Conception et Exploitation des Processeurs

Ensimag 1A 2012–2013

Durée : 2h

Consignes générales :

Le barème donné est indicatif. Les exercices sont indépendants et peuvent être traités dans le désordre.

Les documents sont interdits, **sauf une feuille A4 manuscrite recto-verso**. Photocopies et documents imprimés interdits. Les appareils électroniques (*e.g.* ordinateurs, téléphones, clés USB, *etc.*) sont interdits. **Le sous-répertoire `docs` contient des versions PDF des pages wiki du cours ainsi que des documentations pouvant vous être utiles.**

Dans tous les exercices, on demande de traduire **systématiquement** du code C en assembleur, comme le ferait un compilateur. Vous ne devez donc pas chercher à optimiser le code écrit et **vous devez placer et lire systématiquement les variables locales dans la pile d'exécution comme vu en TP**, sauf indication contraire dans les questions. Vous ferez particulièrement attention aux **types des données manipulées**, c'est à dire leurs tailles et leurs signes.

Pour chaque ligne de C traduite, vous recopierez en commentaire la ligne en question avant d'implanter les instructions assembleur correspondantes. Vous indiquerez aussi la position par rapport au registre `%rbp` de chaque variable locale, ainsi que les registres contenant les paramètres au début des fonctions implantées. Tous les commentaires additionnels sont les bienvenus.

A la fin de l'épreuve, vous devez fermer proprement votre session en cliquant sur l'icône « Sauvegarder et terminer l'examen ». **Une fois déconnecté, vous ne pourrez plus vous reconnecter et votre code sera rendu automatiquement**. Attention, vous ne devez surtout pas vous déconnecter via le menu Ubuntu, au risque de perdre votre travail. On recommande de **sauvegarder régulièrement votre travail** grâce à l'icône « Sauvegarder l'examen sans quitter » présente sur le bureau.

Tout le travail demandé est à rendre dans les fichiers `projet.txt`, `fct_exo3.s` et `fct_exo4.s` : le correcteur ne regardera pas les autres fichiers (vous avez le droit de les modifier pour vos tests).

Pour compiler vos programmes, vous utiliserez le `Makefile` fourni. Pour chaque exercice, vous pouvez compiler les sources en tapant simplement dans un terminal la commande `make exo#` (où `#` est le numéro de l'exercice, par exemple `make exo3`). Si vous voulez « nettoyer » le répertoire pour tout recompiler à partir de zéro, il suffit de taper la commande `make clean`. L'utilisation de `gdb` et `valgrind` est très fortement recommandée pour la mise au point de vos programmes.

Partie « Conception de Processeur »

Ex. 1 : Exercice préliminaire *(1 pt)*

Dans cet exercice particulièrement difficile, on vous demande de compléter le fichier `projet.txt` avec vos nom, prénom et numéro de groupe.

Ex. 2 : Questions sur le projet *(5 pts)*

Le fichier `projet.txt` contient le code VHDL des états de base `S_Init`, `S_Fetch_Wait`, `S_Fetch` et le début de `S_Decode`. On vous demande de compléter ce code pour implanter les instructions demandées ci-dessous.
Vous indiquerez en commentaire la description RTL (*Register Transfer Level*) des commandes que vous écrirez (comme on l'a déjà fait par exemple pour l'état `S_Fetch` en écrivant `IR := Memoire(PC)`).
Vous devez écrire toutes les commandes nécessaires pour implanter complètement ces instructions et revenir à l'état `S_Fetch`. Vous ajouterez tous les états intermédiaires qui vous semblent nécessaires.
On considère dans tout cet exercice qu'on ne gère ni les interruptions, ni les extensions, ni aucune autre instructions que celles demandées.

Question 1 Implanter l'instruction SRA dont on rappelle la description :
- action : Décalage à droite arithmétique immediat ;
- syntaxe : `sra $rd, $rt, sh` ;
- description : le registre $rt est décalé à droite de la valeur immédiate codée dans les 5 bits du champ *sh*, le bit de signe du registre étant introduit dans les bits de poids fort ; le résultat est placé dans le registre $rd ;
- opération : $rd \leftarrow rt_{31}^{sh} || rt_{31..sh}$;
- format R.

Question 2 Implanter l'instruction BGTZ dont on rappelle la description :
- action : branchement ssi registre strictement supérieur à zéro ;
- syntaxe : `BGTZ $rs, etiquette` ;
- description : si le contenu du registre $rs est strictement supérieur à zéro, le programme saute à l'adresse correspondant à l'étiquette ; le champ `IMM16` contient l'écart relatif en instructions correspondant à ce saut ; cet écart est calculé par l'assembleur ;
- opération : $rs > 0 \Rightarrow PC \leftarrow PC + 4 + (IMM16_{15}^{14} \ || \ IMM16_{15..0} \ || \ 00)$;
- format I.

Partie « Exploitation des Processeurs »

Conseil : ne restez pas bloqué sur une question ! Chaque fonction à écrire en assembleur est constituée d'un ensemble de lignes de C que vous devez traduire le plus littéralement possible : certaines lignes sont faciles, d'autres difficiles. Il est possible de valider des points en écrivant les lignes faciles, même si la fonction ne fonctionne pas totalement.

Ex. 3 : Tri de tableau *(6 pts)*

Dans cet exercice, on va travailler sur un tableau d'entiers contenant des valeurs tirées aléatoirement et que l'on va trier.
Le fichier `exo3.c` contient le programme principal et des fonctions utiles pour tester votre code. On ne vous demande pas de rajouter des tests, mais vous pouvez bien sûr le faire si cela vous aide à mettre au point vos fonctions.
Le fichier `fct_exo3.s` contient le squelette des fonctions à implanter en assembleur. On donne à chaque fois le code C à traduire : on vous demande une traduction la plus littérale possible, sans chercher à optimiser le code que vous écrivez. Vous indiquerez systématiquement la ligne de C traduite en commentaire avant la suite d'instructions correspondante.
Dans cet exercice, on manipule des tableaux : vous utiliserez vraisemblablement un mode d'adressage indirect basé sur des registres, par exemple de la forme `depl(regbase, regindex, type)`. On rappelle que les registres de base `regbase` et d'index `regindex` doivent obligatoirement être sur 64 bits. Par exemple, on n'a pas le droit d'écrire `(%rax, %dx)` car `%dx` est un registre sur 16 bits : on doit alors recopier `%dx` dans un registre 64 bits et compléter les bits de poids forts avec des 0 ou le bit de signe, selon si `%dx` contenait un entier signé ou non.
On rappelle ci-dessous l'utilisation de l'instruction `setcc` qui peut faciliter l'écriture du code :

– `setcc dst` affecte la destination `dst` à 1 ssi le code-condition `cc` est vrai et à 0 sinon (cette instruction s'utilise en général après une comparaison) : par exemple, `cmpq %rax, %rdx` suivi de `sete %cl` affecte 1 dans le registre `%cl` ssi `%rax` égal `%rdx` (les code-conditions sont les mêmes que pour les sauts conditionnels tels qu'ils ont été listés à la séance 1).

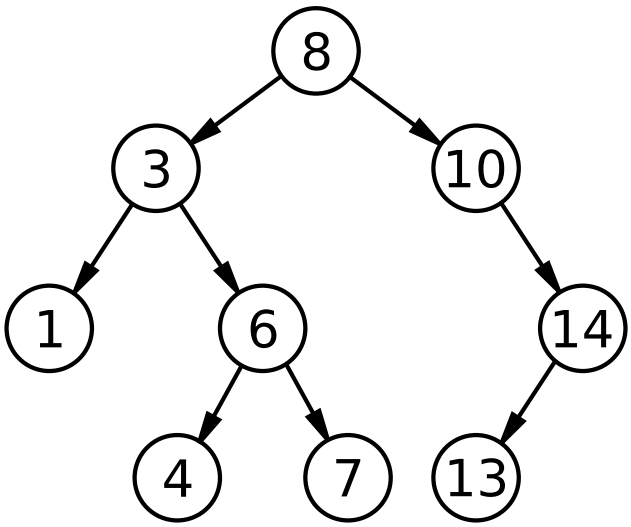
Question 1 Compléter la fonction `tab_aleat` : cette fonction prend en paramètre un tableau `tab` d'entiers signés sur 8 bits ainsi que la `taille` du tableau, représentée par un entier naturel sur 16 bits.
La fonction doit remplir le tableau avec des valeurs aléatoires. Pour cela, vous devez utiliser la fonction `rand()` de la bibliothèque C : cette fonction renvoie un entier naturel compris entre 0 et une très grande valeur. **On veut que le tableau contienne des entiers signés compris entre -8 et 7 inclus**, on doit donc d'abord réduire la valeur à 4 bits (symbolisé par le modulo 16 dans le code C) puis soustraire 8 pour passer dans les négatifs. Indication : il ne serait pas très efficace d'utiliser l'instruction de division pour calculer le modulo.
On va ensuite trier le tableau en utilisant un algorithme de tri particulièrement mauvais : le tri aléatoire. Le principe de ce tri est le suivant : on mélange aléatoirement les éléments du tableau, on regarde si on est arrivé (par hasard) à un tableau trié, et si ce n'est pas le cas, on recommence. Ce tri peut potentiellement prendre beaucoup de temps avant d'arriver à un résultat trié, mais on peut prouver qu'il se terminera en un temps fini si on utilise un générateur de nombres aléatoires de bonne qualité.
On fourni la fonction de tri `tri_aleat` dans le fichier `exo3.c` : vous n'avez pas besoin de la modifier ni de la comprendre précisément. Elle utilise une fonction `est_trie` que vous allez devoir écrire en assembleur.

Question 2 Compléter la fonction `est_trie` : cette fonction prend en paramètre un tableau `tab` d'entiers signés sur 8 bits ainsi que la `taille` du tableau, représentée par un entier naturel sur 16 bits. Elle renvoie un booléen valant vrai ssi le tableau est trié par ordre croissant (non-strict, c'est à dire que chaque élément est supérieur ou égal à son prédécesseur).
On rappelle qu'un booléen est représenté par un entier sur 8 bits valant 1 ssi on veut représenter la valeur vraie et 0 sinon (cela signifie que le bit de poids faible de l'entier vaut 1 ou 0 selon la valeur du booléen, et que les 7 bits de poids forts valent eux toujours 0).

Ex. 4 : Parcours d'arbres binaires de recherche *(8 pts)*

On va travailler dans cet exercice sur des arbres binaires de recherche (ABR) dont les nœuds contiennent une simple valeur entière naturelle. 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 clé 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 {
    uint64_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 simplement par un pointeur vers un nœud : `struct noeud_t *abr`. Si `abr == NULL` alors l'ABR est vide.

Le fichier `exo4.c` contient le programme principal et quelques fonctions servant à tester le code écrit en assembleur.
Le fichier `fct_exo4.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 64 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 renvoie 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 parcours 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 (`uint64_t tab[NBR_ELEM];`). On recopie ensuite son adresse dans une variable globale `uint64_t *ptr` de type « pointeur vers un entier ». Cette variable est physiquement localisée dans la zone `.data` du fichier `fct_exo4.s`, et elle est rendue visible dans le fichier `exo4.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.