Implémentation de la Machine Virtuelle ICFP

Guillaume Hivert

15 février 2016

Introduction

Le but premier réside dans l'implémentation de la machine virtuelle d'Umix, un système d'exploitation écrit dans le but du concours ICFP 2006. Ce système repose sur une machine virtuelle, écrite dans le langage de notre choix (ici le C, pour des questions de performances) capable de lire les instructions des programmes et de les exécuter.

Dans un premier temps, les instructions seront abordés, puis la façon dont les problèmes ont été résolus seront vus dans une seconde partie.

Bases

Spécifications

La machine virtuelle universelle (UM par la suite, pour Universal Machine) est une spécification d'une machine abstraite gérant des mots de 32 bits, reposant sur plusieurs facteurs simples :

- Une mémoire globale, se présentant sous forme d'une collection de tableaux de mots.
- Un tableau spécial : le tableau de code (ou tableau 0).
- Huit registres pouvant chacun contenir un mot de 32 bits.
- Un pointeur de code (ou PC, program counter).

A l'aide de ces éléments il est possible d'écrire une machine capable d'exécuter les quatorze instructions de l'UM (cf. Annexe). En effet, chaque instruction se compose d'un mot binaire, duquel on extrait un opcode (le code de l'opération à effectuer), puis de valeurs à extraires selon l'opération considérée.

Cette machine a également la particularité de traiter des programmes en Big Endian. Contrairement aux processeurs x86 ou x86_64 gérant leur mémoire en Little Endian.

Principe d'implémentation

Dans un premier temps, il est tout d'abord nécessaire de pouvoir lire le fichier de code fourni par l'utilisateur. Cette lecture se charge de créer un premier tableau, le tableau 0, contenant le code à exécuter par la machine virtuelle. Par la suite, le PC est initialisé sur le premier mot du tableau 0, et le code est ensuite executé.

L'exécution se décompose en plusieurs étapes :

1. Lire le mot du code.
2. Incrémenter le pointeur de code.
3. Extraire l'opcode de l'opération.
4. Extraire les différentes valeurs contenus dans le mot selon l'opération (A, B, C ou un Immédiat).
5. Exécuter l'instruction.

L'exécution complet du code consiste donc à boucler sur les différentes étapes dans l'ordre, afin d'obtenir le résultat escompté.

Implémentation

Avec les spécifications et les principes de l'implémentation connues, il a été possible de transcrire le tout dans un langage de programmation afin de pouvoir faire exécuter le code à une machine. Dans un premier temps, le choix du langage a été crucial.

Choix du langage

Afin de pouvoir mettre en pratique ces principes, il fut nécessaire de choisir un langage de programmation adapté. A cet effet, deux implémentations ont été réalisées. une première implémentation en Ruby – parfaitement fonctionnelle – s'est vue bien trop lente pour être utilisée. Cette implémentation exécute parfaitement les tests et programmes, mais à une vitesse bien trop lente (plusieurs heures pour obtenir un résultat).
Un second choix s'est donc porté sur le C, afin de profiter de la manipulation bas niveau du processeur pour augmenter la vitesse d'exécution.

Implémentation de la spécification

Afin de transcrire la machine virtuelle dans un langage informatique, plusieurs choix ont été faits pour les spécifications :

- La mémoire globale se présente comme un tableau de pointeurs sur les tableaux (de code, ou alloués par la machine).

- Les registres sont implémentés dans un tableau de registres, stockant chacun un mot de 32 bits.
- Le PC est une variable stockant le l'index du programme.

L'exécution du code en lui-même repose sur l'écriture d'une boucle d'exécution à l'aide de goto calculés.
Le programme commence par disposer la mémoire, les registres et le PC à zéro, à lire le fichier fourni pour extraire le code, à inverser le code afin d'obtenir du Big Endian puis exécute l'instruction `next();`.
Celle-ci se charge de lire le mot actuel en mémoire, en extrait l'opcode, saute sur le label de l'instruction concerné, et extrait les différentes valeurs contenues dans l'instruction (A, B, et C), puis incrémente le PC.
Le code est ensuite exécuté, puis un nouveau `next();` est appelé à la fin de chaque bloc d'instructions afin de passer à l'itération suivante.

Les goto calculés sont une spécifité du C, permettant de créer un tableau de labels, et de sauter immédiatement sur le label en question, pour exécuter le code correspondant. Ainsi, il est possible de labeler chaque instruction par un label unique, et de référencer ce label dans un tableau, avec un index identique à l'opcode de l'instruction. En accédant à l'index [opcode] du tableau, on obtient donc l'adresse du bloc d'instructions à exécuter. De plus, chaque bloc se termine par l'instruction `next();`, se chargeant d'effectuer le goto.
Un tel dispositif nous évite d'avoir à utiliser une boucle while-switch :

```
while (true) {
  switch(opcode) {
    case x:
      break;
    .
    .
    .
  }
}
```

Une telle boucle, bien que tout à fait adapté, est sujette à un problème de branch prediction : les processeurs modernes effectuent une prédiction de branche à chaque test, mettant en mémoire la prochaine instruction par défaut. Lors d'un switch, le processeur va tenter de mettre en mémoire chaque branche, puisque chaque `case` est en réalité un test `if`. La branch prediction va donc demander un coût réel au processeur, puisque celui-ci doit ensuite "retourner en arrière" à chaque essai. Les goto calculés évitent donc cet écueil, en se plaçant immédiatement à l'emplacement du code à exécuter.

Spécifités de l'implémentation

Comme il a été précisé, l'implémentation gère la mémoire comme une collection de tableau, et plus exactement comme un tableau de tableaux d'entiers (`int`

`**memory`).
En C, chaque tableau a une taille fixe, et non extensible. Il est donc nécessaire, lorsque le programme demande plus de tableaux que la place disponible dans la mémoire globale, de réallouer la mémoire. Un `realloc` est donc effectué afin d'étendre la taille de la mémoire. Cela peut donc forcer – parfois – à réallouer un tableau complet et à recopier ses valeurs dans ce tableau (ce qui prends un temps important), mais cela permet le reste du temps d'accèder au tableau d'index `i` en temps constant. La réallocation de tableau prendra donc un peu de temps lors de l'exécution, mais permet de conserver une certaine rapidité lors de l'exécution (plutôt que de parcourir une liste chaînée par exemple).

Il faut également noter que la libération d'un tableau en mémoire entraîne la disponibilité immédiate de l'index associé à ce tableau pour un nouveau tableau. Afin de gérer ce cas de figure, une itération sur la mémoire globale est effectuée, pour trouver la première place disponible. Bien que naïf, il a été retenu de parcourir la mémoire "de l'avant" : le parcours de la mémoire ne se fait qu'en avançant, en repartant du dernier index alloué. Par exemple, si le dernier tableau possède l'index 40, alors la recherche commencera à 41. Cela évite d'avoir à reparcourir l'ensemble du tableau depuis 0. Le retour au début du tableau ne se fait qu'à condition qu'aucun index libre n'a été trouvé auparavant. Dans le pire des cas (un `realloc` donc), trouver un index libre prendra toutefois un temps linéaire à la taille du tableau.
Le choix a été fait de ne pas implémenter de freelist, pour ne pas alourdir l'exécution avec la création d'une liste chaînée, la gestion de cette liste, etc., les tests allant pour une mémoire globale de grande taille, avec un index s'incrémentant, la majorité des tableaux alloués par le passé étant rapidement libéré (et donc des index contiguë libérés).

Enfin, aucune compaction n'est possible pour la mémoire, puisqu'il faudrait utiliser une barrière d'abstraction des index : il est impossible de modifier tous les endroits où les index de ces tableaux sont conservés, obligeant à utiliser une barrière d'abstraction comme peut le faire la JVM.

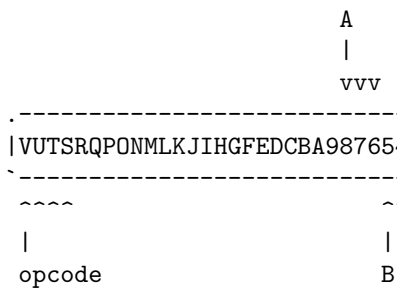
Conclusion

La mise en pratique de cette machine virtuelle demande donc de comprendre son fonctionnement interne, et de pouvoir identifier les difficultés inhérentes, afin de pouvoir passer outre pour garder des performances respectables. Effectivement, les langages de plus haut niveaux (Ruby, Perl, Python, etc.) permettent de se débarasser d'une bonne partie des problèmes, mais se révèlent bien trop lents pour être utilisés comme machine virtuelle, obligeant à utiliser un langage plus proche de la machine.

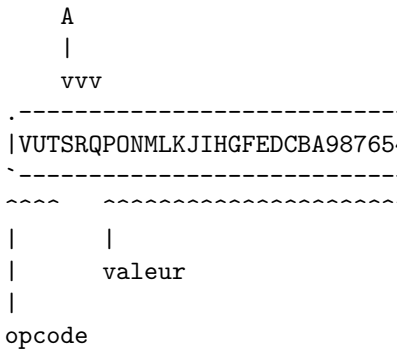
Annexes

Description des instructions

Chaque instruction se décompose selon la manière suivante :



L'instruction Orthography suit une règle différente :



Liste des instructions

0. **Conditional Move** : The register A receives the value in register B, unless the register C contains 0.
1. **Array Index** : The register A receives the value stored at offset in register C in the array identified by B.
2. **Array Amendment** : The array identified by A is amended at the offset in register B to store the value in register C.
3. **Addition** : The register A receives the value in register B plus the value in register C, modulo 2^32.
4. **Multiplication** : The register A receives the value in register B times the value in register C, modulo 2^32.
5. **Division** : The register A receives the value in register B divided by the value in register C, if any, where each quantity is treated treated as an unsigned 32 bit number.

6. **Not-And** : Each bit in the register A receives the 1 bit if either register B or register C has a 0 bit in that position. Otherwise the bit in register A receives the 0 bit.

The following instructions ignore some or all of the A, B and C registers.

7. **Halt** : The universal machine stops computation.
8. **Allocation** : A new array is created with a capacity of platters commensurate to the value in the register C. This new array is initialized entirely with platters holding the value 0. A bit pattern not consisting of exclusively the 0 bit, and that identifies no other active allocated array, is placed in the B register.
9. **Abandonment** : The array identified by the register C is abandoned. Future allocations may then reuse that identifier.
10. **Output** : The value in the register C is displayed on the console immediately. Only values between and including 0 and 255 are allowed.
11. **Input** : The universal machine waits for input on the console. When input arrives, the register C is loaded with the input, which must be between and including 0 and 255. If the end of input has been signaled, then the register C is endowed with a uniform value pattern where every place is pregnant with the 1 bit.
12. **Load Program** : The array identified by the B register is duplicated and the duplicate shall replace the '0' array, regardless of size. The execution finger is placed to indicate the platter of this array that is described by the offset given in C, where the value 0 denotes the first platter, 1 the second, et cetera.
13. **Orthography** : The value indicated is loaded into the register A forthwith.