README.md

Projet de Prog, L3 INFO

Compilation :

1. Se placer dans le dossier arm-simulator-1.4
2. ./configure CFLAGS='-Wall -Werror -g'
3. make

L'option CFLAGS dans l'appel à configure est optionnelle, elle est utilisée pour le débogage.

Nettoyage :

- Suppression des fichiers objets et des exécutables : make clean
- Nettoyage complet (retour à l'état d'origine) : make distclean

Utilisation:

Programmes auxiliaires :

- memory_test:
  - Compilation : `make memory_test`
  - Utilisation : `./memory_test`
  - Effet : Teste l'implémentation de la mémoire
- send_irq:
  - Compilation : `make send_irq`
  - Utilisation : `./send_irq host port [IRQ name]`
  - Effet : Envoie une requête IRQ au simulateur
  - Remarque : Nous ne sommes pas allés jusque là, donc notre simulateur ARM ne peut gérer ces requêtes.
- arm_simulator:
  - Compilation : `make arm_simulator`
  - Utilisation : Voir `./arm_simulator --help`
  - Remarques : Voir paragraphes suivants

Utilisation du Simulateur :

1. Préparation du simulateur

Pour lancer le simulateur : `./arm_simulator [options]` (voir `./arm_simulator -h` pour un descriptif des options).
L'output du programme doit ressembler à cela :

```
Listening to gdb connection on port [port_gdb]
Listening to irq connections on port [port_irq]
```

`port_irq` et `port_gdb` sont des nombres.
Dans cet état, le simulateur attend des instructions de GDB.

2. Lancement de GDB

Il faut ensuite lancer gdb dans un autre terminal : `gdb` ou `arm-none-eabi-gdb`
Dans GDB, il faut ensuite exécuter la suite de commandes suivantes:

```
file [fichier arm compilé contenant le code à exécuter]
target remote localhost:[port_gdb]
load
```

Ici, `port_gdb` est le même que dans le paragraphe précédent.

3. Exécution du programme

À partir de maintenant, vous pouvez exécuter le programme depuis GDB comme si c'était un programme classique (avec les commandes `cont` , `step` ...), les traces seront affichées dans le terminal dans lequel vous avez lancé le simulateur.

Compte-Rendu

1. Gestion des registres

Fichiers concernés : `registers.c` , `registers.h`

Nous avons choisi d'implémenter les registres comme un simple tableau de 37 entiers de 32 bits.
Chaque case du tableau correspond à un registre.
L'ordre des registres dans le tableau correspond à l'ordre de l'enum `Reg_Names` dans `registers.h` .

Quelque soit le mode d'exécution, lorsqu'on essaie d'accéder à un registre, on donne toujours le même numéro (par exemple, 13 pour R13), mais selon le mode d'exécution, on n'utilise pas le même registre (pour reprendre notre exemple, si on est en mode FIQ, et qu'on veut essayer d'accéder à R13, on n'accède pas à la case d'indice 13 du tableau, mais à celle d'indice 27).
Nous avons donc écrit une fonction qui donne l'indice dans le tableau en fonction du mode d'exécution et du numéro de registre fourni.

2. Gestion de la mémoire

Fichiers concernés : `memory.c` , `memory.h`

Nous avons implémenté la mémoire comme un tableau d'octets. L'adresse mémoire d'un octet correspond à son indice dans le tableau.
Le big-endian et le little-endian sont tous deux supportés par cette implémentation, mais les adresses non-alignées ne le sont pas.

3. Décodage des instructions

A. Les conditions d'exécution

Fichiers concernés : `arm_instructions.c` , `arm_instructions.h`

Quelle que soit l'instruction que l'on exécute, elle doit passer par une vérification de condition. Pour cela, nous avons écrit une fonction `check_cond` qui utilise les flags du CPSR pour savoir si l'instruction doit être exécutée ou non.

B. Le parseur d'instructions

Fichiers concernés : `arm_instructions.c` , `arm_instructions.h`

Après avoir passé le test conditionnel, on regarde l'instruction bit par bit pour déterminer à quelle catégorie elle appartient. Cela se fait dans la fonction `get_category_inst` .

Pour grouper les instructions par catégorie, nous nous sommes appuyés sur cette documentation
Voici la liste des catégories :

- [x] Data Processing
- [x] PSR Transfer
- [x] Multiply
- [x] Multiply Long

- [x] Single Data Swap
- [x] Branch Exchange
- [x] Halfword Data Tranfer
- [x] Single Data Transfer
- [ ] Undefined
- [x] Block Data Transfer
- [x] Branch
- [ ] Coprocessor Data Transfer
- [ ] Coprocessor Data Operation
- [ ] Coprocessor Register Transfer
- [x] Software Interrupt

Les instructions appartenant aux catégories cochées ont été implémentées.
Pour la catégorie Software Interrupt, seul le cas `swi 0x123456` a été géré, pour finir le programme et fermer proprement le simulateur. Un SWI avec un autre paramètre fera que le simulateur se fermera avec un code erreur non nul.

4. Les différentes catégories

Voici, pour chaque catégorie d'instruction implémentée, les fichiers correspondant:

- `Data Processing`, `PSR Transfer`, `Multiply` et `Multiply Long` : arm_data_processing.c, arm_data_processing.h
- `Single Data Swap`, `Halfword Data Transfer`, `Single Data Transfer`, `Block Data Transfer` : arm_load_store.c, arm_load_store.h
- `Branch Exchange`, `Branch`, `Software Interrupt` : arm_branch_other.c, arm_branch_other.h

5. Fonctionnalités manquantes

Voici les fonctionnalités demandées dans le sujet que nous n'avons pas implémenté :

- Les gestion des interruptions par les requêtes IRQ
- La gestion des exceptions. À l'heure actuelle, une exception `SOFTWARE_INTERRUPT` provoque la fermeture du simulateur avec un code erreur nul, l'exception `UNDEFINED_INSTRUCTION` ferme le simulateur avec un code erreur non nul, l'exception `RESET` est gérée comme dans le code fourni, et les autres exceptions sont ignorées. Par conséquent, il est impossible de changer de mode d'exécution.

6. Bugs connus mais non résolus

Nous n'avons connaissance d'aucun bug que nous n'avons pas résolu

7. Tests réalisés

Nous avons fait un fichier de test pour chaque catégorie d'instruction. Bien que non exhaustifs, nous estimons que ces tests couvrent une bonne partie des cas possibles.

8. Journal décrivant la progression du travail et la répartition des tâches

Ce journal est construit sur l'historique des commit. Certains membres du groupe utilisent un pseudonyme sur GitHub :

- Esdia : Théo Hermitte
- Walfyr : Leslie Metzger

Nous avons grossièrement réparti le travail comme ceci:

- Théo Hermitte : Implémentation du noyau de base (mémoire + registres + parseur d'instructions) + Data Processing
- Leslie Metzger : PSR Transfer + Single Data Swap
- Nadim Babba : Halfword Data Transfer
- Rodolphe Beguin : Single Data Transfer
- Maxime Bouchenoua : Block Data Transfer

- Thomas Delatte : Branch + Branch Exchange + Multiply + Multiply Long

Journal détaillé :

- 31/12 - Mise en place du dépôt Git, répartition du travail
- 01/01 - Implémentation de la mémoire (Théo Hermitte)
- 02/01 - Implémentation des registres (Théo Hermitte)
- 03/01 - Correction d'un problème fatal dans l'implémentation de la mémoire (Nadim Babba)
- 04/01 - Implémentation du parseur d'instructions + quelques fonctions utilitaires (get_flag, set_flag..) (Théo Hermitte)
- 04/01 - Implémentation des instructions de data processing (Théo Hermitte)
- 06/01 - Implémentation des instructions de halfword data transfer (Nadim Babba)
- 06/01 - Implémentation des instructions de PSR Transfer (Leslie Metzger)
- 07/01-08/01 - Corrections de bugs et modifications mineures dans le parseur d'instructions (Théo Hermitte)
- 11/01 - Implémentation des instructions de branchage (Thomas Delatte)
- 11/01 - Implémentation des instructions de multiplication (Thomas Delatte)
- 12/01 - Implémentation des instructions de block data transfer (Maxime Bouchenoua)
- 13/01 - Implémentation des instructions de single data transfer (Rodolphe Beguin)
- 13/01 - Implémentation des instructions de single data swap (Leslie Metzger)
- 13/01 - Correction de bugs divers dans les instructions de data processing (Théo Hermitte)