UFR IM²AG

UNIVERSITÉ Grenoble Alpes

GBIN5U03 - Projet Logiciel
Réalisation d'un simulateur ARM
Année 2020-2021

L'objectif de ce projet est d'écrire, à partir d'un squelette de code fourni, un simulateur d'un sous-ensemble du jeu d'instructions ARMv5T. Ce jeu d'instructions est décrit dans un manuel de référence téléchargeable sur le site de la compagnie : `www.arm.com`, rubrique `resources/ARM developer site/documentation` puis rechercher ARMv5 pour obtenir un lien vers le manuel de référence.

1 Principe général

Le programme que l'on souhaite construire doit permettre d'exécuter du code machine respectant le jeu d'instructions ARMv5 sur une machine hôte avec un autre jeu d'instructions. Dans le cadre du projet, la machine hôte sera basée sur l'architecture Intel x86 (32 bits) ou x86-64 (64 bits), mais, si le code du simulateur est écrit de manière portable, celui-ci pourra être compilé et exécuté sur d'autres architectures [1].

Pour simplifier la mise en œuvre du projet, on s'appuie sur le debugger `gdb` dans son mode client-serveur (plus précisément `arm-none-eabi-gdb`, puisque l'on s'intéresse à du code exécutable pour machine ARM). `arm-none-eabi-gdb` va nous permettre de faire essentiellement deux choses :

- lire un fichier executable au format ELF, en extraire les différentes sections et les charger dans la mémoire du simulateur
- interagir avec le simulateur, notamment en permettant une execution pas-à-pas et des entrées/-sorties avec la mémoire et les registres

Dans le mode client-serveur de `gdb`, on distingue deux entités :

- la cible, ou serveur `gdb`, qui exécute un programme P en suivant les ordres du client ;
- le client `gdb`, qui interagit avec l'utilisateur et, en fonction de ses demandes, envoie des requêtes à la cible pour contrôler/observer l'exécution du programme P.

Ici, le serveur et le client correspondent respectivement au simulateur ARM et au programme interactif `arm-none-eabi-gdb`. Les points importants du protocole d'interaction entre client et serveur sont décrits en section 2.3.

Le client et le serveur interagissent via un canal de communication. Dans ce projet, il s'agit de canal de communication réseau (sockets TCP/IP, qui seront étudiées au deuxième semestre). Le client et le serveur peuvent donc être lancés sur la même machine ou sur des machines différentes.

Ainsi, après l'exécution de chaque instruction d'un programme chargé sur la cible, on doit pouvoir inspecter l'état de la machine simulée afin d'observer son état courant, c'est-à-dire le contenu des registres ainsi que de la mémoire. Grâce à cela vous pourrez tester le bon fonctionnement de votre simulateur.

1. Dans ce but, il est notamment recommandé d'utiliser les types entiers de taille fixe (tels que `int32_t, uint32_t`, etc.) définis par la norme C99, et par ailleurs d'être vigilant sur le boutisme de certaines informations.

2 Squelette fourni

Le squelette qui vous est fourni contient une gestion à compléter de l'état de la machine simulée (registres et mémoire), une interface pour piloter le simulateur depuis gdb, quelques exemples de code ARM à simuler et une commande permettant d'envoyer un signal d'interruption au simulateur.

2.1 Compilation

Pour compiler le projet initial qui vous est fourni il vous faut effectuer les actions suivantes :

- décompresser l'archive fournie
  `tar xvzf arm_simulator-1.4.tar.gz`
- vous rendre dans le répertoire `arm_simulator-1.4` créé et exécuter le script de configuration
  `cd arm_simulator-1.4`
  `./configure CFLAGS='-Wall -Werror -g'`
  Ici, on précise les `CFLAGS` pour éviter que le script de configuration n'y ajoute `-O2`, ce qui empêche de déboguer, et pour y ajouter `-Wall -Werror`, ce qui ajoute des avertissements/erreurs supplémentaires.
- l'exécution avec succès du script de configuration provoque la création des fichiers `Makefile` qui vous serviront à compiler le projet ; il reste juste à les utiliser :
  `make`
- à l'issue de toutes ces étapes, vous devriez obtenir l'exécutable principal du simulateur :
  `./arm_simulator`

Ce simulateur accepte un certain nombre d'options en ligne de commande brièvement présentées par l'option `-help`. **Attention :** ce simulateur ne fonctionne pas. Tant que vous n'aurez pas complété les fichiers `registers.c` et `memory.c`, il sera incapable de charger un code fourni par le client gdb.

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

```
make distclean
autoreconf
```

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

Dans ce projet vous aurez à modifier et compléter les fichiers source (les `.h` et les `.c`) présents dans le répertoire principal du simulateur. Si vous n'ajoutez pas de fichier, cela ne demande pas de toucher au `Makefile.am`. Vous aurez également à ajouter de nouveaux exemples en assembleur ARM dans le répertoire `Examples`. Ils seront automatiquement compilés une fois que vous les aurez ajoutés au `Makefile.am` du répertoire `Examples` (inspirez vous des exemples déjà présents).

2.2 Utilisation

Comme expliqué dans l'introduction, le simulateur s'utilise comme un serveur piloté par les ordres du client `arm-none-eabi-gdb`. Dans le cadre du projet, il n'est pas demandé de développer un simulateur autonome, c'est-à-dire capable de fonctionner sans le client `arm-none-eabi-gdb` [2]. Pour démarrer une simulation vous devez donc ouvrir deux terminaux pour lancer deux programmes distincts :

- le simulateur lui-même (`./arm_simulator`) qui commence par afficher les numéros de deux ports sur lesquels il attend l'arrivée de connexions TCP. Pour vous simplifier la vie vous pouvez fixer ces deux ports d'écoute ; invoquez le simulateur avec l'option `-help` pour plus de détails.
- un client gdb (`arm-none-eabi-gdb`) qui va nous servir à piloter le simulateur via une connexion TCP.

Le terminal dans lequel s'exécute le simulateur servira à l'affichage de l'état interne de celui-ci et du progrès de la simulation, tandis que le terminal dans lequel s'exécute gdb servira à envoyer des ordres

2. Cela demanderait la lecture du format binaire, typiquement ELF, utilisé pour le code simulé.

au simulateur. Au préalable, il va falloir effectuer quelques étapes d'initialisation à partir du terminal de gdb :

- indiquer sur quel fichier en langage machine ARM nous souhaitons travailler, par exemple
  `file Examples/example1`
- connecter le client gdb au serveur simulateur
  `target remote localhost:<port indiqué par le simulateur>`
- charger le code en langage machine dans le simulateur
  `load`
  Attention : le simulateur qui vous est fourni initialement n'implémente pas la gestion des registres et de la mémoire. Cette étape échouera donc tant que vous n'aurez pas réalisé la toute première partie du travail qui vous est demandé.

Une fois ces premières étapes accomplies, vous pouvez exécuter le programme depuis gdb de la manière usuelle :

- placement de points d'arrêts, à l'aide de la commande `break` ;
- exécution pas-à-pas, avec la commande `stepi` (avancée d'une instruction).
  Attention : le simulateur qui vous est fourni initialement n'implémente pas l'étape de chargement de la prochaine instruction et d'incrémentation du compteur ordinal. L'exécution n'avancera donc pas en mémoire tant que vous n'aurez pas réalisé la toute première partie du travail de décodage des instructions (typiquement en utilisant `arm_fetch`) ;
- exécution continue jusqu'à un point d'arrêt avec la commande `cont` ;
- visualisation du contenu des registres à l'aide de la commande `info registers` ou `print $nom` ;
- visualisation du contenu de la mémoire à l'aide de la commande `print` (à laquelle il est possible de fournir une étiquette par exemple) ;
- ...

Il est à noter que, dans le squelette fourni, la simulation se termine lors de l'exécution de l'instruction `SWI 0X123456`. En conséquence, cela doit être la dernière instruction de tous les programmes d'exemple que vous écrirez. Dans le cas contraire le simulateur continue l'exécution des instructions au delà de la dernière véritable instruction du programme et les conséquences sont indéterminées (boucle infinie, erreur de segmentation, ...). Par ailleurs, il est possible d'arrêter l'exécution d'un programme à n'importe quelle étape via la commande `kill` de `gdb`.

2.3 Interface avec gdb

Lors de son démarrage, le simulateur ARM exécute une procédure (nommée `gdb_listener` jouant le rôle du serveur `gdb`. Cette procédure exécute une boucle qui effectue les étapes suivantes : récupération d'une requête du client, traitement de la requête puis envoi d'une réponse au client. Les requêtes et les réponses doivent respecter des conventions de format bien précises qui ne seront pas détaillées ici (cet aspect est pris en charge par le code fourni dans les fichiers `gdb_protocol.h` et `gdb_protocol.c`).

Les requêtes peuvent être de différents types. Les requêtes qui nous intéressent dans ce projet concernent deux aspects principaux : (i) la consultation et la modification de l'état de la cible (contenu des registres et de la mémoire centrale) et (ii) la progression de l'exécution du programme chargé sur la cible. Le serveur `gdb` doit notamment fournir un ensemble de *traitants* (définis dans `gdb_protocol.c`) pour les requêtes suivantes [3] :

consultation de registre(s) Demande de lecture du contenu actuel d'un registre précis ou de l'ensemble des registres. Les traitants `read_register` et `read_general_registers` appellent les procédures `arm_read_register` et `arm_read_cpsr` définies dans `arm_core.c`. Vous allez devoir écrire ces procédures.

modification de registre(s) Demande de modification du contenu actuel d'un registre précis ou de l'ensemble des registres. Les traitants `write_register` et `write_general_registers` appellent les procédures `arm_write_register` et `arm_write_cpsr` définies dans `arm_core.c`. Vous allez devoir écrire ces procédures.

3. Remarque : la liste de requêtes/traitants donnée ici est incomplète. Seules les requêtes ayant un lien avec le travail d'implémentation demandé dans le projet sont mentionnées.

consultation de la mémoire Demande de lecture des octets associés à une plage d'adresses. Le traitant `read_memory` appelle la procédure `memory_read_byte` définie dans `memory.c`. Vous allez devoir écrire cette procédure.

modification de la mémoire Demande d'écriture des octets associés à une plage d'adresses. Le traitant `write_memory` appelle la procédure `memory_write_byte` définie dans `memory.c`. Vous allez devoir écrire cette procédure.

step Exécution de l'instruction suivante sur la cible. Le traitant `step` appelle la procédure `arm_step`, qui appelle elle-même la procédure `arm_execute_instruction` (cf. `arm_instruction.c`). Vous allez devoir écrire `arm_execute_instruction`.

cont Poursuite de l'exécution du programme sur la cible, jusqu'à ce qu'elle atteigne un point d'arrêt ou la fin du programme. Le traitant `cont` appelle la procédure `arm_step` ainsi que plusieurs procédures d'accès aux registres du processeur simulé.

2.4 Etat de la machine simulée

Le premier travail du simulateur est de stocker et faire évoluer l'état de la machine simulée. Cet état est constitué des registres du processeur ainsi que de la mémoire à laquelle le processeur pourra accéder.

2.4.1 Registres

Vous allez devoir choisir comment stocker les registres d'un processeur ARMv5 et implémenter une interface permettant d'y accéder en complétant le fichier `arm_core.c`. Ce fichier fournit des primitives pour :

— lire le contenu d'un registre généraliste ou d'un registre d'état (voir notamment `arm_read_register` et `arm_read_cpsr`) ;
— modifier le contenu d'un registre généraliste ou d'un registre d'état (voir notamment `arm_write_register` et `arm_write_cpsr`).
— afficher le contenu actuel des registres (voir `arm_print_state`).

Les registres précis manipulés par ces fonctions dépendent du mode d'exécution courant du processeur. Dans un premier temps, vous n'aurez pas à vous préoccuper du mode d'exécution du processeur, qui sera `USR` uniquement. Ce mode d'exécution sera amené à varier si vous vous attaquez au support des exceptions dans le processeur.

2.4.2 Mémoire

Vous allez devoir choisir comment stocker la mémoire du processeur simulé et implémenter une interface permettant d'y accéder en complétant le fichier `memory.c`. L'ensemble de fonctions qui s'y trouve permet d'accéder à la mémoire à différents niveaux de granularité :

— octet (`memory_read_byte` et `memory_write_byte`) ;
— demi-mot de 16 bits (`memory_read_half` et `memory_write_half`) ;
— mot de 32 bits (`memory_read_word` et `memory_write_word`).

Les détails d'implémentation se trouvent dans le fichier `memory.c`. Le fichier `arm_core.c` fournit une surcouche pour les fonctions ci-dessus, qui ne sera pas à modifier : `arm_read_byte`, `arm_write_byte`, `arm_read_byte`, `arm_read_half`, etc. Le rôle de cette surcouche est essentiellement d'intégrer un système de traçage des accès mémoire d'un programme ARM.

Remarque : Pour les accès mémoire sur 16 bits et 32 bits, les processeurs ARM peuvent être configurés pour utiliser la convention de boutisme *big endian* ou bien la convention *little endian*. Dans le cadre du projet, on utilisera la convention *big endian*, activée par le squelette fourni. Il est à noter que le boutisme d'une machine simulée peut être différent de celui de la machine hôte (qui exécute le simulateur). Cette situation se présente d'ailleurs dans le cas de ce projet : on simule un processeur ARM en *big endian* sur une architecture x86 en *little endian*. Il sera necessaire d'en tenir compte lors de l'écriture des fonctions d'accès à la mémoire (dans `memory.c`).

2.5 Système de traces

Le simulateur initialement fourni contient un système permettant de tracer les accès à la mémoire et/ou aux registres afin de déboguer le comportement du simulateur. Ces traces peuvent être simplement affichées à l'écran ou sauvegardées dans un fichier. Ce système est paramétré par l'ensemble d'options suivant, qu'il est possible de donner à la commande `arm_simulator` :

- **--trace-file fichier** : permet de stocker les information de trace dans un fichier plutôt que de les afficher à l'écran ;
- **--trace-registers** : trace l'ensemble des accès aux registres ;
- **--trace-memory** : trace l'ensemble des accès à la mémoire ;
- **--trace-state** : trace l'état complet du processeur après chaque instruction ;
- **--trace-position** : ajoute aux informations de trace le fichier et la ligne auxquels la fonction d'accès a été appelée.

Les exemples qui vous sont fournis sont accompagnés de fichiers de trace obtenus en les simulant avec un simulateur complet exécuté avec les options `--trace-registers` et `--trace-memory`. Ces fichiers de trace peuvent vous servir à tester la correction du décodage de vos premières instructions en les comparant à la trace produite par votre implémentation (souvenez vous de la commande `diff`).

Votre implémentation pouvant différer de l'implémentation ayant servi à générer les traces fournies (ordre des accès, multiples accès identiques si le résultat n'est pas stocké, ...), vous pouvez *normaliser* les traces obtenues en les triant (commande `sort`) et en éliminant les lignes en double (commande `uniq`) avant de les comparer. Après un tel traitement, les traces obtenues ne correspondent plus à un comportement réel du processeur (ni même correct), mais permettent, dans certains cas, de comparer deux implémentations distinctes de manière automatique.

2.6 Système de déboguage

Le simulateur initialement fourni vous offre un système de messages de déboguage activable selectivement pour chacun des fichiers source grâce à l'option `-debug` de la ligne de commande. Ce système est constitué de deux fonctions : `debug` et `debug_raw` qui s'utilisent comme `printf`. La différence entre `debug` et `debug_raw` est que la première ajoute en début de ligne la chaîne de caractères `"[DEBUG]"` suivie du nom de fichier dans lequel elle se trouve ainsi de le numéro de ligne. Ces fonctions ne produisent un affichage que si l'option `-debug` suivie du nom de fichier dans lequel elles se trouvent est fournie sur la ligne de commande.

3 Travail demandé

Le travail demandé consiste à compléter le squelette fourni afin d'implémenter un simulateur fonctionnel pour un sous-ensemble du jeu d'instructions ARMv5T. La liste des instructions à prendre en compte est donnée en annexe. Une fois les instructions demandées initialement simulées correctement, vous pourrez vous attaquer à la gestion des interruptions et des modes d'exécution autres que le mode `USR`.

3.1 Parties à compléter

La toute première étape du projet consiste à compléter les fonctions d'accès aux registres et à la mémoire de la machine simulée (dans les fichiers `registers.c` et `memory.c`). En effet, cette étape est nécessaire pour permettre le dialogue entre le simulateur et le client `gdb` et permettre le chargement d'un programme dans le simulateur. Un programme, `memory_test.c` vous est fourni pour tester votre implémentation des fonctions d'accès à la mémoire (`make memory_test` puis `./memory_test`).

Une fois cette première étape réalisée, vous pourrez charger un programme dans le simulateur. Dans son état initial, le simulateur n'exécute rien. Votre second travail sera donc de coder l'étape de chargement d'une instruction et d'incrémentation du compteur ordinal (regardez la fonction `arm_fetch`). Ensuite seulement, vous pourrez vous attaquer au reste du décodage [4], qui consistera à implémenter le comportement correct d'autant d'instructions que possible en prenant soin de factoriser le code. Pour

4. excepté pour `swi 0x123456` qui est déjà fournie.

cela vous vous aiderez du manuel de référence du jeu d'instructions ARM accessible sur le site de la compagnie ARM.

Les instructions ARM à prendre en charge dans le simulateur peuvent globalement être regroupées selon les catégories suivantes :

- **traitement de données (*data processing*) :** opérations arithmétiques et logiques, transferts entre registres ;
- **accès à la mémoire :** opérations *load/store* entre un registre et la mémoire avec différents niveaux de granularité (mot, demi-mot, octet), transferts multiples entre un ensemble de registres et la mémoire ;
- **rupture de séquence :** branchement avec sauvegarde éventuelle de l'adresse de retour ;
- **autres instructions :** instructions ne rentrant pas dans les catégories précédentes, par exemple transfert entre le registre d'état et un registre généraliste.

3.2 Squelette fourni pour vous guider

Le squelette fourni donne une trame pour l'organisation du code de décodage et de traitement des différentes catégories d'instructions. Les fichiers concernés sont `arm_data_processing.c`, `arm_load_store.c` et `arm_branch_other.c` ainsi que le fichier pivot `arm_instruction.c`. Le fichier `arm_instruction.c` est le premier à compléter. La fonction `arm_execute_instruction` est la toute première partie de la simulation d'un cycle d'exécution d'une instruction dans le processeur. Idéalement, cette fonction devra contenir la partie commune à l'exécution de toute instruction ARM : la lecture de la prochaine instruction, l'incrémentation du compteur ordinal et l'interprétation du champ d'exécution conditionnelle. Ensuite, cette fonction devra sélectionner la classe d'instructions concernée par l'instruction courante et appeler la fonction appropriée, chargée de la suite du décodage.

L'interprétation du champ d'exécution conditionnelle et la détermination de la classe de l'instruction courante peuvent être implémentées principalement selon deux approches : bloc `switch` ou table de pointeurs de fonctions. Les listings (de pseudo-code) donnés en Figure 1 illustrent ces deux techniques de décodage, en considérant un champ de 3 bits extrait des bits 15 à 17 d'un mot de 32 bits.

```
uint_32t mot;
uint8_t champ;
...
champ = (uint8_t)((mot & 0x38000) >> 15);
switch (champ) {
    case 0: ... break;
    case 1: ... break;
    ...
    case 7: ... break;
    default: ... // cas impossible
}
...
```

```
// lors de l'initialisation, remplir un tableau
// contenant des pointeurs (de fonctions)
// vers les routines de decodage
tableau[0] = routine_pour_valeur_champ_0;
tableau[1] = routine_pour_valeur_champ_1;
...
uint_32t mot;
uint8_t champ;
...
champ = (uint8_t)((mot & 0x38000) >> 15);
// appel de la routine correspondante
tableau[champ](mot, ...);
...
```

FIGURE 1 – Pseudo-code de décodage via bloc `switch` (gauche) ou tableau de pointeurs (droite)

Afin de minimiser les difficultés lors du démarrage du projet, vous pouvez commencer par implémenter et tester la prise en charge des instructions présentes dans le jeu de tests fourni, en suivant la numérotation des tests.

4 Tests

Toutes les fonctionnalités implémentées dans le simulateur devront être validées par des tests. Pour cela, vous devrez développer des jeux de tests pertinents pour chaque cas de figure possible (type d'ins-

truction, mode d'adressage, etc.). En complément, certains programmes de tests vous sont fournis, accompagnés de traces de référence (cf. explications dans les sections précédentes).

5 Documents à produire

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

- Descriptif de la structure du code développé : principales fonctions et fichiers correspondants (inutile de décrire le squelette fourni, sauf en cas de modifications importantes) ;
- Liste des fonctionnalités implémentées et manquantes ;
- Liste des éventuels bogues connus mais non résolus ;
- Liste et description des tests effectués ;
- Journal décrivant la progression du travail et la répartition des tâches au sein du groupe (à remplir quotidiennement).

Le code du simulateur ET les documents demandés devront être rendus AVANT la soutenance selon les modalités indiquées sur la page du projet.

Annexe A : liste des principaux fichiers fournis dans le squelette

`arm.h/c` définition de l'enveloppe `arm_init()` pour l'initialisation globale du simulateur

`arm_branch_other.h/c` décodage spécialisé pour les instructions de branchement et les instructions n'appartenant pas aux catégories précédentes (**fichier(s) à compléter**)

`arm_constants.h/c` définition des différents modes d'exception et types d'exceptions

`arm_core.h/c` gestion haut niveau des registres et de la mémoire

`arm_data_processing.h/c` décodage spécialisé pour les instructions de *data processing* (**fichier(s) à compléter**)

`arm_exception.h/c` gestion des exceptions et des interruptions (**fichier(s) à compléter en dernier, après la simulation correcte des instructions à prendre en charge**)

`arm_instruction.h/c` gestion de l'exécution d'une instruction et des premières étapes de son décodage (**fichier(s) à compléter en second**)

`arm_load_store.h/c` décodage spécialisé pour les instructions d'accès à la mémoire (**fichier(s) à compléter**)

`arm_simulator.c` fichier principal du simulateur, gestion du serveur gdb, gestion du serveur pour les demandes d'interruption

`debug.h/c` primitives utiles pour le débogage (cf. 2.6)

`gdb_protocol.h/c, scanner.l/h/c` gestion du protocole gdb (il n'est pas demandé d'étudier cette partie du code).

`memory.h/c` fonctions de base pour la gestion des accès à la mémoire (**fichier(s) à compléter en tout premier**)

`registers.h/c` fonctions de base pour la gestion des accès aux registres (**fichier(s) à compléter en tout premier**)

`trace.h/c, trace_location.h` gestion des traces (cf. 2.5)

`util.h/c` fonctions utiles pour le décodage de certaines instructions

Annexe B : liste des instructions à prendre en charge

Remarque : Le simulateur doit prendre en charge les instructions ci-dessous dans tous les différents modes (modes d'adressages, flags, etc.) possibles. En cas de doute sur la nécessité de prendre en charge une fonctionnalité précise, consulter l'équipe pédagogique.

Instructions de rupture de séquence (branchement) `B/BL`

Instructions de traitement de données `AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, TST, TEQ, CMP, CMN, ORR, MOV, BIC, MVN`

Instructions d'accès à la mémoire `LDR, LDRB, LDRH, STR, STRB, STRH, LDM(1), STM(1)`

Instructions diverses `MRS`

Annexe C : extensions possibles

Pour les groupes ayant terminé l'implémentation et les tests des fonctionnalités demandées impérativement (Annexe B), différentes pistes d'extension peuvent être considérées, telles que la prise en charge d'instructions supplémentaires (par exemple, `LDRSH` et `LDRSB`) ou la prise en compte des exceptions. En cas de doute, demandez conseil à un enseignant.