TP 1 : Assembleur RISCV, environnement de travail et traduction de structure de contrôle

Le but de ce TP est de présenter l'environnement de travail utilisé dans tout le module et d'apprendre à traduire des fonctions écrites en C vers le langage d'assemblage RISCV.

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

Ouvrez le fichier `fct_pgcd.s` qui contient le code assembleur de la fonction `pgcd`.
Ce fichier contient la traduction systématique de la fonction C, mise en commentaire au début du fichier, en langage d'assemblage RISCV. 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`.

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` 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 RISCV renvoie sa valeur de retour dans le registre `a0`, c'est à dire `x10`[1]. Si on écrit une fonction qui ne renvoie rien (`void`), la fonction appelante ignorera le contenu de `a0`.
- 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` (`x1`). On utilisera systématiquement la pseudo-instruction `ret` (qui s'expanse en `jalr zero, 0(ra)`) en fin de fonction, car les scripts d'évaluation se basent sur sa présence pour déterminer la fin de la fonction !
- Les variables globales `a` et `b`, définies dans le fichier `pgcd.c`, sont accessibles dans le fichier en langage d'assemblage `fct_pgcd.s`[2].
- Le contexte permet de dire si la fonction est feuille, c'est à dire qu'elle n'appelle pas d'autres fonctions. Il permet aussi de préciser où sont rangées les variables de la fonction C traduite : `i` et `j` sont les deux seules variables locales de la fonction C, elles seront rangées dans les registres `t0` et `t1`. `a` et `b` sont des variables globales définies dans le fichier `pgcd.c`.
- Les étiquettes `pgcd_fin_prologue` et `pgcd_debut_epilogue` seront utilisées dès le TP2, il ne faut jamais les enlever car elles sont utiles à l'évaluation.

1. La table de correspondance entre l'index d'un registre et son nom logiciel est fournie en annexe.
2. En C, les variables globales sont par défaut accessibles en dehors du fichier dans lequel elles sont définies. Pour limiter la portée d'une variable globale au fichier dans lequel elle est définie, il faut utiliser le mot-clé `static`.

Une feuille résumant les informations à connaître en assembleur (cheat sheet) est à votre disposition sur Chamilo (Documents–Ressources_complementaires), et/ou vous pouvez vous référer aux cours 2 et 3.

Question 2 Relisez les indications ci-dessus et le code assembleur présent dans le fichier `fct_pgcd.s` tant que celui-ci 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 leur portée.

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

Dans tout le module CEP, nous utiliserons l'environnement de développement GNU, auquel vous avez été introduit 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 RISCV, il nous faut donc utiliser un compilateur croisé (cross-compiler) capable de générer du code RISCV sur une machine x86 (les PC et les ordinateurs portables).

De même, les binaires générés avec un environnement de compilation croisée ne peuvent s'exécuter sur la machine hôte, il faut soit les déployer sur du matériel compatible (comme vous le faites dans la partie CP) 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 TP, nous utiliserons le logiciel QEMU pour émuler une plateforme RISCV. Sur les PC de l'Ensimag, cet outil est localisé dans le repertoire `/matieres/3MMCEP/riscv/bin`. Il faut donc ajouter ce répertoire à votre `PATH`. Pour cela, dans le fichier `$HOME/.bashrc` ajoutez la ligne : "`export PATH=/matieres/3MMCEP/riscv/bin:$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.

Pour pouvoir utiliser confortablement `gdb` dans la suite, ajoutez dès maintenant un fichier `.gdbinit` dans votre répertoire personnel, avec la ligne suivante :

```
set auto-load safe-path /
```

Vous pouvez le faire rapidement en tapant la ligne de commande suivante :

```
echo "set auto-load safe-path /" > $HOME/.gdbinit
```

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

Toutes les commandes sont à exécuter dans le répertoire de chaque TP dans le dossier contenant votre copie locale de votre dépôt git (`tp1/`, `tp2/`, etc.). Chacun de ces répertoires de TP contient les fichiers sources assembleur (`.s`) et C (`.c`) ainsi qu'un fichier `Makefile` pour compiler nos programmes.

Question 1 À 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-riscv32 -machine cep -bios none -nographic -kernel pgcd
```

L'option `-machine` fournit le nom de la plateforme RISCV 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. L'option `-bios none` retire le bios par défaut (opensbi) et nous permet de démarrer sur une machine totalement vierge. Enfin, l'option `-kernel` précise l'exécutable à utiliser.
Lancez cette exécution.

Question 3 Comme précisé au cours de présentation, cette partie « exploitation de processeur » (EP) de CEP est évaluée en contrôle continu. Chaque fonction développée sera évaluée par une infrastructure mise en place par vos enseignants. Nous allons vous montrer ici les étapes nécessaires pour évaluer et valider une fonction. Pour cela, nous allons effectuer cette validation pour la fonction `pgcd` fournie.

1. La première étape est de sauvegarder dans le fichier `test/pgcd.sortie` la sortie du programme `pgcd` appelant la fonction `pgcd` écrite en assembleur. Pour cela, vous pouvez lancer l'exécution du programme `pgcd` comme vu à la question précédente en redirigeant la sortie (utilisation de la redirection `> test/pgcd.sortie` en fin de ligne de commande).
2. La deuxième étape est de lancer une vérification locale de l'implémentation en assembleur de la fonction `pgcd`. Pour cela, se placer dans le répertoire du TP et lancer le script par : `../common/verif_etud.sh`.
3. Une fois que vous êtes satisfaits de votre implémentation en assembleur, c'est à dire une fois que la vérification locale de l'étape précédente se termine **sans erreur**, la dernière étape est de mettre à jour votre dépôt distant avec le contenu de votre dépôt local en faisant :
   — `git add fct_xxx.s` (`xxx` doit être remplacé par la fonction à évaluer ; non nécessaire pour `pgcd` car la fonction en assembleur est donnée)
   — `git add test/xxx.sortie` (`xxx` doit être remplacé par la fonction à évaluer ; `pgcd` ici)
   — `git commit`
   — `git push`
   Par la présence d'un fichier de sortie non vide par fonction, et de leur validation par le script `verif_etud.sh`, le fait de pousser lance l'évaluation de la fonction correspondante et les badges apparaissent sur votre page gitlab (attention il y a un délai nécessaire pour l'évaluation). Un clic sur un badge vous donne accès aux informations détaillées sur les étapes d'évaluation de la fonction correspondante, et vous affiche des messages ciblés en cas d'échec.

Ces trois étapes seront à effectuer pour chacune des fonctions développées. **Notez que la ponctualité et la régularité de votre travail sont évaluées : une fonction réussie la semaine du TP correspondant vaudra davantage de points. Néanmoins, toute fonction réussie sera prise en compte dans l'évaluation.**

Question 4 Pour simuler la plateforme que vous rêvez de concevoir en projet, compilez le programme `invaders` (`make invaders`) puis relancez le simulateur sans l'option `-nographic` mais en ajoutant l'option `-display default,show-cursor=on` sur le programme `invaders`. L'affichage principal ne montre pas la sortie standard, mais on peut néanmoins l'obtenir en ajoutant l'option `-serial mon:stdio`. En substance, tapez (et retenez dans votre historique) la ligne de commande suivante :

```
qemu-system-riscv32 -serial mon:stdio -display default,show-cursor=on
                    -machine cep -bios none -kernel invaders
```

Le simulateur vous permet alors d'interagir avec la carte « presque » comme en vrai : les boutons et interrupteurs sont cliquables, les leds, et l'écran sont fidèles. Pour l'observer, il faut changer de fenêtre dans QEMU avec le raccourci `<Ctrl><Alt>2`. On peut revenir à l'affichage principal avec le raccourci `<Ctrl><Alt>1`. On quitte QEMU tout simplement avec `<Ctrl><Alt>Q` ou le menu depuis la fenêtre QEMU.

Méthode de débogage d'un programme Le type d'exécution vue jusqu'à présent ne nous permet pas de déboguer, c'est à dire d'exécuter notre programme pas à pas pour comprendre finement son comportement. Cela est possible en combinant QEMU et `gdb`, voir la vidéo de présentation de la méthode. Inutile de la visionner maintenant, vous le ferez lorsque vous en aurez besoin, c'est-à-dire lorsque vous aurez un bug dans un programme (ce qui normalement devrait arriver dès l'exercice suivant !).

Vous trouverez une carte conceptuelle résumant tous ces éléments sur l'utilisation de `gdb` dans ce module sur Chamilo dans Ressources complémentaires.

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 (TP2)) dans l'espace « Contexte » qui précède immédiatement l'étiquette de la fonction. La grammaire décrivant les contextes se trouve en annexe. Notez que pour ce premier TP, toutes les fonctions sont « feuille » (cf. TP2) car elles n'appellent pas d'autres fonctions.

Ensuite, la traduction de la fonction C est effectuée ligne par ligne en respectant strictement ce contexte. En conséquence, en traduction systématique, on s'interdit de réutiliser un résultat partiel d'une instruction déjà exécutée. De même, chaque séquence d'instructions assembleur doit être précédée d'un commentaire contenant la ligne de C correspondante. Attention à bien laisser les balises "`DEBUT DU CONTEXTE`" et "`FIN DU CONTEXTE`" qui sont utilisées par l'évaluation.
Tous les exercices sont organisés de la même manière : un fichier `exo.c` qui contient le programme principal et les arguments pour votre fonction, un fichier `fct_exo.s` à remplir, et une règle de génération dans le Makefile (`make exo`) pour générer l'exécutable. Notez que `exo` est le nom de l'exercice qui variera.
Dans tous les cas, il convient de vérifier l'exécution pas à pas du programme à l'aide de `qemu` lancé avec les options `-s` et `-S` et de `gdb`. Puis, vous pouvez procéder aux trois étapes permettant d'évaluer cette fonction sur le dépôt.

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`.

`sw` avec une étiquette en argument est une pseudo-instruction, l'assembleur décompose cette pseudo-instruction en deux instructions en utilisant un registre pour stocker l'adresse de l'étiquette. Comme ce registre va être modifié, il faut indiquer explicitement dans la pseudo-instruction `sw` le registre utilisé pour construire cette adresse :
ex : `sw t1,symbole,t2` , où `t1` est le registre dont la valeur sera mise en mémoire à l'adresse

désignée par `symbole` et `t2` un registre utilisé pour construire l'adresse désignée par `symbole`.

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 pages 43 à 45 de la documentation RISCV (Ressources_complémentaires_riscv - riscv-spec.pdf).

Pour aller plus loin...
Question 6 Traduisez la fonction somme8 qui effectue la somme des 30 premiers entiers naturels décrite dans `fct_somme.s`. Attention : la variable globale `res` est sur 8 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 RISCV à la page 24.

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

Ex. 4 : Bien comprendre l'environnement de développement croisé

Question 1 À partir du `Makefile` fourni, générez l'exécutable en tapant simplement : `make pgcd`. Identifiez les commandes utilisées, leurs rôles et ceux des options utilisées. Un `make clean` avant de recompiler peut être nécessaire si vous n'avez pas modifié les sources.

Nous allons maintenant essayer de comprendre ce qui s'est réellement passé en examinant les fichiers intermédiaires. Plusieurs outils des *binutils* de GNU servent à afficher les informations contenues dans un fichier binaire (objet ou exécutable) : `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, ces sections reflètent l'organisation d'un exécutable en mémoire.

Question 2 En utilisant l'utilitaire `riscv32-unknown-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 de code (`.text`) du module `fct_pgcd.o` en utilisant la commande : `riscv32-unknown-elf-objdump -D fct_pgcd.o | less`
Que remarquez-vous ? Qu'est-il arrivé aux 2 premières instructions de notre fonction `pgcd` ? Une option intéressante à connaître pour répondre à cette question est `--disassembler-option=no-aliases`. Une autre option utile, en particulier en liaison avec le projet de conception de processeur est `--disassembler-option=numeric` qui affiche les registres avec leur numéro et non leur nom logiciel.

Les instructions utilisant des symboles non définis ne peuvent qu'être partiellement réso-

3. `man nm` peut vous aider à interpréter le résultat

lues. Il faut remettre à l'édition de lien leur résolution complète. Pour s'y retrouver, les fichiers objets utilisent des entrées relogeables, que vous pouvez consulter avec l'option `-r` de `riscv32-unknown-elf-objdump`.

Question 4 Consultez les symboles relogeables du module `fct_pgcd.o` et vérifiez (toujours avec `riscv32-unknown-elf-objdump`) qu'ils ont bien été mis à jour dans `pgcd` après l'édition de lien.

Pour aller plus loin...
Question 5 Avec `riscv32-unknown-elf-objdump`, observez le code généré pour la fonction `pgcd_c` et comparez le à celui de la fonction `pgcd`. 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 et de produire le fichier assembleur 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 l'on defini `CODE_FAUX` ?[4] Utilisez `riscv32-unknown-elf-nm` sur le fichier généré `put` pour trouver l'adresse des symboles `x` et `y`, puis utilisez `riscv32-unknown-elf-objdump -s` pour en voir le contenu. Que remarquez-vous ? Ecrivez sur papier la zone `.data` correspondant à la définition de `x` et `y`.

4. La règle `put_error` du `Makefile` vous permet de le faire à peu de frais.

Annexes

Noms logiciel des registres à usages généraux du RISCV :

Nom matériel	Nom logiciel	Signification	Préservé lors des appels ?
`x0`	`zero`	Zéro	Oui (Toujours zéro)
`x1`	`ra`	Adresse de retour	Non
`x2`	`sp`	Pointeur de pile	Oui
`x3`	`gp`	Pointeur global	Ne pas utiliser
`x4`	`tp`	Pointeur de tâche	Ne pas utiliser
`x5-x7`	`t0-t2`	Registres temporaires	Non
`x8-x9`	`s0-s1`	Registres préservés	Oui
`x10-x17`	`a0-a7`	Registres arguments	Non
`x18-x27`	`s2-s11`	Registres préservés	Oui
`x28-x31`	`t3-t6`	Registres temporaires	Non

Grammaire des contextes Pour faciliter la compréhension de la grammaire, les non-terminaux sont **en gras**. L'axiome est **contexte**.

contexte	→	Fonction : **ident** : **type** Contexte : **liste_contexte**
type	→	feuille | non feuille
liste_contexte	→	ε | **element_contexte** `\n` **liste_contexte**
element_contexte	→	**ident** : **list_localisation**
list_localisation	→	**localisation** | **localisation** ; **list_localisation**
localisation	→	registre **idreg** | registres **idreg** / **idreg**
	|	pile *(**adresse**) | pile *(**adresse**) / *(**adresse**)
	|	mémoire | mémoire, section **segment**
adresse	→	sp | sp + **nb** | sp - **nb**
segment	→	.data | .text | .rodata | .bss | ...

Les non-terminaux **idreg**, **ident** et **nb** représentent respectivement les noms de registres du processeur, les identifiants valides en C (noms de fonctions, de variables) et les entiers naturels ; ε représente le mot vide et `\n` un retour à la ligne. Un **element_contexte** peut être suivi par des commentaires, qui commencent avec un # et durent jusqu'à la fin de la ligne.