Université Bordeaux 1
Licence STS

ARCHITECTURE DES ORDINATEURS

Projet :

Instructions sur plusieurs cycles

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 lage 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 d'étendre le processeur Y86 pour lui faire exécuter des instructions sur plusieurs cycles, permettant ainsi de lui ajouter des instructions bien plus complexes que ce qu'il était possible jusqu'ici. On pourrait dire que c'est une étape vers une *micro-programmation* du Y86.

Le projet est en trois étapes, qui sont en fait indépendantes. En premier lieu, il s'agira de factoriser certaines instructions, pour libérer des numéros d'opcode. Ensuite, on ajoutera le support des instructions sur plusieurs cycles. Enfin, on pourra ajouter différentes instructions complexes. Si la première partie n'est pas faite, on pourra réutiliser simplement les opcodes de `leave`, `jreg`, `jmem` et `pop2`.

Exercice 1 : De la place dans les opcodes

La première étape est de faire de la place dans les opcodes Y86. Par construction, seuls 16 opcodes sont disponibles, et ils sont déjà tous pris ! Pour libérer des numéros, nous allons factoriser certaines instructions très similaires, en complexifiant un petit peu le code HCL.

Factorisation de `iaddl` etc. avec `addl` etc.

Il n'est pas vraiment nécessaire que les opérations arithmétiques utilisant des constantes (`iaddl`, `isubl`, etc.) utilisent un autre codage que leurs consœurs (`addl`, `subl`, etc.). En effet, si l'on étudie attentivement ces deux groupes d'instructions, on peut remarquer que les instructions avec valeur immédiate possèdent un champ `rA` égal à 8, c'est-à-dire `RNONE`. Il suffit d'utiliser cette caractéristique pour distinguer `iaddl` de `addl`, etc.

Question 1

Dans `misc/isa.h`, dans l'`enum itype_t`, renommez `I_ALUI` en `I_FREE1`, et ajoutez à la place :
`#define I_ALUI I_ALU`
afin que le reste du code utilise le même opcode pour `iaddl` et `addl`, etc.

Vous remarquerez par contre que le fichier `misc/isa.c` ne compile plus. Ce fichier contient l'implémentation en langage C du processeur Y86. Du fait que nous avons identifié `I_ALUI` avec `I_ALU`, le constructeur `switch...case` ne fonctionne plus, car il possède deux cas identiques. Mettez simplement en commentaire (entre `#if 0` et `#endif`) l'ensemble du `case I_ALUI`, puisque nous n'utilisons pas cette implémentation C.

Question 2

Il s'agit maintenant de corriger le fichier HCL. En effet, puisque `IOPL` et `OPL` sont désormais confondus, il s'agit maintenant de les distinguer. Notamment, le premier a besoin de charger une constante, et de l'utiliser comme opérande du côté `A` de l'UAL, alors que l'autre ne doit pas en charger, et utiliser simplement `rA`. Corrigez à la fois la version séquentielle et la version pipe-linée. Testez bien votre implémentation pour vérifier que ces instructions fonctionnent encore !

Factorisation de `irmovl` avec `rrmovl`

Procédez de même pour `irmovl` et `rrmovl`, afin de libérer le numéro d'opcode `I_FREE2`.

(bonus) Factorisation de push/pop/call/ret

`push`/`pop`/`call`/`ret` se ressemblent beaucoup. Il pourrait être utile d'en factoriser certaines, en les distinguant simplement à l'aide du champ `ifun` auquel on donnerait des valeurs différentes. En regardant les similitudes dans le code HCL, est-il intéressant de factoriser par paires `push`/`pop` et `call`/`ret`, ou bien par paires `push`/`call` et `pop`/`ret`, ou bien encore le quadruplet entier `push`/`pop`/`call`/`ret` ?

Réalisez la factorisation.

Exercice 2 : Ajout du support d'instruction sur plusieurs cycles

Pour qu'une instruction se déroule sur plusieurs cycles, l'idée est que le programme contienne une seule instruction, avec `ifun` à zéro, 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 `push %ebp; rrmovl %esp,%ebp`. On injectera donc d'abord une instruction `enter` avec `ifun=0`, à laquelle on donnera le comportement de `push %ebp`, puis on injectera une instruction `enter` avec `ifun=1`, à laquelle on donnera le comportement de `rrmovl %esp,%ebp`.

Version séquentielle

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

Dans la fonction `sim_step`, repérez `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.

Enfin, repérez `pc_in = gen_new_pc();` qui met à jour PC. Ajoutez devant

`if (gen_instr_next_ifun() == -1)`

pour rester sur la même instruction.

Version pipe-linée

Ajoutez de la même façon le bloc `instr_next_ifun` et la déclaration de `gen_instr_next_ifun`.

Repérez également dans la fonction `do_if_stage` l'appel à `get_byte_val`, et corrigez de manière similaire à la version séquentielle. Pensez à passer `fetch_ok` à `TRUE`.

Repérez l'instruction `pc_next->pc = gen_new_F_predPC();`, et corrigez le code de manière similaire à la version séquentielle.

Exercice 3 : Ajout d'instructions

Instruction `enter`

Question 1

Il s'agit d'abord d'ajouter l'instruction `enter` aux outils Y86.

Dans `misc/isa.h`, dans `itype_t`, remplacez un des opcodes non utilisés (soit l'un de ceux libérés à la première partie, soit un parmi `leave`, `jreg`, `jmem`, `pop2`) par `I_ENTER`.

Dans `misc/yas-grammar.lex`, ajoutez `enter` à la règle `Instr`.

Dans `misc/isa.c`, ajoutez l'instruction `"enter"` au tableau `instruction_set`. On pourra observer le contenu des instructions existantes pour savoir comment remplir les champs. Ajoutez également une instruction `"enter1"`, identique à `"enter"` à part le champ `ifun`, qui passera à 1.

Dans `seq/seq-std.hcl` et `pipe/pipe-std.hcl`, ajoutez la définition `intsig ENTER`.

Question 2

Comme précisé précédemment, l'instruction X86 `enter` est équivalente à : `pushl %ebp; rrmovl %esp,%ebp`.

Il vous suffit d'ajouter à `instr_next_ifun`

`icode == ENTER && ifun == 0 : 1;`

pour que la première fois que l'instruction `enter` est vue (avec `ifun=0`, une deuxième instruction `enter1` soit injectée automatiquement, avec `ifun=1`, avant de passer aux instructions suivantes du programme.

Il ne vous reste plus qu'à implémenter dans `seq-std.hcl` et `pipe-std.hcl` le comportement de `enter` pour `ifun=0` (c'est-à-dire l'équivalent de `pushl %ebp`), et de `enter` pour `ifun=1` (c'est-à-dire l'équivalent de `rrmovl %esp,%ebp`).

Testez bien votre instruction !

Instruction `mul`

Pour implémenter la multiplication, on peut utiliser l'algorithme naïf d'additions successives (on supposera dans ce cas que les opérandes sont positifs).

Il est tout à fait possible d'implémenter cette instruction dans notre Y86 étendu. Il suffit pour cela de construire une boucle micro-codée en utilisant `ifun` comme compteur ordinal. Pour savoir quand s'arrêter, il suffit de faire un calcul et d'utiliser la valeur de `cc` pour passer `ifun` à `-1`, ou bien à une autre valeur pour continuer à boucler. Pour accéder à `cc`, il suffit d'ajouter la définition `intsig cc 'cc'`. Le flag `Z` vaut `4`, le flag `S` vaut `2`, le flag `O` vaut `1`. Il suffit donc de tester `cc == 2` pour savoir si l'on a obtenu un résultat strictement négatif.

On supposera implicitement que `%eax` recevra le résultat. L'instruction `mul %ebx,%ecx` effectuera donc `eax = ebx*ecx`, et l'on se permettra de détruire `%ecx`, qu'on supposera différent de `%ebx`. Il vous faudra ajouter bien sûr la définition `intsig REAX`.

Dans la version pipe-linée, à quoi faut-il faire attention ?

(bonus) 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`.

Implémentez `lods` et `stos`.

Implémentez `movs`, en utilisant d'abord le registre `eax` comme intermédiaire. En x86, le registre `eax` est préservé. Pour obtenir ce comportement, utilisez la pile pour le sauvegarder.

(bonus) Instruction `repstos`

En x86, `rep stos` répète l'instruction `stos` autant de fois que le contenu de `ecx`, qui est donc décrémenté jusqu'à zéro. C'est donc en fait une sorte de `memset`.

Question 3

Implémentez une instruction `repstos` dans le processeur séquentiel, et testez-la.

Question 4

Implémentez `repstos` dans le processeur pipe-liné. À quoi faut-il faire attention ?