Université de Bordeaux
Licence STS

ARCHITECTURE DES ORDINATEURS

Projet :

Ajout d'instructions et prédicteurs statiques

N.B. : - Vous devez rendre, pour la date indiquée par vos enseignants, un rapport détaillant le travail que vous avez effectué en binôme. Les noms du binôme seront clairement indiqués sur la page de garde du rapport. Vous adjoindrez à ce rapport les fichiers du simulateur que vous aurez modifiés.
- L'ensemble sera envoyé par courriel à votre chargé de TD, sous forme d'un fichier archive de nom : `projet_nom1_nom2.tgz` .
- Comme la notation sera basée sur les informations fournies dans le rapport, celui-ci doit être complet, bien que concis.
- N'hésitez pas, en particulier, à l'illustrer des fragments de code que vous avez modifiés, afin de bien montrer la façon dont vous avez résolu les différents problèmes.
- La robustesse de votre approche sera également évaluée à la lumière des jeux de test que vous aurez utilisés. Ceux-ci sont donc doublement importants, puisqu'ils vous serviront également à tester votre approche au cours de son développement.

L'objectif de ce projet est à la fois d'étendre le jeu d'instructions du processeur Y86 et d'optimiser son exécution.

Le projet est en trois étapes, qui sont en fait indépendantes. En premier lieu, il s'agira de mettre en œuvre une nouvelle instruction, en recyclant le code d'une instruction existante, pour ne pas consommer d'opcode supplémentaire. Ensuite, on s'intéressera au fonctionnement de la prédiction de branchement, en mettant en œuvre deux types de prédiction statique, de type « jamais » puis « *backward taken, forward not taken* ». En troisième lieu, on mettra en place l'exécution de certaines instructions sur plusieurs cycles, permettant au Y86 d'exécuter des instructions plus complexes que ce qu'il était possible jusqu'ici.

Exercice 1 : Ajout de l'instruction `leal`

Lorsqu'on exécute une instruction `mrmovl` ou `rmmovl`, la zone mémoire à accéder est désignée par une expression de la forme « *`depl(%reg)`* ». On effectue donc l'addition entre le déplacement et le contenu du registre, avant d'utiliser le résultat, appelé *adresse effective*, comme adresse pour l'accès mémoire.

Sur les processeurs tels que le X86, pour lesquels le calcul de l'adresse effective en mode basé-indexé peut conduire à additionner deux registres, dont l'un multiplié par une petite puissance de deux, plus le déplacement, il y a un intérêt évident à pouvoir stocker l'adresse effective ainsi calculée dans un registre destination. Une seule instruction permet alors d'effectuer deux additions (et une pseudo-multiplication) d'un seul coup. Cette instruction s'appelle `leal`, pour « *load effective address (long)* ».

Sur le Y86, le format des instructions fait que les modes d'adressage sont bien moins évolués. Néanmoins, l'instruction « `leal` *`depl(%regS),%regD`* » remplace avantageusement la séquence « `rrmovl` *`%regS,%regD`* ; `iaddl` *`depl,%regD`* ».

Cette instruction peut être mise en œuvre sans consommer un nouvel opcode. En effet, il suffit de considérer qu'elle dérive de `irmovl`, qui a exactement le même format. Il suffira de placer la valeur `ifun` à `1` pour la distinguer.

Question 1

Implémentez l'instruction `leal`.

Tout d'abord, modifiez l'assembleur pour que cette instruction soit reconnue et générée par celui-ci. Modifiez les fichiers `yas-grammar.lex` et `misc/isa.c` pour ajouter la nouvelle instruction (cherchez où `irmovl` apparaît), qui prendra le même opcode qu'`irmovl` (`I_IRMOVL`), mais avec un `ifun` égal à `1`. Une difficulté liée à l'implémentation actuelle de l'assembleur est que le registre servant au calcul d'adresse est toujours situé en poids faible. l'ordre des registres doit donc être copié sur celui de `mrmovl` plutôt que de `irmovl`, ce qui imposera de les utiliser « à l'envers » dans le code HCL lorsque `ifun` vaudra `1`.

Vérifiez que l'assembleur compile bien un code source de test que vous créerez pour l'occasion, et que le code machine généré est bien le bon, avec `ifun=1` et la paire *`%regD,%regS`* en deuxième octet, à la place de `8,`*`%regD`*.

Modifiez ensuite le fichier HCL. En effet, puisque `irmovl` et `leal` utilisent le même opcode, il s'agit de les distinguer à l'exécution. Corrigez à la fois la version séquentielle et la version pipe-linée. Dans cette dernière, pour orienter le fonctionnement du processeur, vous aurez besoin de définir le signal :

```
intsig D_ifun  'if_id_curr->ifun'# Instruction function
```

Testez bien votre implémentation pour vérifier que l'instruction `irmovl` fonctionne encore !

Exercice 2 : Mise en œuve de stratégies de prédiction statique

On s'intéresse maintenant à la mise en œuvre de mécanismes de prédiction de branchement statiques dans l'architecture pipe-linée. Ces prédicteurs se basent uniquement sur la structure du programme pour décider de la prédiction, alors que les prédicteurs dynamiques mettent en œuvre une mémoire d'historique située sur le processeur (sous la forme d'une sorte de cache spécialisé) pour utiliser, comme prédiction d'un branchement situé à une adresse donnée, le résultat des décisions passés mémorisées pour cette adresse.

Gestion des versions

Vous aurez à gérer différentes versions de fichiers hcl. Vous pouvez simplement par exemple copier `pipe-std.hcl` en `pipe-maversion.hcl` et changer la variable `VERSION` dans le fichier `Makefile`.

Analyse du CPI

Reprenez les programmes de test `http://dept-info.labri.fr/ENSEIGNEMENT/archi/enonces/loop.ys` et `http://dept-info.labri.fr/ENSEIGNEMENT/archi/enonces/cpi.ys` utilisés en TP.

Question 1

Observez la valeur CPI en bas à droite du simulateur après exécution complète de chacune des boucles de ces programmes. Expliquer pourquoi on perd du temps avec la prédiction de branchement actuelle de type « toujours ».

Prédiction de branchement « jamais »

La prédiction de branchement actuellement mise en œuvre dans le fichier HCL est de type « toujours » : le simulateur suppose que les branchements conditionnels vont toujours être pris. Il s'agit maintenant de mettre en œuvre des prédictions de branchement statiques d'autres types, qui feront qu'un branchement conditionnel pourra ne pas être toujours pris.

Dans le cas « toujours », c'est `valC` qui est utilisé comme adresse de la prochaine instruction, et `valP` qui est conservé le long du pipe-line pour devenir l'adresse de la prochaine instruction si la prédiction s'avère fausse. Remarquez qu'afin d'économiser de la place dans les registres, `valP` est multiplexé en tant que `valA` dès l'étage `E`.

Dans le cas d'autres prédicteurs, il se pourra que ce soit `valP` qui serve d'adresse de la prochaine instruction, et il faudra alors préserver `valC`. Vous aurez donc besoin de propager `valC` également jusqu'à l'étage `M`, en plus de `valP`/`valA`. La faire cheminer dans l'UAL peut être un bon moyen pour cela, puisque celle-ci ne sert pas pour les branchements.

Vos modifications ne doivent pas impacter les branchements inconditionnels, qui doivent bien évidemment rester toujours pris. Pour pouvoir faire cela, il faut distinguer les branchements conditionnels des branchements inconditionnels. Ajoutez la définition :

```
intsig JUNCOND 'J_YES'
```

après celle de `ALUADD`. Vous pouvez alors l'utiliser pour raffiner les tests en remplaçant par exemple `(M_icode == JXX)` par `(M_icode == JXX) && (M_ifun != JUNCOND)` pour tester les branchements conditionnels seulement.

Question 2

En vous aidant des directives données ci-dessus, réalisez une version du processeur pipe-linée mettant en œuvre une prédiction de branchement de type « jamais ». Les zones à modifier concerneront le calcul de `f_pc`, de `new_F_predPC` et de la logique de contrôle du pipe-line, dans laquelle intervient déjà `Bch`.

Testez bien votre implémentation dans tous les cas de `loop.ys` et `cpi.ys` ; le résultat doit bien sûr rester le même.

Comparez les valeurs CPI obtenues avec celles de la prédiction « jamais ». Expliquez dans quels cas le nouveau prédicteur fonctionne mieux, et dans quels cas il fonctionne moins bien.

Prédiction de branchement « BTFNT »

Une stratégie statique plus élaborée, et qui est réputée donner de meilleurs résultats que les prédicteurs « toujours » et « jamais », est la prédiction « BTFNT », c'est-à-dire : « *backward taken, forward not taken* » : les branchements dont l'adresse de destination `valC` est inférieure à l'adresse de prochaine instruction `valP` seront considérés comme pris, et non pris dans le cas contraire.

Question 3

En vous aidant des directives données ci-dessus, réalisez une version du processeur pipe-linée mettant en œuvre une prédiction de branchement de type « BTFNT »,

Testez bien votre implémentation dans tous les cas de `loop.ys` et `cpi.ys` ; le résultat doit bien sûr rester le même.

Comparez les valeurs CPI obtenues avec celles des prédictions « toujours » et « jamais ». Expliquez dans quels cas le nouveau prédicteur fonctionne mieux, et dans quels cas il fonctionne moins bien.

Exercice 3 : Exécution d'instructions sur plusieurs cycles

Dans cet exercice, on ne travaillera que sur la version pipe-linée.

Les instructions `lods`/`stos`/`movs`

Les instructions `lods`, `stos` et `movs` n'ont pas de paramètre. Elles utilisent implicitement `esi` et `edi` comme adresse source et comme adresse destination, et `eax` comme valeur. `lods` lit en mémoire à l'adresse `esi`, stocke le résultat dans `eax`, et ajoute 4 au pointeur `esi`. `stos` écrit en mémoire à l'adresse `edi` le contenu de `eax`, et ajoute 4 au pointeur `edi`. `movs` lit en mémoire à l'adresse `esi`, écrit la valeur en mémoire à l'adresse `edi`, et ajoute 4 à `esi` et `edi`.

Question 1

Implémentez les instructions `lods` et `stos`. Ces instructions ressemblent beaucoup par leur fonctionnement aux instructions `popl` et `pushl`, desquelles vous pouvez vous inspirer.

Pour créer ces deux instructions, recyclez un opcode existant : celui de `I_POP2` (vous verrez plus tard en quoi ce choix est judicieux). Modifiez le fichier `misc/isa.h` afin de remplacer `I_POP2` par `I_LSM` (pour « `lods`/`stos`/`movs` »). Modifiez ensuite le fichier `misc/isa.c` comme indiqué dans l'exercice précédent, afin que `lods` corresponde à l'opcode `I_LSM` avec `ifun=0`, et `stos` au même opcode, avec `ifun=1`, et

supprimez la définition de « pop2 ». Ceci vous amènera aussi à intervenir dans psim.c, au niveau du décompte des cycles (notez bien cet endroit).

Créez un programme de test spécifique, pour vérifier que l'assembleur génère bien les bons codes machine, et que ce programme s'exécute comme prévu.

Support des instructions sur plusieurs cycles

L'instruction movs ne peut être implémentée en un seul cycle, parce qu'elle effectue deux accès mémoire. Pour la mettre en œuvre, il faut donc permettre qu'une instruction s'exécute en plus d'un cycle.

Pour cela, l'idée est que le programme contienne une seule instruction, avec ifun à une valeur initiale, et que le processeur injecte de lui-même plusieurs instructions, avec des valeurs croissantes de ifun, ayant des comportements différents. Par exemple, l'instruction x86 enter est équivalente à la séquence « pushl %ebp ; rrmovl %esp,%ebp ». On injecterait donc d'abord une instruction enter avec ifun=0+, à laquelle on donnera le comportement de « pushl %ebp », puis on injectera une instruction enter avec ifun=1, à laquelle on donnerait le comportement de « rrmovl %esp,%ebp ».

Le but des modifications décrites ci-dessous est de permettre de ne pas avancer le compteur ordinal lorsqu'on n'est pas à la fin du traitement de telles instructions, mais au contraire d'injecter un nouvel opcode à l'étape de décodage.

Dans le fichier pipe-std.hcl, ajoutez après instr_valid le bloc suivant :

```
int instr_next_ifun = [
       1 : -1;
];
```

Ce bloc indiquera au processeur quel valeur de ifun utiliser après celle que l'on vient de traiter. Par convention, la valeur -1 indique que l'instruction est finie, et qu'il faut passer à la suivante. C'est pourquoi c'est la valeur que l'on utilise par défaut.

Dans le fichier pipe/psim.c, à côté du prototype de gen_instr_valid, ajoutez le prototype
`int gen_instr_next_ifun ();`

Dans la fonction do_if_stage, repérez le code : `if (get_byte_val (mem, valp, &instr))`. Il s'agit du chargement d'instruction. Ajoutez devant :

```
if (gen_instr_next_ifun () != -1)
   ifun = gen_instr_next_ifun ();
else
```

Ainsi, on conserve la valeur précédente de icode, et ifun provient du code HCL. Pensez à passer fetch_ok à TRUE.

Enfin, repérez l'instruction `pc_next->pc = gen_new_F_predPC ();`, qui met à jour le compteur ordinal. Ajoutez devant :
`if (gen_instr_next_ifun () == -1)`
pour rester sur la même instruction.

Question 2

En vous appuyant sur les directives ci-dessus, implémentez movs sous la forme d'un premier cycle mettant en œuvre l'instruction lods, et d'un deuxième cycle mettant en œuvre l'instruction stos.

Pour ce faire, reprenez le même opcode que pour les instructions lods et stos, mais avec une valeur ifun=2 pour l'instruction movs telle que générée par l'assembleur. Le second cycle interne de cette instruction aura donc un ifun égal à 3, avant de passer à l'instruction suivante.

Dans psim.c, au niveau du décompte des cycles, prenez soin de modifier le code qui avait anciennement trait à pop2 pour que le décompte des cycles et des instructions continue à fonctionner avec movs, lods et stos.

Question 3

En fait, dans la sémantique du x86, le registre `eax` n'est pas modifié par l'instruction `movs`. Pour obtenir ce comportement, utilisez la pile pour le sauvegarder et le restaurer. Votre nouvelle instruction `stos` devra donc travailler sur quatre cycles, de valeurs `ifun` croissantes.