffffff 0x0-0xbeef
interp $ show 0xdeadbeef-0xffffffff    0x0-0xbeef 0x1000-0x100f
interp $ exit
Exiting.
```

Figure 6.2 – Exemple de session d'un petit interpréteur. L'invite de commande est `interp $`. Les chaînes de caractères qui suivent l'invite de commande sont les commandes entrées par l'utilisateur. Les lignes 2 et 3 montrent comment passer plusieurs intervalles à la commande `show`, qui est effectivement censée pouvoir en accepter un ou plus (*cf.* l'utilisation du méta-symbole '+' dans la syntaxe BNF de la Fig. 6.1).

6.2.3 Machine à états finis (FSM)

Une machine à états finis (ou FSM pour *finite state machine* en anglais) décrit les états dans lesquels un automate est autorisé à être. Une FSM décrit aussi quoi faire lorsque l'on atteint un état donné, ou même ce qu'îl faut faire lorsque l'on passe de tel à tel autre état. Nous allons donc construire un automate (une FSM) pour reconnaître si une entrée de l'utilisateur dans l'interpréteur correspond à quelque-chose de valide du point de vue de la syntaxe. Dans un premier temps, nous allons décrire ce qu'est une machine à états finis, et dans un second temps nous vous donnerons quelques pistes pour implanter efficacement et facilement une FSM en C [5].

Description d'une FSM

Une machine à états finis est définie par la donnée de ses états, et des transitions valides entre ses états. Lorsque, comme nous nous y emploierons, nous souhaiterons utiliser une FSM pour implanter une syntaxe, nous aurons besoin d'attacher à la description des transitions valides le motif ayant causé cette transition entre deux états. Les motifs seront les terminaux ou les non-terminaux de la syntaxe. Les états de notre FSM comprendront les non-terminaux de notre syntaxe, mais pas seulement.

Il faut bien comprendre qu'une syntaxe BNF représente en quelque sorte une description *positive* d'un langage. Elle ne dit rien *explicitement* des erreurs de syntaxe, elle ne fait finalement que les sous-entendre. Notre FSM, en tant qu'elle sera chargée de l'*implantation* de notre syntaxe BNF, représentera un dispositif opérationnel qui devra, à ce titre, être capable d'indiquer à l'utilisateur s'il a commis une erreur de syntaxe et, le cas échéant, laquelle. Il conviendra donc d'ajouter à notre FSM un état correspondant aux erreurs de syntaxe. Une erreur de syntaxe, *in fine*, sera comprise comme une transition invalide depuis un état qui n"autorise pas cette transition. Pour chaque état, nous désignerons par le même nom de *défaut* un motif invalide, qui provoque donc une transition vers l'état d'erreur.

Il est donc possible qu'une entrée de l'utilisateur produise une erreur de syntaxe. Mais, et puisque personne n'est à l'abri d'un coup de chance, il n'est pas impossible non plus que cette entrée soit valide et doive être effectivement exécutée par l'interpréteur. Si vous observez bien la Fig. 6.1, et que vous vous souvenez que les états de notre FSM vont inclure les non-terminaux de notre syntaxe BNF, il convient de s'interroger sur la signification de l'état correspondant au non-terminal <*wait*>. Cet état est en réalité celui d'une *attente* d'une entrée de la part de l'utilisateur. Et lorsqu'aucun motif n'est attaché à une transition, c'est donc qu'il s'agit d'une transition inconditionnelle.

Pour l'exemple, la FSM correspondante est représentée dans la Fig. 6.3.

5. Tellement facilement d'ailleurs, qu'il ne serait pas impossible que vous n'ayez pas besoin de spécifier votre FSM outre mesure, ou que vous soyez en mesure d'écrire son code au fil de l'eau !

6.2.4 Implantation d'une FSM en C

Pour illustrer la mise en œuvre d'un automate à états finis, prenons l'exemple d'un programme qui accepte en entrée une chaîne de caractères et qui doit déterminer si cette chaîne est un entier décimal, octal ou hexadécimal. La première phase du logiciel est de vérifier que la ligne donnée en entrée contient uniquement des éléments acceptés par le langage et de les identifier. Par exemple, il faut pouvoir reconnaître que la chaîne de caractères `0123` est une valeur octale et que `0x123` est une valeur hexadécimale. Un moyen brutal serait de comparer les caractères du fichier avec toutes les chaînes possibles du langage (avec par exemple `strcmp`). Ceci est bien entendu impossible car : les possibilités sont trop importantes, les tailles des chaînes peuvent varier, il est nécessaire de bien identifier le début et la fin des chaînes d'intérêt pour éviter les recouvrements (p.ex. : trouver `.text` dans le commentaire `# la section .text`)...

Heureusement, le langage est complètement déterministe et il est possible de connaître la composition de chaque élément terminal du langage. Par exemple, on sait qu'une valeur hexadécimale est toujours préfixée par `0x` puis un certain nombre de caractères $\in$ `[0,9]` $\cup$ `[a,b,c,d,e,f]` alors qu'une valeur octale n'est composée que de chiffres inférieurs à 8. En tournant les choses de cette façon le problème devient de trouver des *motifs* de caractères dans le texte et non plus des mots prédéfinis. Un autre problème vient du fait que les motifs peuvent partager des caractéristiques communes qui impliquent qu'il faut avoir lu un certain nombre de caractères avant de reconnaître un motif (p.ex. : tant que l'on a pas lu le 'x' après un zéro on ne sait pas si on lit un nombre hexadécimal ou octal).

Une façon d'aborder le problème est de représenter les motifs et leur parcours par un automate à états. Succinctement, l'automate est composé d'états qui dans notre cas représentent la catégorie courante de la chaîne de caractères lue et de transitions entre états étiquetés par les caractères que l'on va lire. L'exemple de la figure 6.4 montre comment on peut représenter un automate faisant la différence entre nombres décimaux, octaux ou hexadécimaux.

Cet automate lit les caractères un par un jusqu'à arriver à l'état terminal ou erreur. Ainsi, en prenant l'exemple de la chaîne `0567`, l'automate passe successivement par `INIT`, `pref hexa`, et reste dans `octal` jusqu'à la fin donnant ainsi la catégorie de la chaîne. La figure 6.5 donne une traduction en langage C de l'automate (il existe bien d'autre moyens de traduire l'automate en C).

Dans cette traduction, le fichier est parcouru caractère par caractère (boucle `while`). L'automate est tout d'abord dans l'état `INIT`. La lecture de chaque caractère peut provoquer une transition vers un autre état (par exemple si `c` est un chiffre), laisser l'automate dans l'état présent (p.ex., si `c` est un saut de ligne), faire terminer la tâche (p.ex., `EOF` *End Of File*) ou encore détecter une erreur. Ce programme est capable de traiter de gros fichiers textes en peu de temps.

Dans le cadre du projet, l'automate général sera bien plus conséquent et il sera plus judicieux de travailler les chaînes au niveau des tokens. La démarche est d'étudier les différents éléments du langage, d'identifier leur motifs, de construire l'automate, et de prévoir les fichiers de tests, avant de commencer à coder...Il est possible que certains traitements à effectuer dans certains états soient réutilisables, n'hésitez donc pas à fragmenter votre code en fonctions.

6.2.5 Découper une chaîne de caractères en token, `strtok`

Un interpréteur lit généralement les commandes ligne par ligne. C'est à dire que chaque ligne est interprétée comme contenant une (et une seule) commande complète [6] autrement dit, le caractère de retour à la ligne `'\n'` marque la fin d'une commande. Si l'on considère que chaque *token* est séparé les

6. Dans la réalité une ligne peut contenir plusieurs commandes (par exemple, `"ls -l; cd .."` liste le contenu d'un répertoire puis va dans le répertoire parent) et dans certains interpréteurs une commande peut occuper plusieurs lignes.

uns des autres par des caractères particulier (dans notre cas des blancs — espace, tabulation, retour de ligne...) alors le langage C fourni un mécanisme implanté dans la la fonction standard `strtok` qui permet de parcourir la chaine de caractères token par token. L'extrait de code ci-dessous illustre son utilisation avec une chaîne de caractères.

La chaîne de caractères `"Dupond 20 \t 76"` est séparée en éléments délimités par les espaces et tabulations. Chaque appel à `strtok()` renvoie le prochain élément. Des informations complètes sont accessibles dans le `man` de `strtok`.

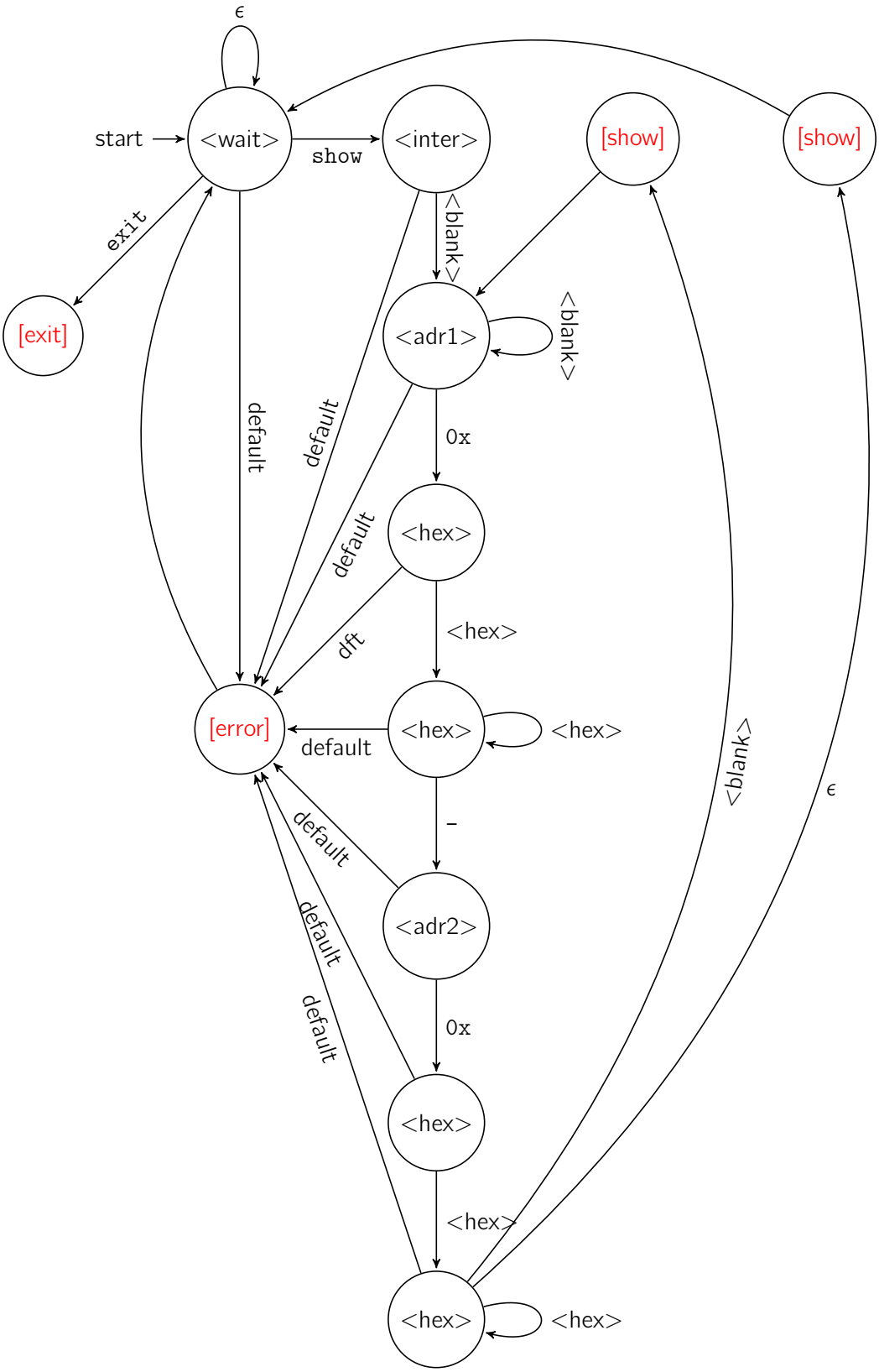


Figure 6.3 – La FSM complète correspondant à notre petit exemple. Pour des raisons évidentes de place manquante, elle est discutée dans le texte.

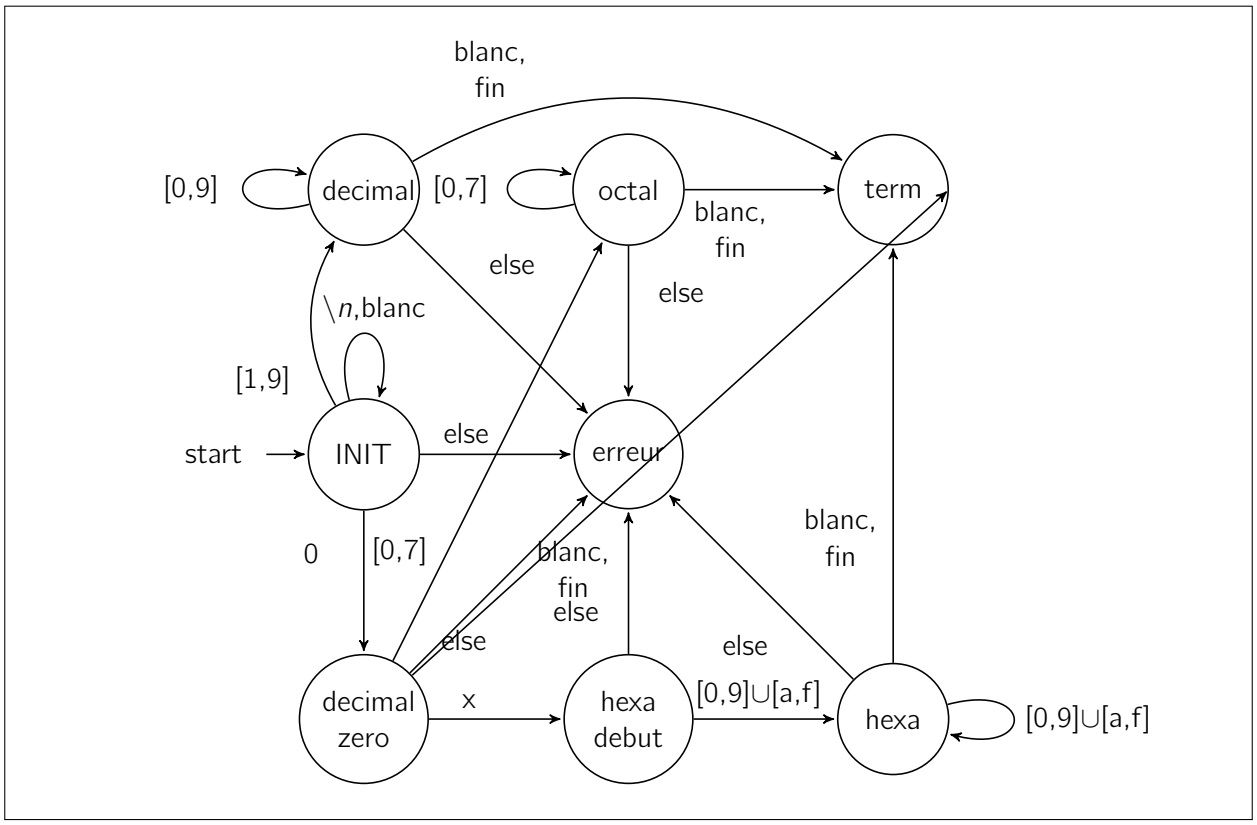


Figure 6.4 – Exemple d'automate faisant la différence entre une valeur décimale, octale et hexadécimale

```
/* definition des etats*/
enum {INIT, DECIMAL_ZERO, DEBUT_HEXA, HEXA, DECIMAL, OCTAL};
/* mise en oeuvre de l'automate*/
int main() {
    int c;              /*caractere analyse courant*/
    int S=INIT;         /*etat de l'automate*/
    FILE *pf;           /*pointeur du fichier à analyser*/

    if((pf=fopen("nombres.txt","rt"))==NULL) {
        perror("erreur d'ouverture fichier");return 1;}

    while(EOF!=(c=fgetc(pf))) {
        switch(S) {
        case INIT:
            i=0;
            if(isdigit(c)) {    /* si c'est un chiffre*/
                S = (c=='0')? DECIMAL_ZERO : DECIMAL;
            }
            else if (isspace(c)) S=INIT;
            else if (c==EOF) return 0; /* fin de fichier*/
            else return erreur_caractere(string,i,c);
            break;
        case DECIMAL_ZERO: /*reperage du prefixe de l'hexa*/
            if (c == 'x' || c == 'X') S=HEXA;
            else if (isdigit(c) && c<'8') S=OCTAL;/* c'est un octal*/
            else if (c==EOF || isspace(c)){ S=INIT;
                printf("la chaine est sous forme édcimale\n");
            }
            else return  erreur_caractere(string,i,c);
            break;
        case DEBUT_HEXA: /* il faut au moins un chiffre apres x*/
            if(isxdigit(c)) S=HEXA;
            else return erreur_caractere(string,i,c);
            break;
        case HEXA: /* tant que c'est un chiffre hexa*/
            if(isxdigit(c)) S=HEXA;
            else if(c==EOF || isspace(c)) { S=INIT;
                printf("la chaine est sous forme éhexadcimale\n");
            }
            else return erreur_caractere(string,i,c);
            break;
        case DECIMAL: /*tant que c'est un chiffre*/
            if(isdigit(c))  S=DECIMAL;
            else if(c==EOF || isspace(c)) {  S=INIT;
                printf("la chaine est sous forme édcimale\n");
            }
            else  return erreur_caractere(string,i,c);
            break;
        case OCTAL: /*tant que c'est un chiffre*/
            if(isdigit(c)&& c<'8') S=OCTAL;
            else if(c==EOF || isspace(c)) { S=INIT;
                printf("la chaine est sous forme octale\n");
            }
            else  return erreur_caractere(string,i,c);
            break;
        }
    }
    return 0;
}
```

Figure 6.5 – Exemple de traduction en C de l'automate de la figure 6.4

```
/* test strtok */
#include <string.h>/* prototype de la fonction strtok*/
#include <stdio.h>
typedef struct {char* nom; int age; int poids;} Personne;

void main(){
char *token;
char *texte = strdup("Dupond␣20␣\t␣76");
char *delimiteur = "␣";
Personne pers;

/*renvoie un pointeur vers "Dupond". */
printf("%s\n", pers.nom=strdup(strtok(texte, delimiteur)));

/* renvoie l'entier "20". */
printf("%d\n", pers.age=atoi(strtok(NULL, delimiteur)));

/* renvoie l'entier "76". */
printf("%d\n", pers.age=atoi(strtok(NULL, delimiteur)));
}
```

Figure 6.6 – Extrait de code illustrant l'usage de `strstok()`

Chapitre 7

Organisation du Projet

La page web du projet informatique est disponible à l'adresse suivante : http://tdinfo.phelma.grenoble-inp.fr/2Aproj/. Veuillez vous y référer pour tous ce qui concerne l'organisation et l'évaluation.

7.1 Objectif général :

À la fin de ce projet vous devrez avoir réalisé un programme capable d'émuler l'execution d'un fichier objet elf pour l'architecture *ARM Cortex-M*. L'interaction avec cet émulateur se fera à travers un interpreteur de commandes. L'émulateur sera appelé en tapant sous Linux la commande :

```
emul-arm
```

ou

```
emul-arm script_filename
```

où `script_filename` est le nom d'un fichier de commandes à exécuter.

7.2 Étapes de développement du programme

Le projet est découpé en 4 étapes principales que vous aurez à achever dans un délai imparti. Au début de chaque étape, une séance globale de tutorat sera utilisée pour la préparation puis quelques séances de codages seront consacrées à la mise en œuvre. Les quatre étapes seront :

1. L'émulateur et son interpréteur de commandes : À la fin de cette étape, vous devrez avoir l'environnement de l'émulateur initialisé et un interpréteur exécutant des commandes simples.
2. Désassemblage des instructions : À la fin de cette étape, vous devrez avoir un émulateur qui est capable de désassembler les instructions d'un fichier objet .
3. Simulation : À la fin de cette étape, vous devrez avoir un programme capable d'exécuter le code machine pas à pas ainsi qu'en utilisant des points d'arrêt.
4. Relocation et appels systèmes : À la fin de cette étape, votre émulateur devra être capable de charger et d'exécuter du code relogeable ainsi que de gérer des appels systemes pour certaines opérations d'entrées/Sorties.

Les détails de chacune de ces étapes sont à recueillir sur le site web du projet.

7.3 Bonus : extensions du programme

Plusieurs extensions possibles du programme peuvent être envisagées, telles que la prise en compte d'un plus grand nombre de relocations, le chargement de plusieurs fichiers objets, la prise en compte des instructions sur les *float*, etc. Toute extension menée de manière satisfaisante amènera un bonus dans la notation.

Bibliographie

[1] B. W. Kernighan et D. M. Ritchie *Le langage C, Norme ANSI* http://http://cm.bell-labs.com/cm/cs/cbook/

[2] Bernard Cassagne. *Introduction au langage C.* http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html

[3] *ARM v7-M Architecture Reference Manual, Revision Derrata 2010_Q3*. 2006-2010, ARM Limited. http://www.arm.com/products/processors/cortex-m

[4] *ELF for the ARM Architecture*. 2012, ARM Limited. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044e/IHI0044E_aaelf.pdf

[5] *Executable and Linkable Format (ELF)*. Tools Interface Standard (TIS). Portable Formats Specification, Ver 1.1. http://www.skyfree.org/linux/references/references.html

[6] Amblard P., Fernandez J.C., Lagnier F., Maraninchi F., Sicard P. et Waille P. *Architectures Logicielles et Matérielles.* Dunod, collection Siences Sup., 2000. ISBN 2 10 004893 7.

[7] Dean Elsner, Jay Fenlason and friends. *Using as, the* GNU *Assembler*. Free Software Foundation, January 1994. http://www.gnu.org/manual/gas-2.9.1/

[8] FSF. *GCC online documentation*. Free Software Foundation, January 1994. http://gcc.gnu.org/onlinedocs/

[9] Steve Chamberlain, Cygnus Support. *Using ld, the* GNU *Linker*. Free Software Foundation, January 1994. http://www.gnu.org/manual/ld-2.9.1/

[10] Linux Assembly http://linuxassembly.org/

[11] Linus Torvalds. *Linux Kernel Coding Style.* https://computing.llnl.gov/linux/slurm/coding_style.pdf

Annexe A

ELF : Executable and Linkable Format

Ce chapitre décrit "brièvement" le format ELF (*Executable and Linkable Format*) et explique comment en extraire les informations nécessaires pour le projet. Le format ELF est le format des fichiers objets dans la plupart des systèmes d'exploitation de type UNIX (GNU/Linux, Solaris, BSD, Androïd...). Il est conçu pour assurer une certaine portabilité entre différentes plates-formes. Il s'agit d'un format standard pouvant supporter l'évolution des architectures et des systèmes d'exploitation.

Le format ELF est manipulable, en lecture et écriture, par utilisation d'une bibliothèque C de fonctions d'accès `libelf`. Cette librairie suffit pour lire et écrire des fichiers ELF, même quand ELF n'est pas le format utilisé par le système d'exploitation (par exemple, on peut l'utiliser sous MacOs X).

Dans ce chapitre, nous nous limitons aux fichiers objets, dits *à lier*, fichiers contenant du *code binaire relogeable* (ou *translatable*), c'est-à-dire du code binaire dont l'affectation en mémoire n'est déterminée qu'au moment de l'exécution. Ce chapitre décrit tout d'abord la structure d'un fichier objet, puis s'intéresse à la notion de relocation.

A.1 Fichier objet au format ELF

Les fichiers ELF sont composés d'un ensemble de sections (éventuellement vides) comme indiqué par la figure A.1.

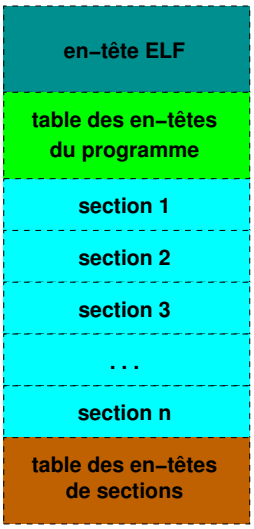


Figure A.1 – Structure d'un fichier ELF

:

— En-tête du fichier ELF
— Table des entêtes de sections
— Table des noms de sections ("".text"", "".data"", "".rel.text"",...)
— Table des chaînes (noms des symboles)
— Table des symboles (informations sur les symboles)

— Section de données (.text, .data, .bss)
— Tables de relocations (.rel.text, .rel.data)

Les seules sections qui contiennent des données sont les sections :

— `.text` qui contient l'ensemble des instructions exécutables du programme.
— `.data` qui contient les données initialisées du programme.
— `.bss` qui contient les données non initialisées du programme. Ces données ne prennent pas de place dans le fichier ELF : seules les tailles des zones mémoire à réserver sont spécifiées et ces zones sont remplies avec des zéros au début de l'exécution du programme.

Les autres sections servent à décrire le programme et le rendre portable et relogeable.

A.2 Structure générale d'un fichier objet au format ELF et principe de la relocation

Un fichier objet à lier au format ELF est formé d'un en-tête donnant des informations générales sur la version, la machine, etc., puis d'un certain nombre de pointeurs et de valeurs décrits ci-dessous. Ce que nous appelons ici pointeur est en fait une valeur représentant un déplacement en nombre d'octets par rapport au début du fichier. Les tailles, elles aussi, sont exprimées en nombre d'octets. La figure A.2 donne une idée de la structure d'un fichier au format ELF. Le fichier est constitué d'un *en-tête* donnant les caractéristiques générales du fichier, puis d'un certain nombre de *sections* contenant différentes formes de données, ces sections étant détaillées par la suite (par exemple, section "`.text`", section "`.data`", section des "relocations en zone text", section de la table des symboles, etc).

Lors de la fabrication d'un fichier objet, les instructions du programme sont logées dans une section binaire correspondant à une zone `.text` alors que certaines des opérandes de ces instructions peuvent appartenir à une section binaire différente, par exemple la zone `.data`. Lors du chargement du fichier en mémoire en vue de son exécution (de sa simulation dans notre cas), ces différentes zones sont placées en mémoire par le chargeur. Ce n'est qu'après cette étape que les adresses des opérandes seront connues. Il faut donc mettre à jour la partie adresse effective des instructions afin que ces dernières accèdent correctement aux opérandes (notons qu'avant cette mise à jour, les adresses des opérandes dans les instructions sont des adresses relatives définies par rapport au début de la zone `.data` du fichier initial). Par ailleurs, cette situation peut être généralisée si ce fichier objet fait partie d'un programme plus vaste utilisant plusieurs fichiers objets susceptibles d'être rassemblés en un seul programme par l'éditeur de liens entre fichiers. Par exemple, une opérande en zone `.data` peut être utilisée par des instructions contenues dans des fichiers objets différents. Cette remarque est également valable pour les zones `.text`, par exemple l'appel d'une fonction déclarée dans un fichier externe. Il faut donc indiquer pour chaque section `.text` concernée, un moyen de retrouver l'adresse de cette opérande. Pour cette raison chaque section `.text` ou `.data` possède une section associée dite de relocation (`.rel.text` et `.rel.data`) contenant les informations nécessaires aux calculs des adresses. Comme explicité précédemment, toutes les adresses non définies avant le chargement, sont finalement mise à jour à l'issue du chargement. La partie du code des instructions correspondant à l'adresse des opérandes en question ne peut donc pas être figée au moment de la compilation du fichier objet initial mais seulement à l'issue du chargement du programme. Du fait de cette relocation, les fichiers ELF sont dits "relogeables".

Dans ce projet, pour des raisons de simplicité, nous n'aborderons pas le traitement de l'édition de liens entre plusieurs fichiers ELF. On se contentera de réaliser la simulation d'un fichier objet simple mais susceptible de nécessiter une relocation lors du chargement dans la mémoire du simulateur. Dans la suite de ce chapitre, nous donnons un exemple précis permettant d'illustrer en détails les principes de la relocation.

Entête d'un fichier ELF L'en-tête est décrite par le type C `struct ELF32_Ehdr` dont voici la description des principaux champs :

- `e_ident` : identification du format et des données indépendantes de la machine permettant d'interpréter le contenu du fichier (Cf. exemple dans le paragraphe suivant).
- `e_type` : un fichier relogeable a le type `ET_REL` (constante égale à 1).
- `e_machine` : le type de processeur cible [1].
- `e_version` : la version courante est 1.
- `e_ehsize` : taille de l'en-tête en nombre d'octets.
- `e_shoff` : pointeur sur la *table des en-têtes de sections* (ou plus simplement "*table des sections*"). Chaque entrée de cette table est l'en-tête d'une section qui décrit la nature de cette section, et donne sa localisation dans le fichier (voir ci-dessous). La table des sections est appelée *shdr* dans la documentation ELF. Dans le reste du format ELF, les sections sont généralement désignées par l'index de leur en-tête dans cette table. Le premier index (qui est 0) est une sentinelle qui désigne la section "non définie".
- `e_shnum` : nombre d'entrées dans la table des sections.
- `e_shentsize` : taille d'une entrée de la table des sections.
- `e_shstrndx` : index de la table des noms de sections (dans la table des en-têtes de sections).
- Les autres champs de l'en-tête ne sont pas utilisés dans le cadre du projet. On les met à 0.

En-têtes des sections Un en-tête de section est décrit par le type C `struct Elf32_shdr`. Il définit les champs suivants :

- `sh_name` : index dans la table des noms de sections (section "`.shstrtab`").
- `sh_type` : type de la section. Les types qu'on utilise dans ce projet sont les suivants :
  - `SHT_PROGBITS` (constante 1) : type des sections "`.text`" et "`.data`". Ce type indique que la section contient une suite d'octets correspondant aux données ou aux instructions du programme.
  - `SHT_NOBITS` (constante 8) : type de la section "`.bss`" (données non initialisées). La section ne contient aucune donnée. Elle sert essentiellement à déclarer la taille occupée par les données non initialisées (voir ci-dessous).
  - `SHT_SYMBTAB` (constante 2) : type des tables des symboles. Dans le projet, on en utilise une seule, appelée "`.symtab`".
  - `SHT_STRTAB` (constante 3) : type des tables de chaînes. Dans le projet, deux sections ont ce type : la table des noms de sections, appelée "`.shstrtab`", et la table des noms de symboles, appelée "`.strtab`" (elles sont souvent désignées par "table des chaînes").
  - `SHT_REL` (constante 9) : type des tables de relocations. Dans le projet, on aura : "`.rel.text`" pour les relocations en zone `.text` et "`.rel.data`" pour les relocations en zone `.data`.
  - `ARM_ATTRIBUTES` : type spécifique aux fichiers ELF de processeurs ARM, pour la section "`.ARM.attributes`".
- `sh_offset` : pointeur sur le début de la section dans le fichier.
- `sh_size` : taille qu'occupera la section une fois chargée en mémoire (en octets). Si le type de la section n'est pas `SHT_NOBITS`, la section doit correspondre effectivement `sh_size` octets dans le fichier, puisque la section sera chargée "telle quelle" en mémoire. Si le type de la section est `SHT_NOBITS`, ce champ sert à déclarer la taille de la zone non initialisée qui sera finalement allouée en mémoire.
- `sh_addralign` : contrainte d'alignement sur l'adresse finale de la zone en mémoire. L'adresse finale doit être un multiple de ce nombre.

1. Le processeur ARM qui nous intéresse est identifié par la valeur `EM_ARM` (constante égale à 40)

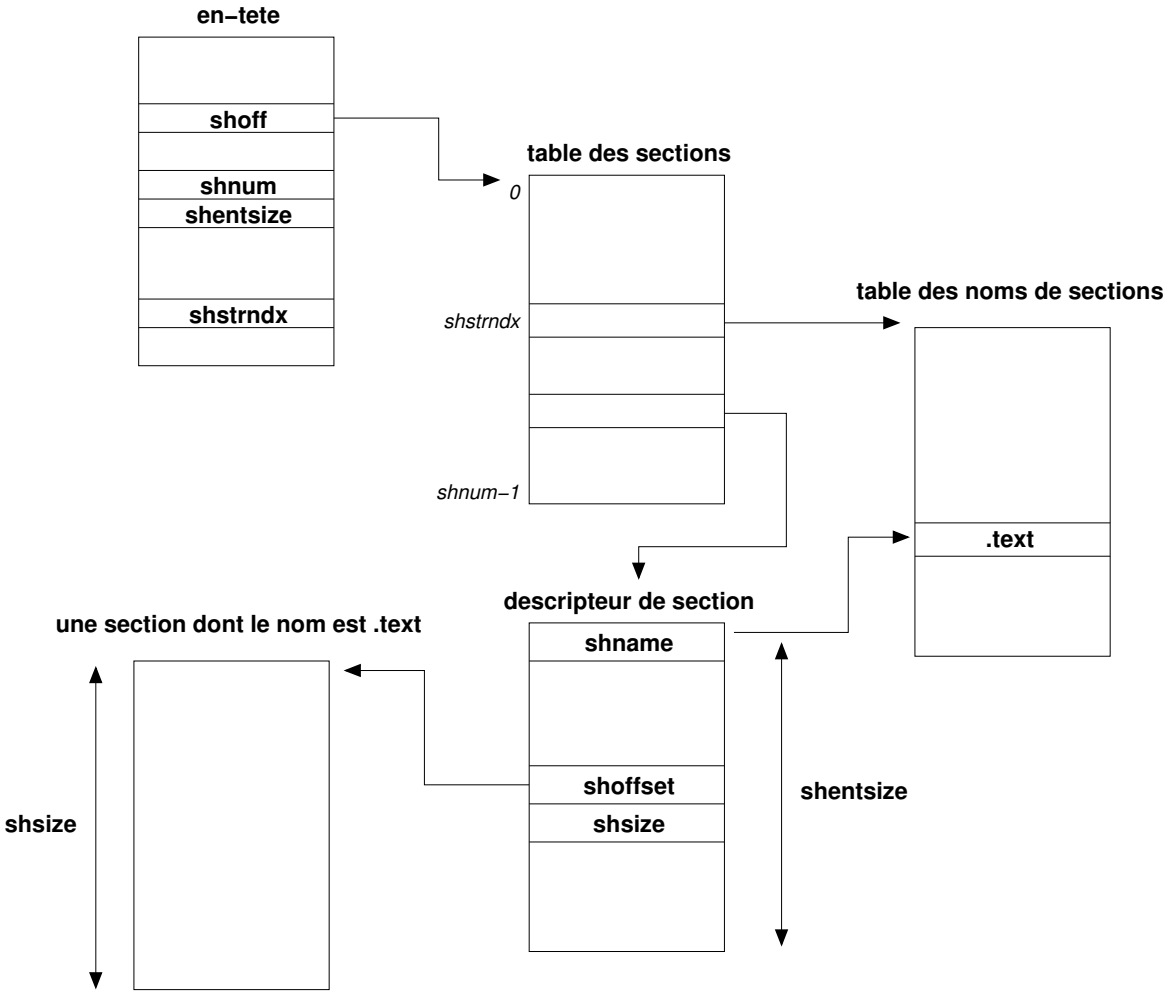


Figure A.2 – Structure d'un fichier relogeable au format ELF

- `sh_entsize` : certaines sections ont des entrées de taille fixe. Cet entier donne donc cette taille. Dans le projet, seules les sections de type `SHT_SYMBTAB` et `SHT_REL` sont concernées par ce champ.
- `sh_link` et `sh_info` : ont des interprétations qui dépendent du type de la section. Dans tous les cas, le champs `sh_link` est l'index d'un en-tête de section (dans la table des en-têtes de sections). Dans le cadre du projet, on a :

`sh_type`	`sh_link`	`sh_info`
`SHT_REL`	l'index de la table de symbole associée	l'index (dans la table des en-têtes de sections) de la section à reloger
`SHT_SYMTAB`	l'index de la table des noms de symboles	l'index (dans la table des symboles) du premier symbole global [2]

Remarque le contenu d'un fichier objet *exécutable* ressemble à celui d'un fichier objet relogeable. Il est formé de segments au lieu de sections et on y trouve ainsi une table des segments (au lieu d'une table des sections). Dans l'en-tête, les informations décrivant la table des segments sont données par les champs dont les noms commencent par `ph` au lieu de `sh`.

A.3 Exemple de fichier relogeable

Pour étudier le format ELF en détaillant le format de chacune des sections considérées dans le projet, considérons le programme en langage d'assemblage `reloc_miam.s` donné Figure A.3.

La commande `arm-as reloc_miam.s -o reloc_miam.o` produit un fichier objet `reloc_miam.o` dont nous donnons en figure A.4 le contenu affiché par la commande Unix `od -t xC reloc_miam.o`.

C'est assez difficile à lire. . . On va donc utiliser les programmes standards `objdump` et `readelf` sous Linux pour analyser les fichiers objets. Il s'agit d'outils capable d'analyser la séquence de bits du fichier pour y retrouver la structure d'un fichier objet, et d'afficher en clair les informations intéressantes. La commande `readelf -a reloc_miam.o` produit :

```
En-tête ELF:
  Magique:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF32
  Données:                           complément à 2, système à octets de poids faible d'abord (little en
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  Version ABI:                       0
  Type:                              REL (Fichier de réadressage)
  Machine:                           ARM
  Version:                           0x1
  Adresse du point d'entrée:               0x0
  Début des en-têtes de programme:          0 (octets dans le fichier)
  Début des en-têtes de section:           180 (octets dans le fichier)
  Fanions:                           0x5000000, Version5 EABI
  Taille de cet en-tête:             52 (bytes)
  Taille de l'en-tête du programme:  0 (bytes)
```

2. On verra en effet en sous-section A.4.6 que tous les symboles globaux doivent être rassemblés à la fin de la table des symboles.

```
# allons au ru
.syntax unified

.global byebye
.text
    MOV r0,#8
    ldr r2,=lunchtime
ldr r1, [r2]
boucle:
CMP r0,r1
ITE NE
    addne r1 , #1
    BLEQ byebye
    B boucle

.thumb_func
byebye:
nop

.bss
tableau: .space 16

.data
debut_cours: .word 8
lunchtime: .word 12
```

Figure A.3 – Programme assembleur nécessitant une relocation.

Figure A.4 – Résultat de `od -t xC reloc_miam.o`.

```
  Nombre d'en-tête du programme:     0
  Taille des en-têtes de section:    40 (bytes)
  Nombre d'en-têtes de section:      9
  Table d'indexes des chaînes d'en-tête de section: 6

En-têtes de section:
  [Nr] Nom               Type            Adr      Décala.Taille ES Fan LN Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00001c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000320 000010 08      7   1  4
  [ 3] .data             PROGBITS        00000000 000050 000008 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000058 000010 00  WA  0   0  1
  [ 5] .ARM.attributes   ARM_ATTRIBUTES  00000000 000058 00001b 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000073 000040 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 00021c 0000d0 10      8  12  4
  [ 8] .strtab           STRTAB          00000000 0002ec 000033 00      0   0  1
Clé des fanions:
  W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes)
  I (info), L (ordre des liens), G (groupe), T (TLS), E (exclu), x (inconnu)
  O (traiterment additionnel requis pour l'OS) o (spécifique à l'OS), p (spécifique au processeur)

Il n'y a pas de groupe de section dans ce fichier.
```

```
Il n'y a pas d'en-têtes de programme dans ce fichier.

Section de réadressage '.rel.text' à l'adresse de décalage 0x320 contient 2 entrées:
 Décalage    Info    Type            Val.-sym   Noms-symboles
0000000e  00000c0a R_ARM_THM_CALL    00000015   byebye
00000018  00000202 R_ARM_ABS32       00000000   .data

Il n'y a pas de section de déroulage dans ce fichier.

Table de symboles « .symtab » contient 13 entrées:
   Num:    Valeur Tail Type    Lien   Vis      Ndx Nom
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    3
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $t
     5: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 lunchtime
     6: 00000008     0 NOTYPE  LOCAL  DEFAULT    1 boucle
     7: 00000000     0 NOTYPE  LOCAL  DEFAULT    4 tableau
     8: 00000000     0 NOTYPE  LOCAL  DEFAULT    4 $d
     9: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 debut_cours
    10: 00000016     0 NOTYPE  LOCAL  DEFAULT    1 $d
    11: 00000000     0 SECTION LOCAL  DEFAULT    5
    12: 00000015     0 FUNC    GLOBAL DEFAULT    1 byebye

Aucune information de version repérée dans ce fichier.
Attribute Section: aeabi
Attributs du fichier
  Tag_CPU_name: "7-M"
  Tag_CPU_arch: v7
  Tag_CPU_arch_profile: Microcontrolleur
  Tag_THUMB_ISA_use: Thumb-2
```

A.4 Détail des sections

A.4.1 L'en-tête

```
0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0000020 01 00 28 00 01 00 00 00 00 00 00 00 00 00 00 00
0000040 b4 00 00 00 00 00 00 05 34 00 00 00 00 00 28 00
0000060 09 00 06 00
```

Les 16 premiers octets constituent le champ `e_ident` (taille définie par la constante `EI_IDENT`). Les quatre premiers identifient le format : notons que `0x45, 0x4c` et `0x46` sont les codes ASCII des caractères `'E', 'L', 'F'`. Le cinquième octet `01` donne la classe du fichier, ici `ELFCLASS32`, ce qui signifie que les adresses sont exprimées sur 32 bits. Le sixième donne le type de codage des données, ici 1, ce qui signifie que les données sont codées en complément à deux, avec les bits les plus significatifs occupant les adresses les plus hautes (little endian).

Par ailleurs on repère : la taille de l'en-tête (0x34) ; le déplacement par rapport au début du fichier donnant accès à la table des sections (0xb4 octets = 180 en décimal) ; la taille d'une entrée de la table des sections (0x28 = 40 octets), le nombre d'entrées dans la table des sections (0x09 = 9).

L'entrée numéro 6 dans la table des sections est celle du descripteur de la table des noms de sections `.shstrtab` (celle-ci est indispensable pour savoir quelle section décrit un autre descripteur).

La commande `readelf -S reloc_miam.o` permet d'obtenir l'en-tête des sections du ELF et de savoir ainsi quelles sont les sections qui le constituent. On peut ainsi voir qu'il existe une section `.text` de type `.PROGBITS` (bits appartenant à un programme) se trouvant à l'offset `0x34` et faisant `0x1c` octets.

```
En-têtes de section:
  [Nr] Nom               Type            Adr      Décala.Taille ES Fan LN Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00001c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000320 000010 08      7   1  4
  [ 3] .data             PROGBITS        00000000 000050 000008 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000058 000010 00  WA  0   0  1
  [ 5] .ARM.attributes   ARM_ATTRIBUTES  00000000 000058 00001b 00      0   0  1
  [ 6] .shstrtab         STRTAB          00000000 000073 000040 00      0   0  1
  [ 7] .symtab           SYMTAB          00000000 00021c 0000d0 10      8  12  4
  [ 8] .strtab           STRTAB          00000000 0002ec 000033 00      0   0  1
Clé des fanions:
  W (écriture), A (allocation), X (exécution), M (fusion), S (chaînes)
  I (info), L (ordre des liens), G (groupe), T (TLS), E (exclu), x (inconnu)
  O (traiterment additionnel requis pour l'OS) o (spécifique à l'OS), p (spécifique au processeur)
```

Figure A.5 – Table des section : résultat de `readelf -S reloc_miam.o`.

A.4.2 La table des noms de sections (`.shstrtab`)

La table des noms de section est accessible par la commande `readelf --hex-dump=6 reloc_miam.o`. En effet, la section `.shstrtab` est à l'indice 6. La table des noms de sections est du type table de chaînes de caractères. Elle contient les noms suivants :

```
Vidange hexadécimale de la section « .shstrtab »:
  0x00000000 002e7379 6d746162 002e7374 72746162  .symtab .strtab
  0x00000010 002e7368 73747274 6162002e 72656c2e  .shstrtab .rel.
  0x00000020 74657874 002e6461 7461002e 62737300 text .data .bss
  0x00000030 2e41524d 2e617474 72696275 74657300 .ARM.attributes.
```

C'est dans cette table que `readelf` va chercher les noms des sections pour remplir la table donnée figure A.5. Le format ELF impose certaines règles à respecter. La première chaîne doit être la chaîne vide (`0x00`). Chaque chaîne doit se terminer par le caractère de code ASCII 0 (comme en "C").

A.4.3 La section table des chaînes (`.strtab`)

Cette section (qui porte l'index 8) rassemble tous les noms des symboles utilisés (directement ou implicitement) dans le code.

```
Vidange hexadécimale de la section « .strtab »:
  0x00000000 00247400 6c756e63 6874696d 6500626f .$t.lunchtime.bo
  0x00000010 75636c65 00746162 6c656175 00246400 ucle.tableau.$d.
  0x00000020 64656275 745f636f 75727300 62796562 debut_cours.byeb
  0x00000030 796500                              ye.
```

C'est dans cette table que `readelf` va chercher les noms des symboles.

A.4.4 La section `.text`

La portion de fichier objet correspondant à la zone TEXT (les instructions) est le résultat de la commande `readelf --hex-dump=1 reloc_miam.o` :

```
Vidange hexadécimale de la section « .text »:
  0x00000000 4ff00800 044a1168 884214bf 0131fff7 O....J.h.B...1..
  0x00000010 fefff9e7 00bf0000 04000000          ............
```

Une version plus lisible de la zone TEXT peut être obtenue avec `arm-objdump -d --section=.text reloc_miam.o` :

```
Disassembly of section .text:

00000000 <boucle-0x8>:
   0: f04f 0008  mov.w r0, #8
   4: 4a04       ldr r2, [pc, #16] ; (18 <byebye+0x4>)
   6: 6811       ldr r1, [r2, #0]

00000008 <boucle>:
   8: 4288       cmp r0, r1
   a: bf14       ite ne
   c: 3101       addne r1, #1
   e: f7ff fffe  bleq 14 <byebye>
  12: e7f9       b.n 8 <boucle>

00000014 <byebye>:
  14: bf00       nop
  16: 0000       .short 0x0000
  18: 00000004  .word 0x00000004
```

On peut par exemple constater que l'instruction `ldr r2, =lunchtime` a été remplacé par `ldr r2, [pc, #16] == 4a04` et que deux 'données' sont apparues en fin de programme `.short 0x0000` et `.word 0x00000004`.

Commencons par la section data. Le chargement de l'adresse lunchtime n'est pas possible directement (une instruction 32 bits ne peut contenir une valeur 32 bits). L'assemble utilise donc la méthode du *Literal pool* qui consiste à copier "en dur" l'addresse à la fin de la section .text. C'est pourquoi on trouve `.word 0x00000004` à la fin de la section .data. le chiffre 0x4 indique bien que la valeur 'pointée par' lunchtime se trouve 4 octets après le debut de la section data. `.short 0x0000` est quand à lui une valeur de 'bourage' (*padding*) qui permet d'assurer que le prochain .word sera bien aligné sur 4 octets (cad, l'adresse du .word sera bien un multiple de 4).

Dans la section text, l'instruction `ldr r2, [pc, #16] == 4a04` est encodée comme suit : le premier octets codent le numéro de l'instruction et le registre r2 tandis que le dernier code le décalage à appliquer à PC (ici 4 << 2 soit 16). Ainsi, le calcul de l'adresse à laquelle cherche le mot s'effectue simplement par PC+offset. Dans notre cas on à PC = 8 (l'adresse du ldr est 4, donc PC vaut 4+4) tandis que l'offset vaut 16 (4 << 2). L'adresse résultante sera donc 8 + 16 = 0x18. Soit l'adresse à laquelle est stockée l'adresse de lunchtime.

Cependant, la valeur de `lunchtime` n'est pas connue. En effet, `lunchtime` est une adresse est cette adresse ne sera connue que lors du chargement final en mémoire, on a donc une valeur temporaire (ici 4) en attendant d'être replacés par une valeur lors du chargement grâce à une donnée de relocation `rel.text` qui est associée à cette instruction (cf. section A.4.7).

A.4.5 La section .data

La portion de fichier objet correspondant à la zone DATA (les données initialisées) est donnée par la commande `readelf --hex-dump=3 reloc_miam.o` :

```
Vidange hexadécimale de la section « .data »:
  0x00000000 08000000 0c000000                    ........
```

La valeur 8 indiquée (pointée par) `debut_cours` est codée sur les 4 premiers octets (c'est un `.word`). La valeur 12 (`0x0c000000 == 12`) indiquée (pointée par) `lunchtime` est codée sur les 4 octets suivants (c'est un `.word`).

A.4.6 La section table des symboles

La table des symboles d'un fichier objet est donnée par la commande `readelf -s reloc_miam.o` :

```
Table de symboles « .symtab » contient 13 entrées:
   Num:  Valeur Taille Type    Lien   Vis      Ndx Nom
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 SECTION LOCAL  DEFAULT    1
     2: 00000000     0 SECTION LOCAL  DEFAULT    3
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 NOTYPE  LOCAL  DEFAULT    1 $t
     5: 00000004     0 NOTYPE  LOCAL  DEFAULT    3 lunchtime
     6: 00000008     0 NOTYPE  LOCAL  DEFAULT    1 boucle
     7: 00000000     0 NOTYPE  LOCAL  DEFAULT    4 tableau
     8: 00000000     0 NOTYPE  LOCAL  DEFAULT    4 $d
     9: 00000000     0 NOTYPE  LOCAL  DEFAULT    3 debut_cours
    10: 00000016     0 NOTYPE  LOCAL  DEFAULT    1 $d
    11: 00000000     0 SECTION LOCAL  DEFAULT    5
    12: 00000015     0 FUNC    GLOBAL DEFAULT    1 byebye
```

Le type décrivant une entrée de la table des symboles est `struct Elf32Sym`. Une entrée occupe 16 octets, il y a ici 12 entrées. La première entrée est à zéro (elle sert de sentinelle). Les entrées de numéros 1 à 3 sont des symboles spéciaux qui représentent en fait respectivement les sections `.text`, `.data`, `.bss`. On remarque en effet qu'elles ont le type `SECTION` (constante 3). Le champ `Ndx` de ces entrées indique dans quelle section le symbole est défini (par exemple `tableau` est défini dans la section `4 == .bss`). Le champ `Valeur` indique l'offset à appliquer à partir de la section des symboles pour trouver l'adresse des symboles. Les entrées numéros 5 à 12 sont de vrais symboles de l'utilisateur, correspondants aux étiquettes `lunchtime`, `boucle`, `byebye` , `tableau`, `debut_cours` et `adresse_lunchtime`. On constate aussi l'apparition de $t et $d qui indique respectivement un début de région contenant du code Thumb et un début de région contenant des données.
Les différents champ de la structure `Elf32Sym` sont les suivants :

- `st_name` : index du symbole dans la table des noms de symboles. Lorsque le symbole est de type `STT_SECTION`, aucun nom ne lui est associé. La valeur de l'index est alors 0 (qui désigne donc la chaîne vide, d'après les contraintes sur `.strtab`).
- `st_value` : il vaut 0 pour un symbole non défini ; pour un symbole défini localement, `st_value` est le déplacement (en octets) par rapport au début de la zone de définition.
- `st_shndx` : indique l'index (dans la table des en-têtes de sections) de la section où le symbole est défini. Si le symbole n'est défini dans aucune section (symbole externe), cet index vaut 0.

Ainsi **l'index 0 de la table des en-têtes de section est une sentinelle qui sert à marquer les symboles externes**. Si le symbole est de type `STT_SECTION`, cet index est directement l'index de la section.

- `st_size` et `st_other` : non utilisés dans le projet (`st_size` sert à associer une taille de donnée au symbole).
- `st_info` : ce champ codé sur un octet sert en fait à coder 2 champs : le champ "bind" et le champ "type". Les macros suivantes (définie dans les fichiers d'en-tête du format ELF) permettent d'encoder ou de décoder le champ `st_info` :

```
#define ELF32_ST_BIND(i)   ((i)>>4)               /* de info vers bind */
#define ELF32_ST_TYPE(i)   ((i)&0xf)              /* de info vers type */
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))   /* de (bind,type) vers info */
```

  Le champ "bind" indique la portée du symbole. Dans le cadre du projet, on considère les 2 cas suivants :
  - `STB_LOCAL` (constante 0) indique que le symbole est défini localement et non exporté.
  - `STB_GLOBAL` (constante 1) indique que le symbole est soit défini et exporté, soit non défini et importé. Il faut noter qu'à l'édition de lien, il est interdit à 2 fichiers distincts de définir deux symboles de même nom ayant la portée `STB_GLOBAL`.

  Dans le cadre du projet, on considérera les cas suivants pour le champ "type" :
  - `STT_SECTION` (constante 3) indiquant que le symbole désigne en fait une section.
  - `STT_FUNC` (constante 2) indiquant que le symbole désigne en fait une fonction.
  - `STT_NOTYPE` (constante 0), dans les autres cas.

Le format ELF impose certaines contraintes sur l'ordre des symboles dans la table : le premier symbole n'est pas utilisé. Tous les symboles globaux doivent être regroupés à la fin de la table.

A.4.7 Les sections de relocation

Les données de relocation `.rel.text` décrivent comment réécrire certains champs d'instruction incomplets le codage des instructions de la zone `.text`. Ces instructions sont partiellement codées, car contenant des références à des symboles (étiquettes) dont on ne peut pas connaître l'adresse au moment de la fabrication du fichier objet. L'instruction est codée en réservant les 4 octets nécessaires au stockage de la valeur d'adresse mais la valeur indiquée dans le champ est provisoire. Cependant, cette valeur est très importante, comme nous allons le voir. Elle dépend du mode de relocation, et permet de calculer la valeur finale du champs. Les données de relocation `.rel.date`, selon le même principe, décrivent comment réécrire certaines valeurs dans la section data.

Format de la table de relocation

Une entrée de la table de relocation est représentée par le type `struct Elf32_Rel` :

- `r_offset` : contient un décalage en octet par rapport au début de la section. Cette valeur indique la position dans la zone à reloger de l'instruction qui contient un champ incomplet.
- `r_info` : est un champ codé sur 4 octets composé en fait de deux champs :.
  - Le champ "`sym`" est l'index du symbole à reloger dans la table des symboles.
  - Le champ "`type`" indique le mode de calcul de la relocation.

Les macros de codage/décodage du champ `r_info` défini dans les fichiers d'en-têtes de la librairie ELF sont :

```
#define ELF32_R_SYM(i)    ((i)>>8)                      /* info vers sym */
#define ELF32_R_TYPE(i)   ((unsigned char)(i))          /* info vers type */
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t)) /* (sym,type) vers info */
```

Modes de calcul de la relocation

Détaillons maintenant le mode de calcul de la relocation, c'est-à-dire la valeur que l'éditeur de lien (ou le chargeur de notre simulateur) met finalement dans les champs incomplets des instructions. Les notations sont les suivantes :

- **V** désigne la **V**aleur "finale" du champ accueillant le relogement.
- **P** désigne la **P**lace, c'est à dire l'adresse "finale" de l'élément à reloger.
- **S** désigne l'adresse "finale" du **S**ymbole par rapport auquel on reloge.
- **A** désigne la valeur à **a**jouter pour calculer la valeur du champ à reloger (c'est la valeur du champ avant relogement).
- **T** vaut 1 si S est de type `STT_FUNC` et que le symbole pointe sur une instruction Thumb (ce qui est toujours le cas pour l'ARMv7-M). T vaut 0 sinon. [3].

Les adresses finales des sections `.text`, `.data`, `.bss` sont normalement déterminées lors de l'édition de liens. Dans notre simulateur, elles sont calculées lors du chargement des sections en mémoire suivant les contraintes définies section 5.2.1.

Le mode de calcul dépend du type de relocation, codé dans le champs `type` de `r_info`. Les principaux modes de calcul sont :

- `R_ARM_ABS32` (constante 2) : la valeur mise à l'adresse P vaut $V = (S + A)|T$. Ce mode sert pour les adressages immédiats.
- `R_ARM_ABS8` (constante 8) : la valeur mise à l'adresse P vaut $V = (S + A)$. Ce mode sert pour les adressages immédiats avec une valeur sur 8 bits.
- `R_ARM_THM_CALL` (constante 10) : la valeur mise à l'adresse P vaut $V = ((S+A)|T)–P$. Ce mode sert pour les branchements.

La section de relocation `.rel.text`

La relocation est peut-être plus simple à comprendre en regardant le résultat de la commande `readelf -r reloc_miam.o`. Sur l'exemple, la table des relocations en texte est :

```
Section de réadressage '.rel.text' à l'adresse de décalage 0x320 contient 2 entrées:
 Décalage    Info    Type            Val.-sym   Noms-symboles
0000000e  00000c0a R_ARM_THM_CALL    00000015   byebye
00000018  00000202 R_ARM_ABS32       00000000   .data
```

Pour la première entrée le champs `r_offet` vaut `0xE`, ce qui signifie que l'instruction à reloger est à 14 octets (ou 0xE) du début de la section `.text`. Effectivement, il s'agit bien de `BLEQ byebye`.

Le champ `info` vaut 0x00000c0a, donc se décompose en :

```
sym = (info)>>8 = 0x00000c0a >> 8 = 0x0000000c = 12
type = (unsigned char)(info) = (unsigned char)(0x00000c0a) = 0x0a
```

Le champ `type` vaut 10, ce qui signifie que le mode de relocation est `R_ARM_THM_CALL`. Il faut donc déterminer S et T. Comme byebye est un symbole `STT_FUNC` et que l'instruction à reloger est une instruction Thumb (ce sera toujours le cas) $T = 1$. Pour S, le champ `sym` vaut 12 et correspond au symbole byebye. Donc ici $S =$ `Val.-sym` $= 0x15$. Le addend de l'instruction vaut $0xFFFE << 1$ soit $A = -4$, et $P = 0xE$ et l'adresse l'instruction à reloger `BLEQ byebye`. On a donc $V = ((S+A)|T)–P$ soit $V = ((0x15 - 4)|0x1) - 0xE = 0x11 - 0xE = 0x03$. La valeur d'offset codée dans le BL vaudra $0x3 >> 1 = 1$. Donc lorsque le saut sera effectué, le microprocesseur utilisera l'offset $0x1 << 1 = 2$ la valeur de PC qui vaut ici l'adresse de BLEQ + 4 soit $PC = 0xE + 4 = 0x12$ et donc l'adresse finale de branchement sera $V = 0x12 + 2 = 0x14$ et cela correspond bien à l'adresse de byebye.

Essayez maintenant d'effectuer la relocation de l'entrée 2 de la table.

Annexe B

Grammaire des commandes de l'émulateur

La grammaire des commandes de l'interpreteur doit respecter les règles de production ci-dessous :

```
<commande> ::=  <assert> | <break> | "debug" | <disasm> | <disp> | "exit" | <help> |
                <load> | "resume" |<run> | <set> | <step>

<assert> ::= "assert" "reg" <reg> <integer32>
<assert> ::= "assert" "word" <@> <integer32>
<assert> ::= "assert" "byte" <@> <integer8>

<break> ::= "break" "add" <@>+
<break> ::= "break" "del" <@>|"all"
<break> ::= "break" "list"

<disasm> ::= "disasm" <range>

<disp> ::= "disp" "mem" "map"|<range>+
<disp> ::= "disp" "reg" "all"|<reg>+

<help> ::=  "help" "assert" | "break" | "debug" | "disasm" | "disp" | "exit" | "help" |
                   "load" | "resume" | "run"  | "set" | "step"

<load> ::= "load" <filename> [<@>]

<run> ::= "run" [<@>]

<set> ::= "set" "mem" (("byte" <@> <integer8>) | ("word" <@> <integer32>))+
<set> ::= "set" "reg" (<reg> <integer32>)+

<step> ::= "step" ["into"]
```

Les symboles terminaux sont

```
<reg>       ::= un parmi tous les mnemoniques et numéros de registre
<integer32> ::= entier signé sur 32 bits
<integer8>  ::= entier signé sur 8 bits
<integer>   ::= integer32|integer8
<@>         ::= entier non-signé sur 32 bits représenté en hexadécimal
<range>     ::= @+integer | @:@
```

Annexe C

Spécifications détaillées des instructions

Cette annexe contient les spécifications des instructions étudiées dans ce projet. Elles sont directement issues de la documentation [3].

C.1 Définitions et notations

Commençons par rappeller quelques définitions et notations utiles.

Octet/Mot Un *octet* (byte en anglais) est une suite de 8 bits qui constitue la plus petite entité que l'on peut adresser sur la machine. La concaténation de deux octets forme un *demi-mot* (half-word) de 16 bits, et la concaténation de quatre octets, ou de deux demi-mots, forme un *mot* (word) de 32 bits. Les bits sont numérotés de la droite (poids faible) vers la gauche (poids fort) de 0 à 7 pour l'octet, de 0 à 15 pour un demi-mot et de 0 à 31 pour un mot.

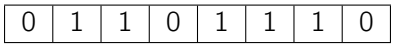


Figure C.1 – Octet

Représentation hexadécimale d'un octet/mot On représente par 0xij la valeur d'un octet dont les 4 bits de poids fort valent i et les 4 bits de poids faible j (avec $i, j \in ([0...9, A...F])$. Par exemple, la valeur de l'octet de la figure C.1 s'écrit 0x6E en hexadécimal. Pour un demi-mot ou un mot, on aura respectivement 4 ou 8 chiffres hexadécimaux, chacun représentant 4 bits.

Codage binaire d'un entier non signé Quand on parle d'un entier non signé codé sur n bits ou plus simplement d'un entier codé sur n bits, il s'agit de sa représentation en base 2 sur n bits, donc d'une valeur entière comprise entre 0 et $2^n - 1$. Un entier codé sur un octet a donc une valeur comprise entre 0 et 255 correspondant aux images binaires 0x00 à 0xFF, un entier codé sur un demi-mot a une valeur comprise entre 0 et 65535 correspondant aux images binaires 0x0000 à 0xFFFF, et un entier codé sur un mot a une valeur comprise entre 0 et 4294967295 correspondant aux images binaires 0x00000000 à 0xFFFFFFFF.

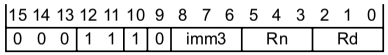
Codage binaire d'un entier signé Les entiers signés sont représentés en complément à 2. Le codage sur n bits du nombre i est la représentation en base 2 sur n bits de $2^n + i$, si $-2^{n-1} \leq i \leq -1$, et de i, si $0 \leq i \leq 2^{n-1} - 1$. Un entier signé codé sur un octet est compris entre -128 à 127 correspondant aux images binaires 0x80 à 0x7F. Un entier signé sur un demi-mot est compris entre -32768 et 32767 correspondant à l'intervalle binaire 0x8000 à 0x7FFF. Enfin, un entier signé sur un mot est compris entre -2147483648 et 2147483647 correspondant à l'intervalle binaire 0x80000000 à 0x7FFFFFFF. On remarque que le bit de plus fort poids d'un octet/mot/long mot représentant un entier négatif est toujours égal à 1 alors qu'il vaut 0 pour un nombre positif (c'est le bit de signe).

A7.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

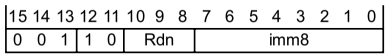
```
ADDS <Rd>,<Rn>,#<imm3>                    Outside IT block.
ADD<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```




```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb instruction set.

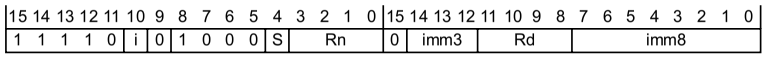
```
ADDS <Rdn>,#<imm8>                        Outside IT block.
ADD<c> <Rdn>,#<imm8>                      Inside IT block.
```




```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

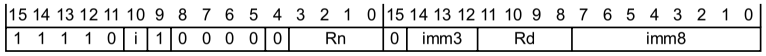
```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```




```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```




```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```


Assembler syntax

```
ADD{S}<c><q>  {<Rd>,} <Rn>, #<const>      All encodings permitted
ADDW<c><q>  {<Rd>,} <Rn>, #<const>        Only encoding T4 permitted
```

where:

S — If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> — See *Standard assembler syntax fields* on page A7-207.

<Rd> — Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

<Rn> — Specifies the register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus immediate)* on page A7-225. If the PC is specified for <Rn>, see *ADR* on page A7-229.

<const> — Specifies the immediate value to be added to the value obtained from <Rn>. The range of allowed values is 0-7 for encoding T1, 0-255 for encoding T2 and 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A5-166 for the range of allowed values for encoding T3.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the ADDW syntax). Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.

The pre-UAL syntax ADD<c>S is equivalent to ADDS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

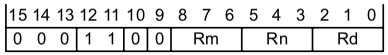
A7.7.4 ADD (register)

ADD (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

`ADDS <Rd>,<Rn>,<Rm>` Outside IT block.

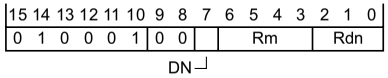
`ADD<c> <Rd>,<Rn>,<Rm>` Inside IT block.




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb instruction set.

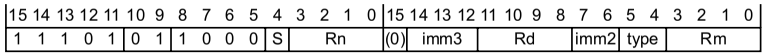
`ADD<c> <Rdn>,<Rm>`




```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = UInt(DN:Rdn);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if d == 15 && m == 15 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

`ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`




```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```


Assembler syntax

`ADD{S}<c><q> {<Rd>,} <Rn>, <Rm> {,<shift>}`

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>` and encoding T2 is preferred to encoding T1 if both are available. This can only happen inside an IT block. If `<Rd>` is specified, encoding T1 is preferred to encoding T2. If `<Rm>` is not the PC, the PC can be used in encoding T2.

`<Rn>` Specifies the register that contains the first operand. If the SP is specified for `<Rn>`, see *ADD (SP plus register)* on page A7-227. If `<Rm>` is not the PC, the PC can be used in encoding T2.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand. The PC can be used in encoding T2.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and all encodings are permitted. If `<shift>` is specified, only encoding T3 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A7-212.

Inside an IT block, if `ADD<c> <Rd>,<Rn>,<Rd>` cannot be assembled using encoding T1, it is assembled using encoding T2 as though `ADD<c> <Rd>,<Rn>` had been written. To prevent this happening, use the `.W` qualifier.

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

A7.7.5 ADD (SP plus immediate)

ADD (SP plus immediate) adds an immediate value to the SP value, and writes the result to the destination register.

Encoding T1 All versions of the Thumb instruction set.

```
ADD<c> <Rd>,SP,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1	Rd			imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

Encoding T2 All versions of the Thumb instruction set.

```
ADD<c> SP,SP,#<imm7>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Encoding T3 ARMv7-M

```
ADD{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 && S == '0' then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```


Assembler syntax

`ADD{S}<c><q> {<Rd>,} SP, #<const>`	All encodings permitted
`ADDW<c><q> {<Rd>,} SP, #<const>`	Only encoding T4 is permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is SP.

`<const>` Specifies the immediate value to be added to the value obtained from `<Rn>`. Allowed values are multiples of 4 in the range 0-1020 for encoding T1, multiples of 4 in the range 0-508 for encoding T2 and any value in the range 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A5-166 for the range of allowed values for encoding T3.

When both 32-bit encodings are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `ADDW` syntax).

The pre-UAL syntax `ADD<c>S` is equivalent to `ADDS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

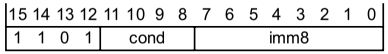
None.

A7.7.12 B

Branch causes a branch to a target address.

Encoding T1 All versions of the Thumb instruction set.

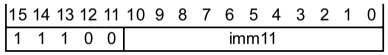
`B<c> <label>` Not allowed in IT block.




```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb instruction set.

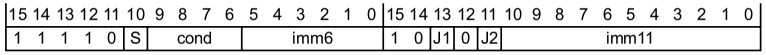
`B<c> <label>` Outside or last in IT block




```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

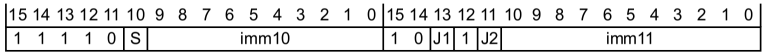
Encoding T3 ARMv7-M

`B<c>.W <label>` Not allowed in IT block.



Encoding T4 ARMv7-M

`B<c>.W <label>` Outside or last in IT block




```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```


Assembler syntax

`B<c><q> <label>`

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

Note

Encodings T1 and T3 are conditional in their own right, and do not require an `IT` instruction to make them conditional.

For encodings T1 and T3, `<c>` is not allowed to be `AL` or omitted. The 4-bit encoding of the condition is placed in the instruction and not in a preceding `IT` instruction, and the instruction is not allowed to be in an IT block. As a result, encodings T1 and T2 are never both available to the assembler, nor are encodings T3 and T4.

`<label>` Specifies the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that will set `imm32` to that offset.

Allowed offsets are even numbers in the range -256 to 254 for encoding T1, -2048 to 2046 for encoding T2, -1048576 to 1048574 for encoding T3, and -16777216 to 16777214 for encoding T4.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

Exceptions

None.

Related encodings

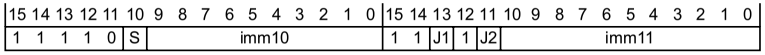
If the cond field of encoding T3 is '1110' or '1111', a different instruction is encoded. To determine which instruction, see *Branches and miscellaneous control* on page A5-169.

A7.7.18 BL

Branch with Link (immediate) calls a subroutine at a PC-relative address.

Encoding T1 All versions of the Thumb instruction set.

`BL<c> <label>` Outside or last in IT block




```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
targetInstrSet = CurrentInstrSet();
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
BL<c><q>  <label>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<label>` Specifies the label of the instruction that is to be branched to.

The assembler calculates the required value of the offset from the PC value of the `BL` instruction to this label, then selects an encoding that will set `imm32` to that offset. Allowed offsets are even numbers in the range -16777216 to 16777214.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    next_instr_addr = PC;
    LR = next_instr_addr<31:1> : '1';
    BranchWritePC(PC + imm32);
```

Exceptions

None.

Note

Before the introduction of Thumb-2 technology, J1 and J2 in encodings T1 and T2 were both 1, resulting in a smaller branch range. The instructions could be executed as two separate 16-bit instructions, with the first instruction `instr1` setting LR to `PC + SignExtend(instr1<10:0>:'000000000000', 32)` and the second instruction completing the operation. It is not possible to split the `BL` instruction into two 16-bit instructions in ARMv6-M and ARMv7-M.

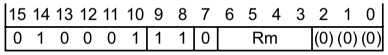
A7.7.20 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register. ARMv7-M only supports the Thumb instruction set. An attempt to change the instruction execution state causes the processor to take an exception on the instruction at the target address.

BX can also be used for an exception return, see *Exception return behavior* on page B1-652.

Encoding T1 All versions of the Thumb instruction set.

`BX<c> <Rm>` Outside or last in IT block




```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
BX<c><q>  <Rm>
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rm>` Specifies the register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

Exceptions

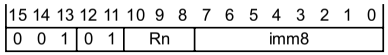
UsageFault.

A7.7.27 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb instruction set.

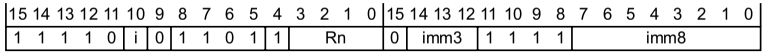
```
CMP<c> <Rn>,#<imm8>
```




```
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
```

Encoding T2 ARMv7-M

```
CMP<c>.W <Rn>,#<const>
```




```
n = UInt(Rn);  imm32 = ThumbExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```


Assembler syntax

```
CMP<c><q>  <Rn>, #<const>
```

where:

<c><q> See *Standard assembler syntax fields* on page A7-207.

<Rn> Specifies the register that contains the operand. This register is allowed to be the SP.

<const> Specifies the immediate value to be added to the value obtained from <Rn>. The range of allowed values is 0-255 for encoding T1. See *Modified immediate constants in Thumb instructions* on page A5-166 for the range of allowed values for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

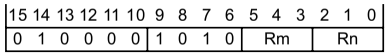
None.

A7.7.28 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

Encoding T1 All versions of the Thumb instruction set.

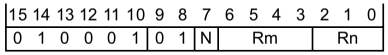
`CMP<c> <Rn>,<Rm>` `<Rn>` and `<Rm>` both from R0-R7




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb instruction set.

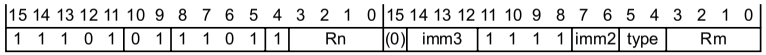
`CMP<c> <Rn>,<Rm>` `<Rn>` and `<Rm>` not both from R0-R7




```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

`CMP<c>.W <Rn>, <Rm> {,<shift>}`




```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m IN {13,15} then UNPREDICTABLE;
```


Assembler syntax

`CMP<c><q>  <Rn>, <Rm> {,<shift>}`

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rn>` Specifies the register that contains the first operand. The SP can be used.

`<Rm>` Specifies the register that is optionally shifted and used as the second operand. The SP can be used, but use of the SP is deprecated.

`<shift>` Specifies the shift to apply to the value read from `<Rm>`. If `<shift>` is omitted, no shift is applied and all encodings are permitted. If shift is specified, only encoding T3 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A7-212.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A7.7.37 IT

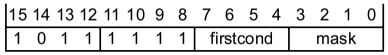
If Then makes up to four following instructions (the *IT block*) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

IT does not affect the condition code flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than CMP, CMN and TST, do not set the condition code flags. The AL condition can be specified to get this changed behavior without conditional execution.

Encoding T1 ARMv7-M

IT{x{y{z}}} <firstcond> Not allowed in IT block




```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

Related encodings

If the mask field of encoding T1 is '0000', a different instruction is encoded. To determine which instruction, see *If-Then, and hints* on page A5-162.

Assembler syntax

IT{x{y{z}}}<q> <firstcond>

where:

<x> Specifies the condition for the second instruction in the IT block.

<y> Specifies the condition for the third instruction in the IT block.

<z> Specifies the condition for the fourth instruction in the IT block.

<q> See *Standard assembler syntax fields* on page A7-207.

<firstcond> Specifies the condition for the first instruction in the IT block.

Each of <x>, <y>, and <z> can be either:

T Then. The condition attached to the instruction is <firstcond>.

E Else. The condition attached to the instruction is the inverse of <firstcond>. The condition code is the same as <firstcond>, except that the least significant bit is inverted. E must not be specified if <firstcond> is AL.

Table A7-3 shows how the values of <x>, <y>, and <z> determine the value of the mask field.

Table A7-3 Determination of mask[a] field

<x>	<y>	<z>	mask[3]	mask[2]	mask[1]	mask[0]
omitted	omitted	omitted	1	0	0	0
T	omitted	omitted	firstcond[0]	1	0	0
E	omitted	omitted	NOT firstcond[0]	1	0	0
T	T	omitted	firstcond[0]	firstcond[0]	1	0
E	T	omitted	NOT firstcond[0]	firstcond[0]	1	0
T	E	omitted	firstcond[0]	NOT firstcond[0]	1	0
E	E	omitted	NOT firstcond[0]	NOT firstcond[0]	1	0
T	T	T	firstcond[0]	firstcond[0]	firstcond[0]	1
E	T	T	NOT firstcond[0]	firstcond[0]	firstcond[0]	1
T	E	T	firstcond[0]	NOT firstcond[0]	firstcond[0]	1
E	E	T	NOT firstcond[0]	NOT firstcond[0]	firstcond[0]	1
T	T	E	firstcond[0]	firstcond[0]	NOT firstcond[0]	1
E	T	E	NOT firstcond[0]	firstcond[0]	NOT firstcond[0]	1
T	E	E	firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1
E	E	E	NOT firstcond[0]	NOT firstcond[0]	NOT firstcond[0]	1

a. In any mask, at least one bit must be 1.

See also *ITSTATE* on page A7-210.

Operation

```
EncodingSpecificOperations();
ITSTATE.IT<7:0> = firstcond:mask;
```

Exceptions

None.

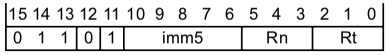
A7.7.42 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A7-214 for information about memory accesses.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb instruction set.

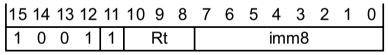
```
LDR<c> <Rt>, [<Rn>{,#<imm5>}]
```




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb instruction set.

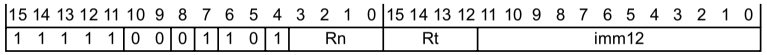
```
LDR<c> <Rt>,[SP{,#<imm8>}]
```




```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T3 ARMv7-M

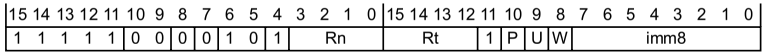
```
LDR<c>.W <Rt>,[<Rn>{,#<imm12>}]
```




```
if Rn == '1111' then SEE LDR (literal);
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
LDR<c> <Rt>,[<Rn>,#-<imm8>]
LDR<c> <Rt>,[<Rn>],#+/-<imm8>
LDR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```




```
if Rn == '1111' then SEE LDR (literal);
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if Rn == '1101' && P == '0' && U == '1' && W == '1' && imm8 == '00000100' then SEE POP;
if P == '0' && W == '0' then UNDEFINED;
```



```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

Assembler syntax

`LDR<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`LDR<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`LDR<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rt>` Specifies the destination register. This register is allowed to be the SP. It is also allowed to be the PC, provided the instruction is either outside an IT block or the last instruction of an IT block. If it is the PC, it causes a branch to the address (data) loaded into the PC.

`<Rn>` Specifies the base register. This register is allowed to be the SP. If this register is the PC, see *LDR (literal)* on page A7-289.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-124 for encoding T1, multiples of 4 in the range 0-1020 for encoding T2, any value in the range 0-4095 for encoding T3, and any value in the range 0-255 for encoding T4. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A7.7.43 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. See *Memory accesses* on page A7-214 for information about memory accesses.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 All versions of the Thumb instruction set.

```
LDR<c> <Rt>,<label>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rt			imm8							

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

Encoding T2 ARMv7-M

```
LDR<c>.W <Rt>,<label>
LDR<c>.W <Rt>,[PC,#-0]                                  Special case
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	U	1	0	1	1	1	1	1	Rt				imm12											

```
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
LDR<c><q>  <Rt>, <label>                    Normal form
LDR<c><q>  <Rt>, [PC, #+/-<imm>]            Alternative form
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rt>` The destination register. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded into the PC.

`<label>` The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the PC value of this instruction to the label. Permitted values of the offset are:

Encoding T1 multiples of four in the range 0 to 1020

Encoding T2 any value in the range -4095 to 4095.

If the offset is zero or positive, `imm32` is equal to the offset and `add == TRUE`.

If the offset is negative, `imm32` is equal to minus the offset and `add == FALSE`. Negative offset is not available in encoding T1.

Note

In code examples in this manual, the syntax `=<value>` is used for the label of a memory word whose contents are constant and equal to `<value>`. The actual syntax for such a label is assembler-dependent.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax* on page A4-127.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,4];
    if t == 15 then
        if address<1:0> == '00' then LoadWritePC(data);  else UNPREDICTABLE;
    else
        R[t] = data;
```

Exceptions

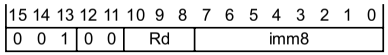
UsageFault, MemManage, BusFault.

A7.7.75 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 All versions of the Thumb instruction set.

`MOVS <Rd>,#<imm8>`	Outside IT block.
`MOV<c> <Rd>,#<imm8>`	Inside IT block.

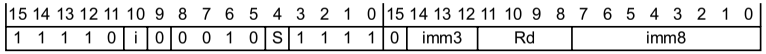


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Encoding T2 ARMv7-M

```
MOV{S}<c>.W <Rd>,#<const>
```

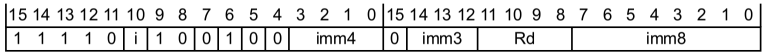


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	1	1	1	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
MOVW<c> <Rd>,#<imm16>
```



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```


Assembler syntax

`MOV{S}<c><q> <Rd>, #<const>`	All encodings permitted
`MOVW<c><q> <Rd>, #<const>`	Only encoding T3 permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rd>` Specifies the destination register.

`<const>` Specifies the immediate value to be placed in `<Rd>`. The range of allowed values is 0-255 for encoding T1 and 0-65535 for encoding T3. See *Modified immediate constants in Thumb instructions* on page A5-166 for the range of allowed values for encoding T2.

When both 32-bit encodings are available for an instruction, encoding T2 is preferred to encoding T3 (if encoding T3 is required, use the `MOVW` syntax).

The pre-UAL syntax `MOV<c>S` is equivalent to `MOVS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A7.7.76 MOV (register)

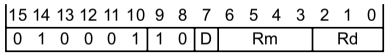
Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M

If <Rd> and <Rm> both from R0-R7, otherwise all versions of the Thumb instruction set.

If <Rd> is the PC, must be outside or last in IT block

```
MOV<c> <Rd>,<Rm>
```



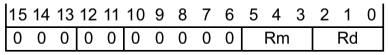
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T2 All versions of the Thumb instruction set.

```
MOVS <Rd>,<Rm>
```

Not permitted inside IT block

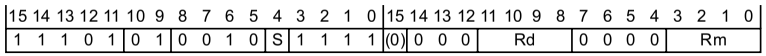


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
if InITBlock() then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
MOV{S}<c>.W <Rd>,<Rm>
```



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	0	S	1	1	1	1	(0)	0	0	0	Rd				0	0	0	0	Rm			

```
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
if setflags && (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
if !setflags && (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
```


Assembler syntax

```
MOV{S}<c><q>  <Rd>, <Rm>
```

where:

S — If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> — See *Standard assembler syntax fields* on page A7-207.

<Rd> — The destination register. This register can be the SP or PC, provided S is not specified.

If <Rd> is the PC, then only encoding T1 is permitted, and the instruction causes a branch to the address moved to the PC. The instruction must either be outside an IT block or the last instruction of an IT block.

<Rm> — The source register. This register can be the SP or PC.The instruction must not specify S if <Rm> is the SP or PC.

Encoding T3 is not permitted if either:

- <Rd> or <Rm> is the PC
- both <Rd> and <Rm> are the SP.

Note

ARM deprecates the use of the following MOV (register) instructions:

- ones in which <Rd> is the SP or PC and <Rm> is also the SP or PC is deprecated.
- ones in which S is specified and <Rm> is the SP, or <Rm> is the PC.

The pre-UAL syntax MOV<c>S is equivalent to MOVS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[m];
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            // APSR.C unchanged
            // APSR.V unchanged
```

Exceptions

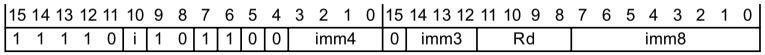
None.

A7.7.78 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

Encoding T1 ARMv7-M

```
MOVT<c> <Rd>,#<imm16>
```




```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MOVT<c><q>  <Rd>, #<imm16>
```

where:

<c><q> See *Standard assembler syntax fields* on page A7-207.

<Rd> Specifies the destination register.

<imm16> Specifies the immediate value to be written to <Rd>. It must be in the range 0-65535.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

Exceptions

None.

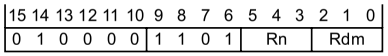
A7.7.83 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

It can optionally update the condition flags based on the result. This option is limited to only a few forms of the instruction in the Thumb instruction set, and use of it will adversely affect performance on many processor implementations.

Encoding T1 All versions of the Thumb instruction set.

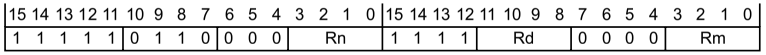
```
MULS <Rdm>,<Rn>,<Rdm>          Outside IT block.
MUL<c> <Rdm>,<Rn>,<Rdm>        Inside IT block.
```




```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

Encoding T2 ARMv7-M

```
MUL<c> <Rd>,<Rn>,<Rm>
```




```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
MUL{S}<c><q>  {<Rd>,} <Rn>, <Rm>
```

where:

S — If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> — See *Standard assembler syntax fields* on page A7-207.

<Rd> — Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

<Rn> — Specifies the register that contains the first operand.

<Rm> — Specifies the register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

Exceptions

None.

A7.7.98 POP

Pop Multiple Registers loads a subset, or possibly all, of the general-purpose registers R0-R12 and the PC or the LR from the stack.

If the registers loaded include the PC, the word loaded for the PC is treated as a branch address or an exception return value. Bit<0> complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit<0> is 0, a UsageFault exception occurs.

Encoding T1 — All versions of the Thumb instruction set.

```
POP<c> <registers>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register_list							

```
registers = P:'0000000':register_list;  if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 — ARMv7-M

```
POP<c>.W <registers>
```

<registers> contains more than one register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	0	1	1	1	1	0	1	P	M	(0)	register_list												

```
registers = P:M:'0':register_list;
if BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding T3 — ARMv7-M

```
POP<c>.W <registers>
```

<registers> contains one register, <Rt>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	1	1	1	0	1	Rt				1	0	1	1	0	0	0	0	0	1	0	0

```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t == 13 || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

Assembler syntax

```
POP<c><q>  <registers>                    Standard syntax
LDMIA<c><q>  SP!, <registers>             Equivalent LDM syntax
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<registers>`

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The registers are loaded in sequence, the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address. If the PC is specified in the register list, the instruction causes a branch to the address (data) loaded into the PC.

If the list contains more than one register, the instruction is assembled to encoding T1 or T2. If the list contains exactly one register, the instruction is assembled to encoding T1 or T3.

The SP cannot be in the list.

If the PC is in the list, the LR must not be in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;

    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    SP = SP + 4*BitCount(registers);
```

Exceptions

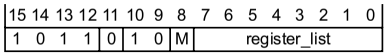
UsageFault, MemManage, BusFault.

A7.7.99 PUSH

Push Multiple Registers stores a subset, or possibly all, of the general-purpose registers R0-R12 and the LR to the stack.

Encoding T1 All versions of the Thumb instruction set.

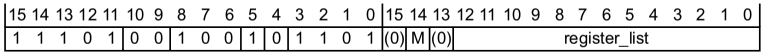
```
PUSH<c> <registers>
```




```
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Encoding T2 ARMv7-M

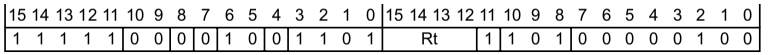
```
PUSH<c>.W <registers>                    <registers> contains more than one register
```




```
registers = '0':M:'0':register_list;
if BitCount(registers) < 2 then UNPREDICTABLE;
```

Encoding T3 ARMv7-M

```
PUSH<c>.W <registers>                    <registers> contains one register, <Rt>
```




```
t = UInt(Rt);  registers = Zeros(16);  registers<t> = '1';
if t IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`PUSH<c><q>  <registers>`	Standard syntax
`STMDB<c><q>  SP!, <registers>`	Equivalent `STM` syntax

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<registers>` Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored. The registers are stored in sequence, the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address.

If the list contains more than one register, the instruction is assembled to encoding T1 or T2. If the list contains exactly one register, the instruction is assembled to encoding T1 or T3.

The SP and PC cannot be in the list.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            MemA[address,4] = R[i];
            address = address + 4;

    SP = SP - 4*BitCount(registers);
```

Exceptions

UsageFault, MemManage, BusFault.

A7.7.158 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A7-214 for information about memory accesses.

Encoding T1 All versions of the Thumb instruction set.

```
STR<c> <Rt>, [<Rn>{,#<imm5>}]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb instruction set.

```
STR<c> <Rt>,[SP,#<imm8>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt			imm8							

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T3 ARMv7-M

```
STR<c>.W <Rt>,[<Rn>,#<imm12>]
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	1	1	0	0	Rn				Rt				imm12											

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
STR<c> <Rt>,[<Rn>,#-<imm8>]
STR<c> <Rt>,[<Rn>],#+/-<imm8>
STR<c> <Rt>,[<Rn>,#+/-<imm8>]!
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	1	0	0	Rn				Rt				1	P	U	W	imm8							

```
if P == '1' && U == '1' && W == '0' then SEE STRT;
if Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100' then SEE PUSH;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

Assembler syntax

`STR<c><q>  <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`STR<c><q>  <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`STR<c><q>  <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rt>` Specifies the source register. This register is allowed to be the SP.

`<Rn>` Specifies the base register. This register is allowed to be the SP.

`+/-` Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.

`<imm>` Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Allowed values are multiples of 4 in the range 0-124 for encoding T1, multiples of 4 in the range 0-1020 for encoding T2, any value in the range 0-4095 for encoding T3, and any value in the range 0-255 for encoding T4. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = R[t];
    if wback then R[n] = offset_addr;
```

Exceptions

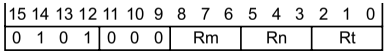
UsageFault, MemManage, BusFault.

A7.7.159 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits. See *Memory accesses* on page A7-214 for information about memory accesses.

Encoding T1 All versions of the Thumb instruction set.

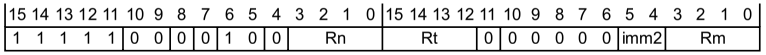
`STR<c> <Rt>,[<Rn>,<Rm>]`




```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

`STR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]`




```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if t == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
STR<c><q>  <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

where:

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rt>` Specifies the source register. This register is allowed to be the SP.

`<Rn>` Specifies the register that contains the base value. This register is allowed to be the SP.

`<Rm>` Contains the offset that is shifted left and added to the value of `<Rn>` to form the address.

`<shift>` Specifies the number of bits the value from `<Rm>` is shifted left, in the range 0-3. If this option is omitted, a shift by 0 is assumed and both encodings are permitted. If this option is specified, only encoding T2 is permitted.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    data = R[t];
    MemU[address,4] = data;
```

Exceptions

UsageFault, MemManage, BusFault.

A7.7.171 SUB (immediate)

Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

```
SUBS <Rd>,<Rn>,#<imm3>                    Outside IT block.
SUB<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb instruction set.

```
SUBS <Rdn>,#<imm8>                        Outside IT block.
SUB<c> <Rdn>,#<imm8>                      Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn							imm8			

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

```
SUB{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
SUBW<c> <Rd>,<Rn>,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	Rn				0	imm3			Rd				imm8							

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

`SUB{S}<c><q> {<Rd>,} <Rn>, #<const>`	All encodings permitted
`SUBW<c><q> {<Rd>,} <Rn>, #<const>`	Only encoding T4 permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is the same as `<Rn>`.

`<Rn>` Specifies the register that contains the first operand. If the SP is specified for `<Rn>`, see *SUB (SP minus immediate)* on page A7-499. If the PC is specified for `<Rn>`, see *ADR* on page A7-229.

`<const>` Specifies the immediate value to be subtracted from the value obtained from `<Rn>`. The range of allowed values is 0-7 for encoding T1, 0-255 for encoding T2 and 0-4095 for encoding T4. See *Modified immediate constants in Thumb instructions* on page A5-166 for the range of allowed values for encoding T3.

When multiple encodings of the same length are available for an instruction, encoding T3 is preferred to encoding T4 (if encoding T4 is required, use the `SUBW` syntax). Encoding T1 is preferred to encoding T2 if `<Rd>` is specified and encoding T2 is preferred to encoding T1 if `<Rd>` is omitted.

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

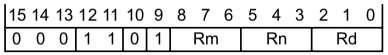
A7.7.172 SUB (register)

Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1

All versions of the Thumb instruction set.

`SUBS <Rd>,<Rn>,<Rm>`	Outside IT block.
`SUB<c> <Rd>,<Rn>,<Rm>`	Inside IT block.

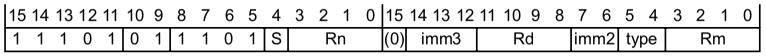



```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2

ARMv7-M

`SUB{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`




```
if Rd == '1111' && S == '1' then SEE CMP (register);
if Rn == '1101' then SEE SUB (SP minus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler syntax

```
SUB{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}
```

where:

S — If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

<c><q> — See *Standard assembler syntax fields* on page A7-207.

<Rd> — Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.

<Rn> — Specifies the register that contains the first operand. If the SP is specified for <Rn>, see *SUB (SP minus register)* on page A7-501.

<Rm> — Specifies the register that is optionally shifted and used as the second operand.

<shift> — Specifies the shift to apply to the value read from <Rm>. If <shift> is omitted, no shift is applied and both encodings are permitted. If <shift> is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A7-212.

The pre-UAL syntax SUB<c>S is equivalent to SUBS<c>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A7.7.173 SUB (SP minus immediate)

Subtract (SP minus immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

Encoding T1 — All versions of the Thumb instruction set.

```
SUB<c> SP,SP,#<imm7>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

Encoding T2 — ARMv7-M

```
SUB{S}<c>.W <Rd>,SP,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMP (immediate);
d = UInt(Rd);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 15 && S == '0' then UNPREDICTABLE;
```

Encoding T3 — ARMv7-M

```
SUBW<c> <Rd>,SP,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Assembler syntax

`SUB{S}<c><q> {<Rd>,} SP, #<const>`	All encodings permitted
`SUBW<c><q> {<Rd>,} SP, #<const>`	Only encoding T4 permitted

where:

`S` If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

`<c><q>` See *Standard assembler syntax fields* on page A7-207.

`<Rd>` Specifies the destination register. If `<Rd>` is omitted, this register is SP.

`<const>` Specifies the immediate value to be added to the value obtained from SP. Allowed values are multiples of 4 in the range 0-508 for encoding T1 and any value in the range 0-4095 for encoding T3. See *Modified immediate constants in Thumb instructions* on page A5-166 for the range of allowed values for encoding T2.

When both 32-bit encodings are available for an instruction, encoding T2 is preferred to encoding T3 (if encoding T3 is required, use the `SUBW` syntax).

The pre-UAL syntax `SUB<c>S` is equivalent to `SUBS<c>`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A7.7.175 SVC

The Supervisor Call instruction generates a call to a system supervisor. For more information see *ARMv7-M exception model* on page B1-631.

Use it as a call to an operating system to provide a service.

Note

In older versions of the ARM architecture, `SVC` was called `SWI`, Software Interrupt.

Encoding T1

All versions of the Thumb instruction set.

`SVC<c> #<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

A5.3.2 Modified immediate constants in Thumb instructions

The encoding of modified immediate constants in Thumb instructions is:

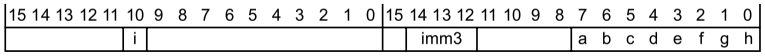


Table A5-11 shows the range of modified immediate constants available in Thumb data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table A5-11 Encoding of modified immediates in Thumb data-processing instructions

i:imm3:a	<const> [a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh [b]
0010x	abcdefgh 00000000 abcdefgh 00000000 [b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh [b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000
.	.
.	. 8-bit values shifted to other positions
.	.
11101	00000000 00000000 000001bc defgh000
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0

a. In this table, the immediate constant value is shown in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).
b. UNPREDICTABLE if abcdefgh == 00000000.

Carry out

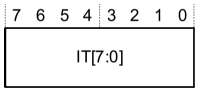
A logical operation with `i:imm3:a` of the form `00xxx` does not affect the carry flag. Otherwise, a logical operation that sets the flags sets the Carry flag to the value of bit [31] of the modified immediate constant.

Operation of modified immediate constants

```
// ThumbExpandImm()
// ================

bits(32) ThumbExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = ThumbExpandImm_C(imm12, APSR.C);

    return imm32;

// ThumbExpandImm_C()
// ==================

(bits(32), bit) ThumbExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then

        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                if imm12<7:0> == '00000000' then UNPREDICTABLE;
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;

    else

        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

A7.3.3 ITSTATE

The bit assignments of the ITSTATE field are:



This field holds the If-Then execution state bits for the Thumb `IT` instruction. See *IT* on page A7-277 for a description of the `IT` instruction and the associated IT block.

ITSTATE divides into two subfields:

IT[7:5] Holds the *base condition* for the current IT block. The base condition is the top 3 bits of the condition specified by the IT instruction.

This subfield is 0b000 when no IT block is active.

IT[4:0] Encodes:

- The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is implied by the position of the least significant 1 in this field, as shown in Table A7-2 on page A7-211.
- The value of the least significant bit of the condition code for each instruction in the block.

  Note

  Changing the value of the least significant bit of a condition code from 0 to 1 has the effect of inverting the condition code.

  This subfield is 0b00000 when no IT block is active.

When an IT instruction is executed, these bits are set according to the condition in the instruction, and the *Then* and *Else* (T and E) parameters in the instruction, see *IT* on page A7-277 for more information.

An instruction in an IT block is conditional, see *Conditional instructions* on page A4-126. The condition used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, `ITSTATE` is advanced to the next line of Table A7-2 on page A7-211.

See *Exception entry behavior* on page B1-643 for details of what happens if such an instruction takes an exception.

Note

Instructions that can complete their normal execution by branching are only permitted in an IT block as its last instruction, and so always result in `ITSTATE` advancing to normal execution.

Table A7-2 Effect of IT execution state bits

IT bits [a]						
[7:5]	**[4]**	**[3]**	**[2]**	**[1]**	**[0]**	
cond_base	P1	P2	P3	P4	1	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	1	0	Entry point for 3-instruction IT block
cond_base	P1	P2	1	0	0	Entry point for 2-instruction IT block
cond_base	P1	1	0	0	0	Entry point for 1-instruction IT block
000	0	0	0	0	0	Normal execution, not in an IT block

a. Combinations of the IT bits not shown in this table are reserved.

Pseudocode details of ITSTATE operation

`ITSTATE` advances after normal execution of an IT block instruction. This is described by the `ITAdvance()` pseudocode function:

```
// ITAdvance()
// ===========

ITAdvance()
    if ITSTATE<2:0> == '000' then
        ITSTATE.IT = '00000000';
    else
        ITSTATE.IT<4:0> = LSL(ITSTATE.IT<4:0>, 1);
```

The following functions test whether the current instruction is in an IT block, and whether it is the last instruction of an IT block:

```
// InITBlock()
// ===========

boolean InITBlock()
    return (ITSTATE.IT<3:0> != '0000');

// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (ITSTATE.IT<3:0> == '1000');
```

Pseudocode details of addition and subtraction

In pseudocode, addition and subtraction can be performed on any combination of unbounded integers and bitstrings, provided that if they are performed on two bitstrings, the bitstrings must be identical in length. The result is another unbounded integer if both operands are unbounded integers, and a bitstring of the same length as the bitstring operand(s) otherwise. For the precise definition of these operations, see *Addition and subtraction* on page AppxF-971.

The main addition and subtraction instructions can produce status information about both unsigned carry and signed overflow conditions. This status information can be used to synthesize multi-word additions and subtractions. In pseudocode the `AddWithCarry()` function provides an addition with a carry input and carry and overflow outputs:

```
// AddWithCarry()
// ==============

(bits(N), bit, bit) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    signed_sum   = SInt(x) + SInt(y) + UInt(carry_in);
    result       = unsigned_sum<N-1:0>;  // same value as signed_sum<N-1:0>
    carry_out    = if UInt(result) == unsigned_sum then '0' else '1';
    overflow     = if SInt(result) == signed_sum then '0' else '1';
    return (result, carry_out, overflow);
```

An important property of the `AddWithCarry()` function is that if:

```
(result, carry_out, overflow) = AddWithCarry(x, NOT(y), carry_in)
```

then:

- If `carry_in == '1'`, then `result == x-y` with `overflow == '1'` if signed overflow occurred during the subtraction and `carry_out == '1'` if unsigned borrow did not occur during the subtraction (that is, if `x >= y`).
- If `carry_in == '0'`, then `result == x-y-1` with `overflow == '1'` if signed overflow occurred during the subtraction and `carry_out == '1'` if unsigned borrow did not occur during the subtraction (that is, if `x > y`).

Together, these mean that the `carry_in` and `carry_out` bits in `AddWithCarry()` calls can act as *NOT borrow* flags for subtractions as well as *carry* flags for additions.