TD-TP 1 : Assembleur RISCV, 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 RISCV.

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 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_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_as` 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`).
- 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), et/ou vous pouvez vous référer aux cours 2 et 3.

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 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 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 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 (par exemple, dans le fichier `$HOME/.bashrc` ajouter la ligne : "`export PATH=/matieres/3MMCEP/riscv/bin:$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.

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

Toutes les commandes sont à exécuter dans le répertoire qui vous a été fourni contenant les sources et le Makefile.

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 -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. Enfin, l'option `-kernel` précise l'exécutable à utiliser.
Lancez cette exécution.

Notez que pour quitter QEMU, il faut faire la combinaison de touches `<Ctrl>Ax`, soit `<Ctrl>` et `A` en même temps, suivi de `x` sans appuyer sur `<Ctrl>`.

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.

Pour la suite, ajouter un fichier `.gdbinit` dans votre répertoire perso, 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
```

Question 3 Relancez QEMU en ajoutant les options `-s` et `-S` et, depuis *un autre terminal* dans *le même répertoire*, connectez-y gdb avec la commande :

```
riscv32-unknown-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 commande `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 avec un `print $t0` que GDB a effectué la pseudo-instruction `lw` en 1 pas (alors qu'elle fait 2 instructions, comme on le verra dans l'exercice 4) ;
- 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 `ret`), afficher le registre `a0`, 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, compiler le programme `invaders` (`make invaders`) puis relancez le simulateur sans les options `-nographic`, `-s` et `-S` mais en ajoutant l'option `-show-cursor` 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 -show-cursor -machine cep -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.

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. 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 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`.

`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,valeur,t2` , où `t1` est le registre dont la valeur sera mise en mémoire à l'adresse désignée par `valeur` et `t2` un registre utilisé pour construire l'adresse désignée par `valeur`.

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 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 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`.

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 l'environnement de développement croisée

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 (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 `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_as` ? 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 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 `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_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 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 on décommente // y++ ; ? 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.

3. man nm peut vous aider à interpréter le résultat

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

TD-TP 2 : Appel de fonctions

Le but de cette séance est d'apprendre à programmer et appeler des fonctions en langage d'assemblage, et de comprendre les conventions de gestion des registres et de la mémoire imposées par une ABI (Application-Binary Interface : ensemble de conventions permettant aux différentes parties du programme de communiquer, quels que soient leurs langages de développement respectifs) et la bibliothèque C que l'on utilise pour écrire nos programmes en langage d'assemblage. En respectant ces conventions, on verra qu'il est possible de mélanger du code C et du langage d'assemblage sans ambiguïté dans le même programme.

L'ABI du processeur RISCV est un document complexe à lire, car il fait des hypothèses sur la comprehension des mécanismes liés à la compilation à partir des langages de haut niveau [1]. L'ABI a pour but de définir les règles que doivent respecter les compilateurs et assembleurs, dans toutes les situations possibles, y compris certaines dépassant largement le but de ce cours. Les conventions présentées dans le cours et utilisées par la suite constituent donc un sous-ensemble volontairement simplifié des conventions détaillées dans l'ABI complète. Elles ont été sélectionnées de façon à être compatibles avec les besoins spécifiques de ce cours.

Pour ce TP, vous avez besoin de récupérer les sources mises à disposition sur gitlab. Comme dans le dernier TP, chaque programme est composé d'un fichier en C et d'un fichier en langage d'assemblage ; les commandes `make`, `qemu-system-riscv32` et `riscv32-unknown-elf-gdb` doivent se lancer depuis la racine des sources fournies pour ce TP.

Ex. 1 : Compréhension d'exemples détaillés

Le but de cet exercice est de vérifier et de consolider la compréhension des conventions utilisées pour réaliser des appels de fonctions en suivant l'ABI RISCV. Cet exercice vous montre aussi les « bonnes » manières d'écrire du code en langage d'assemblage dans le cadre de ce cours (commentaires précis et complets sur le contexte de la fonction, utilisation de commentaires pour indiquer les instructions C traduites). Veillez à choisir des valeurs pertinentes de paramètres pour pouvoir tracer l'exécution des programmes. Quand les arguments sont à saisir, ils le sont dans la console dans laquelle le simulateur, `qemu-system-riscv32`, s'exécute.

Question 1 Ouvrez le fichier `fct_pgcd.s` et observez la traduction en langage d'assemblage proposée. Justifiez le choix du contexte. Exécutez ensuite pas à pas ce programme avec GDB et le simulateur QEMU.

Pour la question suivante, et pour comprendre comment on manipule une variable locale dans la pile, on va placer exceptionellement la variable locale dans la pile même si c'est une fonction feuille.

Question 2 Même question que la question 1 avec le fichier `fct_mult.s`.

On placera les variables locales dans la pile uniquement lorsque c'est nécessaire (dû à un appel de fonction qui peut écraser la valeur de la variable, fonction non-feuille) ou si c'est indiqué dans

1. Les curieux la trouveront sur `https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md`.

un contexte imposé.

Question 3 Même question avec le fichier `fct_fibo.s`. Dessinez la pile lorsque `fibo(0)` est appelée la première fois pour le calcul de `fibo(4)`. Vérifiez ce dessin en observant dans GDB le contenu de la pile avec la commande `x /16x $sp`.

Ex. 2 : Exercice de traduction systématique : C vers langage d'assemblage

Dans les questions suivantes, on vous demande traduire du code C en langage d'assemblage en respectant les règles suivantes :

- préciser le contexte ;
- traduire de manière systématique les instructions C (Pour chaque instruction C, il faut récupérer les données aux emplacements désignés par le contexte. On ne cherche donc pas à optimiser le code en récupérant un calcul fait dans la traduction d'une instruction C précédente) et en particulier on va systématiquement rechercher les variables C là où elles se trouvent (pile, section `.data` ou `.bss`, ou tas) ;
- commenter avec au minimum le code C traduit.

Pour chaque traduction, tester et/ou déboguer avec les fichiers C fournis.

Question 1 Traduisez et testez la fonction `age` donnée en commentaires dans le fichier `fct_age.s`.

Question 2 Traduisez et testez la fonction `hello` donnée en commentaires dans le fichier `fct_hello.s`.

Question 3 Traduisez et testez la fonction `affine` donnée en commentaires dans le fichier `fct_affine.s`.

Question 4 Traduisez et testez la fonction `fact` donnée en commentaires dans le fichier `fct_fact.s`.

Pour aller plus loin... **Question 5** Traduisez et testez l'appel à la fonction C suggéré dans le fichier `fct_fact.s`.

Pour aller plus loin... **Question 6** Traduisez et testez la fonction `val_binaire` donnée en commentaires dans le fichier `fct_valbin.s`.

TD-TP 3 : Traduction de types de données avancés

Le but de cette séance est d'apprendre à traduire les types de données suivants : pointeurs, tableaux, chaines de caractères et structures,

Commencez par récupérer les sources pour cette séance sur Chamilo.

Les exercices sont organisés comme dans les TP précédents : 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 exercices, il convient de vérifier l'exécution pas à pas du programme avec le debugger (`gdb`)et le simulateur(`qemu`).

Ex. 1 : Manipulation de chaînes de caractères

Cet exercice vise à traduire en langage d'assemblage des programmes C manipulant des chaînes de caractères. Le fichier `fct_chaines.s` est le fichier de travail.

Question 1 Donnez le contexte de la fonction `taille_chaine` et traduisez-la en assembleur RISCV [1].

Question 2 Exécutez votre code avec le simulateur comme dans les séances précédentes. Si besoin mettez au point le programme en utilisant GDB et en affichant la chaine résultat à la fin. Les formats d'affichage /c et /s de la commande display peuvent vous être utile pour afficher dans le débogeur respectivement un caractère ou une chaîne de caractères.

Question 3 En respectant le contexte fourni, traduisez la fonction `inverse_chaine`.

Ex. 2 : Manipulation de tableaux

Le but de cet exercice est de travailler avec des tableaux d'entiers signés 32 bits. Le fichier de travail s'appelle `fct_tableaux.s`.

Question 1 Donnez le contexte de la fonction `tri_min` et traduisez-la en assembleur RISCV.

Ex. 3 : Manipulation de structures (agrégats)

Cet exercice permet d'appréhender la traduction de code C utilisant des types structurés (ou enregistrements). Les deux premières questions portent sur des paramètres et/ou des valeurs de retour qui sont structurées, et non des pointeurs vers de tels types. Les pointeurs sont eux en fait des scalaires assimilables à des entiers (des adresses machines, en fait).

1. On rappelle que le caractère ASCII ' \0' vaut 0. Par ailleurs pour lire un mémoire un seul octet on utilisera l'instruction lbu.

Question 1 Les fichiers `fct_struct.s` et `struct.c` contiennent des fonctions très simples utilisant un type structuré de taille inférieure ou égale à 64 bits.

Analysez ces fichiers pour répondre aux questions suivantes : Quelle convention utilise l'ABI pour passer en paramètres des structures de cette taille ? Comment sont représentées les structures en mémoire ? Comment accède-t-on à ces champs à partir d'un pointeur vers la structure dans ce cas ?

Pour aller plus loin... **Question 2** Que doit-on faire si le champ `entier` est remplacé par deux champs `p` et `q` de type `uint16_t` ? Même question s'il est remplacé par quatre champs `a, b, c, d` de type `uint8_t` ?

Question 3 Les fichiers `fct_rect.s` et `rect.c` contiennent des fonctions utilisant un type structuré de taille strictement supérieure à 64 bits. Quelle convention utilise l'ABI pour récupérer une valeur de retour qui est une telle structure ? Quelle convention utilise l'ABI pour passer en paramètres une telle structure ?

Pour répondre à ses questions, on va regarder le code de `fct_rect.s`, mais aussi désassembler `rect`, l'exécutable résultant de la compilation, et regarder la manière donc les variables `rin` et `rout` sont allouées et utilisées. (C'est ce qu'on appelle de la rétro-ingénierie de logiciel).

Question 4 Dans le fichier `fct_liste.s`, définissez le contexte de la fonction `inverse`, puis implantez-la[2]. Vérifiez le bon fonctionnement de votre code (`make`, `qemu`, `gdb`).

Pour aller plus loin... **Question 5** Implantez la fonction `decoupe` en respectant le contexte imposé, puis testez votre programme.

Pour aller plus loin... Ex. 4 : Palindrômes

Cet exercice vous permettra de manipuler des chaines de caractères et de faire des appels de fonctions. Le fichier de travail se nomme `fct_palin.s`.

Question 1 Donnez le contexte de la fonction palin, puis implantez cette fonction. Testez enfin le résultat.

2. On rappelle que NULL est codé par la valeur 0.

TD-TP 4 : traduction systéatique d'exemples complexes

Le but de cette séance est de consolider les acquis des séquences précédentes.

Commencez par récupérer les sources pour cette séance sur l'Ensiwiki.

Les exercices sont organisés comme dans les TP précédents : 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 exercices, il convient de vérifier l'exécution pas à pas du programme avec GDB et le simulateur.

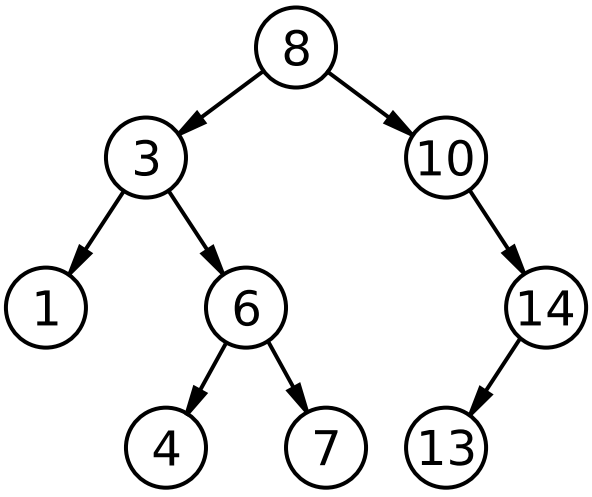
Ex. 1 : Parcours d'arbres binaires de recherche (annale d'examen de TP)

On va travailler dans cet exercice sur des arbres binaires de recherche (ABR) dont les nœuds contiennent une valeur représentable sur un entier naturel de 32 bits. On rappelle les principales propriétés de cette structure de données :

— chaque nœud de l'arbre a au plus 2 fils ;
— le sous-arbre gauche d'un nœud N donné ne contient que des nœuds dont la valeur est strictement inférieure à la valeur de N ;
— le sous-arbre droit d'un nœud N donné ne contient que des nœuds dont la valeur est strictement supérieure à la valeur de N ;
— les sous-arbres gauche et droit d'un nœud donné sont aussi des arbres binaires de recherche.

Il vient naturellement de ses propriétés que chaque valeur est unique (*i.e.* il n'existe pas plusieurs nœuds de même valeur).

Le schéma ci-dessous est un exemple d'ABR qu'on va utiliser dans cet exercice :



On représente le type des nœuds d'un ABR par la structure suivante :

```
struct noeud_t {
    uint32_t val;       // valeur d'un noeud
    struct noeud_t *fg; // fils gauche du noeud
    struct noeud_t *fd; // fils droit du noeud
};
```

Un ABR est représenté par un pointeur vers un nœud de type `struct noeud_t *abr`. Si `abr == NULL` alors l'ABR est vide.

Le fichier `abr.c` contient le programme principal et quelques fonctions servant à tester le code écrit en assembleur.

Le fichier `fct_abr.s` contient le squelette des fonctions à écrire. Le code C à traduire est donné en commentaires avant chaque fonction.

Question 1 Compléter la fonction `est_present` : cette fonction prend en paramètre une valeur entière naturelle sur 32 bits et un ABR. Elle renvoie vrai *ssi* la valeur est présente dans l'ABR.
Le principe de la fonction à écrire est simple :

- si l'arbre est vide, alors la valeur n'est sûrement pas présente, on retourne *faux* :
- sinon, si le nœud courant contient la valeur recherchée, on renvoie *vrai* :
- sinon, si la valeur recherchée est plus petite que la valeur du nœud courant, on poursuit la recherche dans le fils gauche :
- sinon (c'est à dire si la valeur recherchée est plus grande que la valeur du nœud courant), on poursuit la recherche dans le fils droit.

Question 2 Compléter la fonction `abr_vers_tab` : cette fonction prend en paramètre un ABR (c'est à dire un pointeur vers la racine de l'arbre).
La fonction parcourt l'ABR et copie les valeurs des nœuds de l'arbre dans un tableau. Comme le parcours se fait en profondeur d'abord dans le sous-arbre gauche, puis dans celui de droite, le tableau final sera donc trié par ordre strictement croissant. Au passage, la fonction détruit les nœuds pour récupérer l'espace mémoire.

Le tableau est alloué dans le programme principal (`uint32_t tab[NBR_ELEM];`). On recopie ensuite son adresse dans une variable globale `uint32_t *ptr` de type « pointeur vers un entier ». Cette variable est physiquement localisée dans la zone `.data` du fichier `fct_abr.s`, et elle est rendue visible dans le fichier `abr.c` grâce au mot clé `extern`.

Le principe de la fonction de copie est simple, pour un arbre non-vide :

- on copie récursivement tous les éléments du fils gauche du nœud courant dans le tableau :
- on copie la valeur du nœud courant dans le tableau :
- on incrémente la variable `ptr` de façon à ce qu'elle pointe sur la case suivante du tableau :
- on détruit le nœud courant (avec la fonction `free`) : pour pouvoir continuer à parcourir le reste de l'arbre, on doit donc d'abord sauvegarder un pointeur vers le fils droit du nœud :
- on copie récursivement tous les éléments du fils droit dans le tableau.

La variable `ptr` sert donc à désigner la prochaine case libre dans le tableau, et on l'avance d'une case à chaque fois qu'on copie une valeur dans le tableau.

Pour aller plus loin... **Ex. 2 : Passage de tableaux en paramètre**

Dans cet exercice, on cherche à implanter l'algorithme du tri du nain de jardin, dont voici le principe.

Un nain de jardin souhaite trier des pots de fleurs par taille croissante en appliquant la stratégie suivante. Il regarde le pot devant lui :

— s'il est plus petit que le pot à sa droite, le nain avance d'un pas vers la droite (s'il n'est pas arrivé à la fin de la file de pots) ;

— si le pot devant lui est plus grand que le pot à sa droite, le nain échange les deux pots, et recule d'un pas vers la gauche (s'il n'est pas revenu au début de la file de pots).

Le fichier `fct_tri_nain.s` contient en commentaires une implantation en C de cet algorithme. On note que la fonction `tri_nain` prend en paramètre le tableau d'entiers à trier. On rappelle qu'en C, lorsqu'on passe un tableau en paramètre d'une fonction, c'est l'adresse du tableau qui est passée (le premier paramètre d'une fonction est stockée dans $a0 et on ne peut évidemment par recopier l'ensemble des éléments du tableau dans un seul registre !).

Pour tester votre programme, le fichier `tri_nain.c` fourni calcule les temps d'exécution de votre tri et d'un tri de référence, et vérifie que votre tri est correct (un message d'erreur sera affiché si le tableau résultat est faux).

Attention, pour tester les performances nous utilisons un tableau de grande (tout est relatif) taille. Il est alors nécessaire d'avoir sur la plate-forme cep simulé plus de mémoire interne qu'elle n'en a en réalité sur la carte Zybo. Pour cela, on spécifiera avec l'option `-m` de QEMU la quantité de mémoire dont on a besoin. Par ex. 96 kB : `qemu-system-riscv32 -machine cep -m 96k -nographic -kernel tri_nain`.

Question 1 Traduisez en langage d'assemblage la fonction `tri_nain`.

On remarque qu'on fait beaucoup d'accès mémoire redondants dans cette traduction systématique, et qu'il semble facile d'optimiser le tri en utilisant intelligemment les registres pour stocker des valeurs.

Question 2 Implantez dans le fichier `tri_nain.s` une fonction `tri_nain_opt`, qui évite autant que possible les accès mémoires.

Notez bien que le tri de référence est un *quicksort*, c'est à dire un tri en $O(n \log(n))$ alors

que le tri du nain est un tri en $O(n^2)$. Même en optimisant le code produit, cela ne compensera pas le fait que l'algorithme du nain est intrinsèquement peu efficace.