ARCHITECTURE DES ORDINATEURS

TP : **06**

Étude de quelques assembleurs autres que x86/y86

Copiez le répertoire `/net/cremi/sathibau/Archi/asms` dans votre *home*. Oui, il n'y a qu'un seul fichier, `Makefile`, qui contient simplement les chemins vers les *cross-compilateurs* que l'on va utiliser pour traduire des programmes C en différents assembleurs (i.e. on donne l'option `-S` à gcc pour obtenir des fichiers `.S`)

1 Un premier programme tout simple

Dans un fichier `sub.c` écrivez une fonction `int sub(int x, int y)` qui retourne `x-y` (et rien de plus, pas de `#include` ni de fonction `main`). Pour traduire ce fichier C en différents assembleurs, lancez `make` qui génèrera plusieurs fichiers `.S` Commencez par lire la version `x86`, vous devriez n'avoir aucun problème pour la comprendre. Sauf si vous êtes curieux, vous pouvez ignorer les détails autour (toutes les directives `.bidule` du genre `.file`, `.text`, etc., ainsi que `@truc` etc.).

1.1 Version ARM

Il y a toutes les chances que le processeur de votre téléphone portable soit un ARM :)

L'ARM a des registres numérotés `r0`, `r1`, etc ainsi que des registres spéciaux `sp`, `pc`, `lr`, ... La syntaxe des instructions est `instruction dest, src1, src2`. Comment les paramètres sont-ils passés à la fonction ? Dans quel registre le résultat doit-il être passé ? Quel est l'équivalent du `ret` de x86 ? Où est stockée l'adresse de retour donc ?

1.2 Version PowerPC

Le PowerPC était auparavant utilisé dans les Macs, il est désormais surtout connu dans le grand public pour être dans la PS3. Il est aussi utilisé pour des supercalculateurs.

Les registres du PowerPC sont simplement désignés par leur numéro : `1`, `2`, etc. La syntaxe des instructions est la même que pour l'ARM. Comment les paramètres sont-ils passés à la fonction ? Dans quel registre le résultat doit-il être passé ? Quelle instruction est équivalente au `ret` de x86 ? Elle est en fait équivalente au `mov pc,lr` de l'ARM.

1.3 Pour aller plus loin : Version Mips

Le Mips a été beaucoup utilisé pour l'embarqué avant que l'ARM ne vienne le détrôner.

Les registres du Mips sont numérotés `$1`, `$2`, etc. La syntaxe des instructions est la même que pour l'ARM. Comment les paramètres sont-ils passés à la fonction ? Dans quel registre le résultat doit-il être passé ? Quelle instruction est *a priori* équivalente au `ret` de x86 ? La raison pour laquelle elle apparait en premier est qu'en Mips, les branchements n'ont pas effet immédiatement, l'instruction juste après un branchement est effectuée *avant* ledit branchement. Où est stockée l'adresse de retour ?

1.4 Pour aller plus loin : Version Itanium (ia64)

L'Itanium (aussi appelé ia64) devait être le remplaçant de l'x86, reprenant de zéro l'architecture en incluant de nombreuses fonctionalités prometteuses. Cela a été cependant un échec commercial. Ne travaillez sur la version Itanium que si vous avez le temps :)

Les registres de l'Itanium sont numérotés `r0`, `r1`, etc. La syntaxe des instructions est `instr dest = src1, src2`. Comment les paramètres sont-ils passés à la fonction ? Dans quel registre le résultat doit-il être passé ? Quel est l'équivalent du `ret` de x86 ? Dans quel registre spécial l'adresse de retour est-elle stockée ?

Une fonctionalité intéressante de l'ia64 est que le parallélisme des instructions est explicité : toutes les instructions non séparées par des `;;` peuvent être potentiellement exécutées en parallèle par le processeur (vous verrez les détails en cours plus tard). Pourquoi ici les deux instructions peuvent-elles être exécutées en parallèle ?

2 Pour aller plus loin : Jouez un peu

Essayez de modifier un peu votre programme pour vérifier vos hypothèses et observer le code assembleur produit : calculer $2 * x$, $2 * x + y$, $2 * x + 3 * y$. Rappel : `sal` et `asl` effectuent un décalage binaire à gauche, donc une multiplication par une puissance de 2. En ARM, il peut être appliqué directement à l'une des sources quelle que soit l'instruction. En x86, l'instruction `leal` se comporte comme `movl`, sauf qu'elle n'effectue en fait pas un transfert mémoire, elle calcule seulement l'adresse à laquelle elle aurait lu, et met le résultat de ce calcul dans le registre de destination. L'adressage `(%reg1,%reg2,i)` signifie `%reg1+i*%reg2`.

Essayez de calculer $x * y + z$. Remarquez que certains assembleurs n'utilisent qu'une seule instruction ! Dans quelle mesure un tel calcul est courant ?

3 Appel de fonction

Dans un autre fichier (pour pouvoir retourner voir le premier), déclarez `extern int g(int x);`, et écrivez une fonction `int f(int x, int y)` calculant $g(x) - g(y)$. Lisez la version x86, de nouveau elle ne devrait pas poser de problème.

3.1 Version ARM

Deux instructions complexes apparaisent : `stmfd` et `ldmfd`. Elles permettent de sauvegarder et restaurer une suite de registres en mémoire (ici, sur la pile car utilisées avec `sp`). Vous devriez alors être en mesure de comprendre le code, en sachant que tous les registres sont *callee-saved* et que `bl` sauvegarde elle-même l'adresse de retour dans le registre `lr`. Où est sauvegardée l'adresse de retour ? Quel est l'équivalent du `ret` de x86 ? (ce n'est pas le même que précédemment)

3.2 Version PowerPC

Il faut savoir qu'en PowerPC il n'y a pas de nom spécial pour le pointeur de pile, on utilise simplement par convention le registre `1`. La première instruction (`stwu`) effectue deux choses : soustraire 16 à `sp`, et écrire l'ancienne valeur à cette adresse-là. L'instruction `stw reg,mem` écrit en mémoire tandis que l'instruction `lwz reg,mem` lit depuis la mémoire. L'instruction `mr` effectue juste un transfert de valeur entre registres. Le registre `lr` où se situe l'adresse de retour (qu'il faut donc sauvegarder avant d'appeler `g`) est manipulé à l'aide des instructions `mflr` et `mtlr`. Vous devriez désormais pouvoir comprendre le code en sachant que tous les registres sont *callee-saved*. (oui, c'est normal que l'on sauvegarde éventuellement une des valeurs à l'adresse `sp+20`, c'est un emplacement déjà réservé pour cela)

3.3 Pour aller plus loin : Version Mips

L'instruction `sw reg,mem` écrit en mémoire tandis que l'instruction `lw reg,mem` lit depuis la mémoire. Les registres 8 à 15 et 24 et 25 sont *caller-saved*, les registres 16 à 23 sont *callee-saved*. L'appel de fonction n'est ici pas très simple, il y a d'abord lecture depuis une table de la véritable addresse :

calcul de l'adresse de la table dans `$28` par `lui` puis `addiu`, puis lecture dans la table par `lw`, et enfin le branchement proprement dit avec un `jalr`. De plus, l'opération `.cprestore` stocke le contenu du registre `$28` à l'adresse `$sp+16`, d'où il est restauré pour le deuxième appel. Rappel : l'instruction juste après le `jalr` est exécutée *avant* le branchement. `nop` ne fait rien. Vous devriez désormais être en mesure de comprendre le code.

3.4 Pour aller plus loin : Version Itanium (ia64)

La version Itanium fait intervenir un coulissement des registres, ce qui la rend plutôt difficile à comprendre. Pour les personnes vraiment curieuses, il faut comprendre que dans l'instruction `mov r38 = r32`, `r38` est en fait le paramètre qui sera passé à `g` (et que `g` verra en fait dans `r32`) : c'est l'instruction `alloc` qui a mis en place cette convention.