Université de Bordeaux
Licence STS

université de BORDEAUX

ARCHITECTURE DES ORDINATEURS

TP : **12**

Mini-projet avec l'architecture Y86(seq)
Sujet 3

N.B. : - Vous devez rendre, pour la date indiquée par vos enseignants, un rapport au format PDF détaillant le travail que vous avez effectué en binôme, accompagné des différentes sauvegardes du simulateur correspondant à l'environnement que vous aurez modifié. Les noms du binôme devront être clairement indiqués sur la page de garde du rapport.
- Les attendus devront être rendus au moyen de la plate-forme Moodle de l'établissement : `https://moodle1.u-bordeaux.fr/course/view.php?id=6211`
- 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.

Introduction

Le projet est en deux étapes, qui sont en fait indépendantes. En premier lieu, il s'agira de factoriser certaines instructions, pour libérer des numéros d'*opcodes*. Ensuite, on ajoutera le support de nouvelles instructions.

Attention : Pour des raisons de précédence des opérateurs HCL parfois surprenante, n'hésitez pas à utiliser des parenthèses pour isoler chaque opérateur.

Exercice 1 : De la place dans les *opcodes* Y86

La première étape est de faire de la place dans les *opcodes* Y86. Par construction, seuls 16 *opcodes* sont disponibles. Pour libérer des numéros, il vous est demandé de factoriser certaines instructions très similaires, en complexifiant un petit peu le code HCL.

Exercice 2 : Factorisation de `popl` et `ret`

Les instructions `popl` et `ret` sont en fait très similaires : le calcul d'adresse est identique, la seule chose qui diffère étant la façon dont est utilisée la donnée lue en mémoire. On peut donc les factoriser facilement en un même `icode`, le champ `ifun` étant utilisé pour distinguer les deux instructions.

On conservera donc uniquement l'`icode` correspondant à `POPL`, et l'on prendra `ifun` $= 0$ pour `popl`, et `ifun` $= 1$ pour `ret`.

Pour être sûr de traiter toutes les occurrences dans le code HCL, vous pourrez remplacer la définition du symbole `POPL` par un nouveau symbole « `PORE` » (pour « *pop-ret* »). Ainsi, toutes les occurrences de `POPL` et `RET` donneront des erreurs dans le code HCL tant que vous ne les aurez pas traitées. Sinon, il vous faudra supprimer les instances de `RET` pour les remplacer par des `POPL` s'il n'est pas déjà présent, tout en distinguant les quelques cas où le comportement des signaux doit varier selon la valeur de `ifun`.

Question 1

Dans le jeu d'instructions du simulateur, modifiez l'instruction `ret` pour qu'elle corresponde au même `icode` que `popl`, mais avec un `ifun` égal à 1.

Vérifiez qu'un code source Y86 compile bien sous cette nouvelle forme. Pour cela, compilez un programme qui contient à la fois l'instruction `popl` et `ret`, et regardez le code hexadécimal correspondant.

Question 2

Modifiez le code HCL du simulateur afin que le comportement des nouvelles formes des instructions `popl` et `ret` soit conforme aux anciennes.

Pour cela, et puisque `POPL` et `RET` sont désormais confondus, il s'agit maintenant de les distinguer dans les quelques cas où ils ne doivent pas avoir le même comportement. Notamment, il faut distinguer, aux étages *memory*, *write-back* et *next IP*, entre les cas « `((icode == PORE) && (ifun == 0))` » et « `((icode == PORE) && (ifun == 1))` ».

Exercice 3 : Ajout des instructions `negl` et `cmpl`

L'objectif de cet exercice est d'enrichir le jeu d'instructions de l'architecture Y86 avec deux nouvelles instructions : `negl` et `cmpl`, qui existent dans le jeu d'instructions du processeur X86 (vous pouvez les chercher sur Internet). Ces instructions servent à écrire du code plus compact qu'avec les instructions de soustraction classiques.

Linstruction `negl` (pour « *negate* ») sert à remplacer la valeur d'un registre par la valeur opposée. Ainsi, après l'instruction « `negl %reg` », la valeur de `%reg` est remplacée par $-$`%reg`, c'est-à-dire le résultat du calcul ($0 -$ `%reg`). Si cette instruction n'est pas disponible, calculer l'opposé d'une valeur demande un registre intermédiaire pour effectuer le calcul « `%reg` $= 0 -$ `%reg` ».

L'instruction `cmpl` sert à comparer les valeurs de deux registres, pour positionner les codes de condition `Z`, `S` et `O` sans modifier la valeur des registres comparés. « `cmpl %rA,%rB` » est en fait identique à « `subl %rA,%rB` », à la différence que la valeur de `%rB` n'est pas modifiée par l'instruction.

Ces deux instructions sont utilisées pour augmenter la compacité des codes objets (moins de place occupée en mémoire à fonctionnalité égale).

Question 1

Modifiez le jeu d'instructions pour que l'instruction `negl` soit reconnue par l'assembleur. Pour cela, utilisez un `icode` dédié, associé au code `NEGL`, avec un `ifun` égal à 0. Vérifiez qu'un code source Y86 compile bien avec cette nouvelle instruction. Pour cela, créez un code de test censé effectuer l'inversion de plusieurs registres contenant des valeurs différentes.

Question 2

Modifiez le code HCL pour implémenter l'instruction `negl`. Pour cela, ajoutez la déclaration des signaux `intsig NEGL` et `ALUSUB`, en vous inspirant des autres, puis traitez le cas `negl` dans les différents blocs.

Comme on voudra que le code HCL soit le moins modifié possible, réfléchissez bien avant de choisir si le registre utilisé pour `%reg` est `rA` ou `rB`. Justifiez votre choix dans votre rapport (votre choix doit bien sûr être cohérent avec celui de la question précédente).

Testez votre implémentation en vérifiant bien, dans chaque étage du processeur, que les valeurs des signaux et le comportement du processeur sont bien ceux attendus.

Question 3

Modifiez le jeu d'instructions et le code `HCL` pour ajouter l'instruction `cmpl`. Pour cela, réutilisez l'*opcode* `NEGL`, mais avec un `ifun` égal à 1, en factorisant les cas communs.

Comme on voudra que le code HCL de l'étage *execute* soit le moins modifié possible, réfléchissez bien à la façon de mettre en œuvre les soustractions au niveau de l'unité arithmétique et logique. Justifiez votre choix dans votre rapport.

Question 4

En utilisant les instructions `negl` et `cmpl`, codez en langage d'assemblage Y86 une fonction `maxabs`, qui calcule le maximum en valeur absolue des éléments d'un tableau contenant à la fois des éléments négatifs et positifs.

Testez cette fonction dans le simulateur, au moyen d'une fonction principale.

Exercice 4 : Ajout de l'instruction `loop`

L'objectif de cet exercice est d'enrichir le jeu d'instructions de l'architecture Y86 avec trois nouvelles instructions : `LOOP`, `LOOPE` et `LOOPNE`, qui existent dans le jeu d'instructions du processeur X86 (vous pouvez les chercher sur Internet). Ces instructions servent à faciliter l'écriture des instructions de boucle.

L'instruction « `loop label` » est équivalente à la combinaison des deux instructions « `isubl 1,%ecx ; jne label` », à la différence qu'elle ne doit pas modifier les codes de condition `Z`, `S` et `O` lors de la décrémentation (on verra pourquoi lors des questions suivantes). Elle décrémente donc le registre `ecx`, puis branche à l'étiquette `label` si `ecx` n'a pas atteint la valeur 0. Cette instruction est typiquement utilisée pour effectuer des boucles *for*, dans lesquelles le compteur de boucle décroissant est placé dans le registre `ecx`.

Question 1

Modifiez le jeu d'instructions pour que cette instruction soit reconnue par l'assembleur. Pour cela, utilisez un `icode` dédié, associé au code `LOOP`, et utilisez une valeur d'`ifun` quelconque (ce peut être 0 ou non, pour cette question). Vérifiez qu'un code source Y86 compile bien avec cette nouvelle instruction. Pour cela, créez un code de test censé effectuer quelques tours de boucle au moyen de cette instruction `loop`.

Question 2

Modifiez le code HCL pour implémenter l'instruction `loop`. Pour cela, ajoutez la déclaration des signaux `intsig LOOP` et `RECX`, en vous inspirant des autres, puis traitez le cas `LOOP` dans les différents blocs. Notamment, modifiez le calcul du signal `new_pc`. Pour décider de prendre le branchement ou pas, comme on ne teste pas les codes de condition `Z`, `S` et `O`, il faut simplement tester si `valE` vaut 0 ou non.

Testez votre implémentation en vérifiant bien, dans chaque étage du processeur, que les valeurs des signaux et le comportement du processeur sont bien ceux attendus.

Exercice 5 : Ajout des instructions `loope`/`loopne`

Les instructions `loope` et `loopne` se comportent comme `loop`, sauf qu'elles examinent également le drapeau `Z` : `loope` ne branche que si `ecx` est différent de 0 et que `Z` vaut 1, alors que `loopne` ne branche que si `ecx` est différent de 0 et que `Z` vaut 0. C'est pour cela que les instructions `loop`x ne doivent pas modifier les codes de condition `Z`, `S` et `O` lorsqu'elles effectuent la décrémentation de `ecx` : elles doivent en utiliser la valeur actuelle.

Dans le code HCL, vous ne pouvez pas utiliser directement les valeurs des codes de condition `Z`, `S` et `O` pour effectuer ou pas le branchement. En regardant le code HCL associé à `JXX`, vous verrez que la décision de branchement dépend de la valeur d'un signal binaire `Bch`. En fait, ce signal est calculé à partir de la valeur courante des codes de condition `Z`, `S` et `O` et de la valeur `ifun` correspondant au type de branchement.

Question 1

Pour que le codage des instructions `loope` et `loopne` puisse tirer parti du circuit qui calcule le signal de décisiion de branchement à partir de la valeur courante des codes de condition `Z`, `S` et `O`, il faut donc que la valeur de `ifun` que vous choisirez positionne bien le signal `Bch`.

Proposez un codage des instructions `loop`, `loope` et `loopne` qui fasse que le signal `Bch` soit conforme au comportement de branchement que l'on souhaite, en ce qui concerne la valeur du bit `Z`. Ajoutez les instructions `loope` et `loopne` au jeu d'instructions, en utilisant le même `icode` que pour l'instruction `loop`.

Question 2

Ajoutez le support des instructions `loope` et `loopne` dans le code HCL.

Question 3

En utilisant l'instruction `loopne`, codez en langage d'assemblage Y86 un clone de la fonction `strncpy`. Du fait des limitations du langage Y86, qui ne manipule que des mots sur 32 bits, cette fonction `strncpy` opérera sur des tableaux d'entiers sur 32 bits et non sur des tableaux de caractères sur un octet.

Testez cette fonction dans le simulateur, au moyen d'une fonction principale testant les deux cas de figure : dans le premier, la valeur sentinelle 0 est trouvée avant que le nombre de mots maximum à copier soit atteint, alors que dans le second, la valeur sentinelle se trouve plus loin que le nombre de mots à copier.