TD-TP 1 : Assembleur MIPS, environnement de travail et traduction de structure de contrôle

Le but de ce TD-TP est de présenter l'environnement de travail utilisé dans tout le module et d'apprendre à traduire des portions de code décrite en C vers le langage d'assemblage MIPS.

Ex. 1 : Etude d'un exemple : le PGCD

Récupérez les sources fournies sur Chamilo et ouvrez le fichier *fct_pgcd.s* qui contient le code assembleur de la fonction pgcd_as.
Ce fichier contient la traduction systématique de la fonction C, mise en commentaire au début du fichier, en langage d'assemblage MIPS. Cette fonction calcule le PGCD de deux variables globales, a et b, de type `uint32_t` définies dans un autre fichier (pgcd.c).

Question 1 Trouvez dans ce fichier le point d'entrée de la fonction `pgcd_as`.

Voici quelques éléments d'informations, devant vous permettre de comprendre le contenu du fichier `fct_pgcd.s` :

- Par défaut, toutes les étiquettes (ou symboles) déclarées dans un programme assembleur sont privées et invisibles à l'extérieur du fichier. Or la fonction `pgcd_as` doit être visible pour pouvoir être appelée par le programme principal. C'est le but de la directive .globl qui rend l'étiquette pgcd publique.
- Par convention, une fonction en assembleur MIPS renvoie sa valeur de retour dans le registre `v0`, c'est à dire $2 [1]. Si on écrit une fonction qui ne renvoie rien (`void`), la fonction appelante ignorera le contenu de `v0`.
- En fin de fonction, il faut rendre la main à la fonction appelante. Pour cela, il faut sauter à l'adresse de l'instruction suivant celle qui a appelé notre fonction. Par convention, cette adresse, usuellement appelée adresse de retour, est stockée dans le registre `ra` ($31).
- Les variables globales `a` et `b`, déclarées dans le fichier `pgcd.c`, sont accessibles dans le fichier en langage d'assemblage `fct_pgcd.s` [2].

Une feuille résumant les informations à connaître en assembleur (cheat sheet) est à votre disposition sur Chamilo (Documents–Ressources_complementaires).

Question 2 Relisez les indications ci-dessus et le code assembleur tant que ce dernier ne vous parait pas limpide. Relevez les symboles définis dans ce fichier et leur portée (locale ou globale).

Question 3 Ouvrez le fichier `pgcd.c` et relevez-y les symboles déclarés et définis en précisant leurs portées.

1. La table de correspondance entre l'index d'un registre et son nom logiciel est fournie en annexe.
2. En C, pour limiter la portée d'une variable globale à un seul fichier, il faut utiliser le mot-clé `static`

Ex. 2 : Prise en main de l'environnement de travail

Dans tout le module CEP, nous utiliserons la suite de compilation GCC, qui vous a été introduite lors de votre initiation au C. Il s'agissait alors de compiler du code source (.c) en code objet pour l'architecture x86 (.o), la compilation s'exécutant elle-même sur une architecture x86. Comme le module CEP repose sur le processeur MIPS, il nous faut donc utiliser un compilateur croisé (cross-compiler) capable de générer du code MIPS sur une machine x86 (les PC et les ordinateurs portables). De même, les binaires générés avec une suite de compilation croisée ne peuvent s'exécuter sur la machine hôte, il faut soit les déployer sur du matériel compatible (comme vous l'avez fait dans la partie projet) soit les exécuter avec un logiciel émulant ce matériel compatible. Cette dernière solution permet de faciliter le développement logiciel et en particulier le débogage.

Durant ces TD-TP, nous utiliserons le logiciel QEMU pour émuler une plateforme MIPS. Sur les PC de l'Ensimag, cet outil est localisé dans le repertoire `/opt/mips-tools-cep/bin`. Il faut donc ajouter ce répertoire à votre PATH (par exemple, dans le fichier `$HOME/.bashrc` ajouter la ligne : "`export PATH=/opt/mips-tools-cep/bin:$PATH`" et entrer la commande suivante dans le terminal "`source ~/.bashrc`"). Cela aura également le bon goût de rendre accessibles les outils de compilation croisée.

Les questions suivantes présentent plusieurs manières d'utiliser le simulateur selon vos besoins.

Question 1 A partir du Makefile fourni, générez l'exécutable en tapant simplement : *make pgcd*

Question 2 Pour une exécution simple, vous pouvez lancer QEMU depuis un terminal avec la commande suivante :
`qemu-system-mips -M mipscep -nographic -kernel pgcd`
L'option `-M` fournit le nom de la plateforme MIPS à simuler, qui a été créée pour les besoins du module. L'option `-nographic` élimine les fenêtres graphiques de QEMU et redirige la sortie standard (donc vos `printf`) vers le terminal. Enfin, l'option `-kernel` précise l'exécutable à utiliser.
Lancez cette exécution.

Ce type d'exécution ne nous permet pas de suivre l'exécution pas à pas ou de le déboger efficacement. L'option `-s` indique à QEMU que nous allons le piloter via le debugger gdb en lui parlant sur le port tcp 1234. L'option `-S` indique à QEMU de ne pas démarrer la simulation. Ces deux options permettent de suivre l'exécution via un gdb connecté à notre émulateur. Vous trouverez une carte conceptuelle résumant tous ces éléments sur Chamilo dans Ressources complémentaires.

Question 3 Relancez QEMU en ajoutant les options `-s` et `-S` et depuis un autre terminal connectez y gdb avec la commande :
`mips-elf-gdb pgcd`
Le déboggeur va se connecter automatiquement au simulateur en réalisant les actions GDB décrite dans le script `.gdbinit` fourni. Utilisez maintenant `gdb` pour tracer l'exécution du programme `pgcd` instruction par instruction. Par exemple, on peut :

- ajouter un point d'arrêt au début du programme en utilisant la commande `break main` ;
- continuer l'exécution de programme avec la commande `continue` (Le stub QEMU a déjà fait le run et a posé un point d'arrêt sur le point d'entrée du binaire - ici, `_start`)
- afficher immédiatement les valeurs des variables `a` et `b` en utilisant les commandes `print a` et `print b` ;
- afficher de manière persistante la valeur des variables `res_c, res_as` avec la commande `display res_c` (Cette commande permet d'afficher le contenu de la variable après chaque

commande GDB. Dans notre cas, c'est intéressant pour observer le retour des fonctions `pgcd_c` et `pgcd_as`)
- continuer l'exécution du programme pas à pas avec la commande `next` ; On remarquera que GDB ne rentre pas dans la fonction `pgcd_c`.
- remarquer que GDB vous indique après chaque commande d'exécution la prochaine instruction (C ou asm) qu'il va exécuter dans votre programme et son numéro de ligne ; Pour voir les environs du code exécuté, utiliser la command `list` ;
- continuer l'exécution avec un `step`, qui exécute la prochaine instruction même à l'intérieur d'une fonction. Ici, nous entrons dans la fonction `pgcd_as` ;
- répéter la commande et remarquer que GDB a effectué la pseudo-instruction `lw` en 1 pas (alors qu'elle fait 2 instructions comme on verra à l'exercice 4) avec un `print $t0` ;
- avancer d'une seule vraie instruction avec la commande `stepi` (`print /x $t1` pour vérifier que le `lw` n'est pas encore réalisé)
- afficher de manière permanente (display) le contenu des registres `t0`, `t1` et `t2` ;
- avancer pas à pas (step) en vérifiant les valeurs des registres après l'exécution de chaque instruction qui les modifie, en vérifiant aussi la pertinence des sauts (e.g. vers else ou fin_if) ;
- une fois de retour dans la fonction `main` (après avoir exécuté l'instruction jr), afficher le registre `$v0`, contenant la valeur de retour et vérifier qu'elle a bien été affectée à la variable `res_as` ;
- terminer proprement l'exécution du programme en tapant `continue`.

Question 4 Pour simuler la plateforme que vous rêvez de concevoir en projet, relancez le simulateur sans les options `-nographic`, `-s` et `-S` sur le programme `mips-invader`. L'environnement graphique fournit vous permet d'interagir avec la carte presque comme en vrai : les boutons et interrupteurs sont cliquables, les LED, le 7-segment et l'écran sont fidèles à condition de respecter la même organisation en mémoire. L'affichage principal ne montre pas la sortie standard. Pour l'observer, il faut changer de fenêtre dans QEMU avec le raccourci ctrl+alt+3. On peut revenir à l'affichage principal avec le raccourci ctrl+alt+1.

Vous trouverez sur Chamilo (dans ressources externes et annales :DocGDB) un aide-mémoire pour gdb qui contient beaucoup plus d'information que ce qui vous sera nécessaire ce semestre. Parmi les commandes utiles non abordées, vous trouverez : `backtrace`, `x` ou `info reg`.

Ex. 3 : Traduction des structures de contrôle élémentaires

Dans cet exercice de mise en pratique, chaque question demande de traduire une fonction en langage d'assemblage à partir d'une description en langage C. Le code doit être traduit de manière systématique. Pour cela, la première étape consiste à fixer l'emplacement de chaque variable (numéro du registre, adresse mémoire, emplacement dans la pile (td2)). Ensuite, la traduction de la fonction C est effectuée ligne par ligne en respectant ce contexte. En conséquence, en traduction systématique, on s'interdit de réutiliser un résultat partiel d'une instruction déjà exécutée. Dans vos traductions, il vous sera toujours demandé d'indiquer l'emplacement de vos variables en commentaires au début de la fonction. De même, chaque séquence d'instructions assembleur doit être précédée d'un commentaire contenant la ligne de C correspondante.
Tous les exercices sont organisés de la même manière : un fichier exo.c qui contient le programme principal, un fichier `fct_exo.s` à remplir, et une règle de génération dans le Makefile (`make exo`) pour générer l'exécutable.
Dans tous les cas, il convient de vérifier l'exécution pas à pas du programme avec GDB et le

simulateur.

Question 1 Traduisez la fonction de `somme` des 10 premiers entiers naturels décrite dans `fct_somme.s`. Cette fonction ne manipule que des variables locales.

Question 2 Traduisez la fonction `sommeMem` qui effectue la somme des 10 premiers entiers naturels décrite dans fct_somme.s. La différence avec la question précédente vient du fait que res est maintenant une variable globale, un mot mémoire à réserver et manipuler avec `sw` et `lw`.

Question 3 Traduisez la fonction de multiplication simple `mult_simple` décrite dans `fct_mult.s`.

Question 4 Traduisez la fonction de multiplication egyptienne `mult_egypt` décrite dans `fct_mult.s`.

Question 5 Traduisez la fonction de multiplication native `mult_native` décrite dans `fct_mult.s`. (Pour cela vous pourrez consulter l'utilisation des instructions natives de multiplications notamment décrites pp. 112 et 133 de la documentation MIPS 3000)

Pour aller plus loin... **Question 6** Traduisez la fonction somme8 qui effectue la somme des 24 premiers entiers naturels décrite dans `fct_somme.s`. Attention : la variable globale `res` est sur 32 bits. Quand il s'agit de zones mémoires à manipuler sur 8 bits utilisez les instructions de transfert de mémoire adaptées `sb` et `lbu` (Pour cela vous pourrez consulter la documentation MIPS 3000 notamment aux pages p. 113, p. 134)

Pour l'exécution, vous pouvez afficher le contenu d'une variable sur 8 bits en utilisant par exemple `display (char)res`.

Pour aller plus loin... **Question 7** Retraduisez les fonctions de multiplication en optimisant le code. Précisez en début de fonction les optimisations réalisées. Comparez les performances avec le code non optimisé.

Ex. 4 : Prise en main de la suite de compilation

Question 1 A partir du Makefile fourni, générez l'exécutable en tapant simplement : *make pgcd* Identifiez les commandes utilisées, leurs rôles et ceux des options utilisées.

Nous allons maintenant essayer de comprendre ce qui s'est réellement passé en examinant les fichiers intermédiaires. Plusieurs des outils de la suite GNU `binutils` servent à afficher les informations contenues dans un fichier (objet ou exécutable) binaire : `nm` pour lister les symboles d'un fichier binaire, `objdump` pour en exposer le contenu brut (le terme "dump" est couramment utilisé) section par section. Comme vu en cours, ses sections reflètent l'organisation d'un exécutable en mémoire.

Question 2 En utilisant l'utilitaire `mips-elf-nm`, vérifiez que les objets générés définissent bien les symboles comme escompté dans l'exercice 1 [3].

Question 3 Observez le contenu de la section texte du module `fct_pgcd.o` en utilisant la commande : `mips-elf-objdump -D fct_pgcd.o | less`
Que remarquez-vous ? Qu'est-il arrivé aux 2 premières instructions de notre fonction `pgcd_as` ?

Les instructions utilisant des symboles non définies ne peuvent qu'être partiellement résolues. Il faut remettre à l'édition de lien leur résolution complète. Pour s'y retrouver, les fichiers objets utilisent des entrées relogables, que vous pouvez consulter avec l'option `-r` de `mips-elf-objdump`.

3. man nm peut vous aider à interpréter le résultat

Question 4 Consultez les symboles relogeables du module `fct_pgcd.o` et vérifiez (toujours avec `mips-elf-objdump`) qu'ils ont bien été mis à jour dans `pgcd` après l'édition de lien.

Pour aller plus loin... **Question 5** Avec `mips-elf-objdump`, observez le code généré pour la fonction `pgcd_c` et comparez le à celui de la fonction `pgcd_as`. Pour mieux observer les différences entre le langage d'assemblage écrit à la main et généré par un compilateur, vous pouvez demander à gcc de s'arrêter après la compilation avec l'option `-S`.

Pour aller plus loin... **Question 6** Voici un lot de questions permettant de comprendre la représentation d'un programme en mémoire. Ouvrez le fichier `put.c` et expliquez en quoi x et y diffèrent. Pourquoi ne peut-on pas compiler si on décommente // y++ ; ? Utilisez `mips-elf-nm` sur le fichier généré `put` pour trouver l'adresse des symboles x et y, puis utilisez `mips-elf-objdump -s` pour en voir le contenu. Que remarquez-vous ? Ecrivez sur papier la zone .sdata correspondant à la définition de x et y.

Annexes

Noms logiciel des registres à usages généraux du MIPS :

Registre	Nom logiciel	Usage
`$0`	`$zero`	toujours 0
`$1`	`$at`	réservé à l'assembleur
`$2-$3`	`$v0-$v1`	valeur(s) de retour (caller saved)
`$4-$7`	`$a0-$a3`	4 premiers arguments (caller saved)
`$8-$15,$24,$25`	`$t0-$t9`	temporaires (caller saved)
`$16-$23,$30`	`$s0-$s8`	à sauver (callee saved)
`$26-$27`	`$k0-$k1`	réservés au noyau (ne pas toucher)
`$28`	`$gp`	pointeur global (ne pas toucher)
`$29`	`$sp`	pointeur de pile (callee saved)
`$31`	`$ra`	adresse de retour (caller saved)