Sorbonne Université
Faculté des Sciences et Ingénierie
Departement d'Informatique

Interpréteur et Compilateur pour la Machine Universelle

Dans le cadre de l'UE Compilation Avancée

Auteur
Amine BENSLIMANE

Master 1 STL 2020/ 2021
28 mars 2021

Table des matières

0.1 Présentation du sujet 1
0.2 Langage de programmation 1
0.3 Inspiration 1

1 Machine Universelle **2**
1.1 Description de la Machine Universelle 2
1.2 Opérateurs de la Machine Universelle 3
1.3 Description des opérateurs 3
1.4 Stratégie d'implementation 4
1.5 Evaluation de l'approche 5
1.6 Architecture de tests 6
1.7 Conclusion 6

2 Compilateur de microJS **7**
2.1 Exemple de compilation manuelle 7

0.1 Présentation du sujet

Le travail demandé est la réalisation d'un interpréteur de bytecode et d'un compilateur pour une machine universelle. Cette machine universelle est inspirée d'un sujet de la conférence *ACM International Conference on Functionnal Programming* de 2006.

Le travail demandé est divisé en deux taches principales :

- Implémentation d'un interpréteur de bytecode de la Machine Universelle
- Implémentation d'un compilateur du langage *MICROJS* qui génère ensuite du bytecode de l'UM.

0.2 Langage de programmation

Nous choisissons le langage JAVA pour réaliser ce travail. Ce choix de ce langage est raisonnable quant aux performances connus de la littérature pour exécuter les bytecodes dont il jouit. De plus, la gestion implicite de la mémoire facilite cette implémentation, à défaut de devoir la gérer par exemple avec le langage C.

0.3 Inspiration

La documentation étant vaste pour cette machine universelle et hormis celle indiquée dans l'énoncé du projet, on s'est inspiré de diverses sources. On peut citer notamment :

- Guillaume Hivert
- Mniip
- Garett Tarczuk, Phillip Ng
- Stackoverflow (parsing file -> byte)

Chapitre 1

Machine Universelle

Dans ce chapitre, nous présentons le système à réaliser (en l'occurrence la machine universelle), ses composants, ses instructions et opérateurs mais aussi la stratégie d'implémentation que nous avons choisi. Nous finissons par des conclusions et perspectives sur le travail réalisé.

1.1 Description de la Machine Universelle

Nous entendons dans ce travail par Machine Universelle (UM), l'instanciation d'une machine abstraite recevant et interpretant des instructions de 32 bits. Elle se compose de :

— Une quantité infinie de plateaux de 32 bits illustrée dans la figure 1 qui suit :

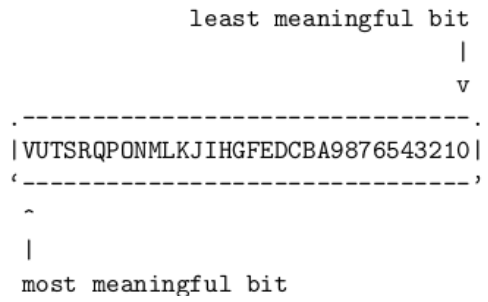

FIGURE 1.1 – Plateau de 32 bits

— Un lot de 8 registres de 32 bits

— Une collection de références vers des tableaux de plateau (Mémoire globale)

— Une console d'output/input de code ASCII

— Un registre contenant l'instruction courante (Program Counter)

Avec ces éléments, la machine universelle présente un système formel complet au sens de Turing "TURING-COMPLET" et peut donc répondre à l'interprétation des diverses instructions distinguées par des "OPÉRATEURS" que l'on décrit dans la section suivante.

1.2 Opérateurs de la Machine Universelle

Cette UM se compose de 14 opérateurs distincts et donc de 14 types d'instructions possibles à interpréter. Chaque instruction est composée d'un OPCODE codé sous les 4 bits de poids fort faisant référence à l'opérateur et de trois registres A, B et C prenant chacun 3 bits comme illustré ci-dessous :

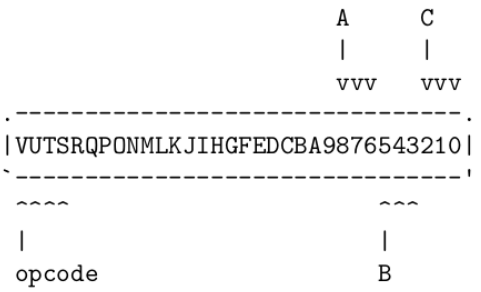

FIGURE 1.2 – Codage d'une instruction

A l'exception de "ORTHOGRAPHY" qui ne prend qu'un registre comme illustré ci-dessous :

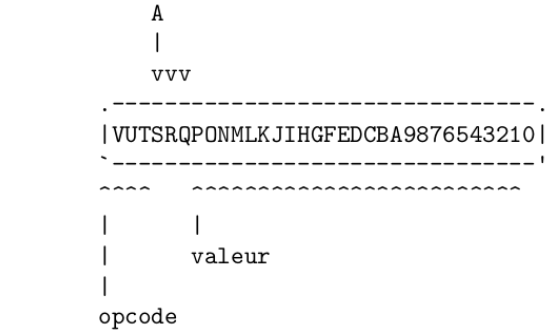

FIGURE 1.3 – Codage de l'instruction Orthography

1.3 Description des opérateurs

Nous présentons la liste des opérateurs dits "standards" :

- CMV : Le registre A reçoit la valeur du registre B sauf si le registre C est nul (0).
- AINDX : Le registre A reçoit la valeur stockée à l'offset dans le registre C dans le tableau identifié par B.
- AAMD : Le tableau identifié par A est modifié au décalage dans le registre B pour stocker la valeur dans le registre C.
- ADD : Le registre A reçoit la valeur du registre B plus la valeur dans le registre C, modulo 2^32.
- MUL : Le registre A reçoit la valeur du registre B multipliée par la valeur dans le registre C, modulo 2^32.
- DIV : Le registre A reçoit la valeur du registre B divisée par la valeur dans le registre C, le cas échéant, où chaque quantité est traitée comme un numéro 32 bits non signé.
- NAND : Chaque bit du registre A reçoit le bit 1 si l'un ou l'autre des registres B ou le registre C a un bit 0 dans cette position. Sinon le bit dans le registre A reçoit le bit 0.

— HALT : La machine universelle arrête le calcul.

— ALLOC : Un nouveau tableau est créé avec une capacité de plateaux commensurable à la valeur du registre C. Ce nouveau tableau est entièrement initialisé avec des plateaux ayant la valeur 0. Un modèle de bits ne se composant pas exclusivement de bit 0, et qui n'identifie aucun autre tableau alloué actif, est placé dans le registre B.

— ABANDON : Le tableau identifié par le registre C est abandonné. Les allocations futures pourront alors réutiliser cet identifiant.

— OUT : La valeur dans le registre C est affichée sur la console immédiatement. Seules les valeurs comprises entre 0 et 255 inclus sont autorisées.

— IN : La machine universelle attend une entrée sur la console. Dès que l'entrée arrive, le registre C est chargé avec l'entrée, qui doit être comprise entre 0 et 255. Si la fin de l'entrée a été signalée, alors le le registre C est doté d'un modèle de valeur uniforme où chaque place est enceinte avec le bit à 1.

— LPROG : Le tableau dont l'identificateur est B est dupliqué et sa copie remplace le tableau '0', quelle que soit sa taille. L'indice d'exécution doit être placé sur le plateau dont l'indice est contenu par le registre C.

— ORTH : Syntaxiquement spéciale (FIGURE 1.3), cette instruction stock la valeur indiquée dans le registre voulu.

1.4 Stratégie d'implementation

Il s'agit d'abord de lire un à un les plateaux d'instructions (découper le fichier), décoder ces dernières en reconnaissant le bon OPCODE jusqu'à la fin du fichier.

Nous représentons les registres sous forme de tableau d'entiers d'une capacité de 32 bits, et la mémoire globale comme une collection (dictionnaire) de pointeurs sur les tableaux. (la JVM s'occupant de la gestion de la mémoire).

Bien sûr et comme tout bon langage de "bas niveau", nous avons eu recours à un Program counter (PC), souvent appelé registre d'instruction (RI) dans la littérature, qui stock l'instruction courante à exécuter, en l'occurrence le plateau parsé courant.

Le programme commence par une série d'initialisation où les registres et le PC sont mis à zéro. Ainsi, l'indice pointera vers le premier plateau qui doit avoir l'indice zéro. On parsera alors à chaque fois 4 bytes (4 blocs de 8 bits) correspondants à une seule instruction. On veille à ce que le PC s'incrémente correctement pour passer à l'instruction suivante, si elle existe.

L'interprétation dépendra alors de l'OPCODE. Ainsi, nous "SWITCHONS" sur l'ensemble des valeurs que peut prendre ce dernier en effectuant la bonne interpretation à chaque fois. Nous avons préféré traité plus tôt dans le code l'OPCODE "ORTHO" vu que l'on shift différemment pour obtenir le registre et la valeur à y stocker. la boucle prend la forme suivante :

```
while(pc < longueur(plateau)){

switch(OPCODE){
```

```
case op1 :
    interpreter(op1);
    .
    break;
case op2 :
    .
    .
    break;
.
.
pc++;
}

}
```

A part pour HALT où lever une RuntimeException suffit pour la traiter, nous shiftons en masque logique pour obtenir les registres A, B et C sur lesquels s'effectueront les calculs.

La condition d'arrêt de parsing (recherche d'opcode) est que le PC dépasse la taille du nombre de plateaux initials. (ou encore qu'il l'atteigne, car début de 0)

Le schéma de notre interpretation est donc représenté par l'algorithme suivant :

Algorithm 1 Interprétation des instructions

Require: Fichier d'instructions
Ensure: Interpretation des instructions

- 1. Initialisation (parseur, PC, registres..)
- 2. Découpe du programme en instructions de 4 byte

while ∃ instruction **do**

- 3. Interpreter l'instruction courante
- 4. Incrémenter PC vers la prochaine instruction

end while

1.5 Evaluation de l'approche

La stratégie de reconnaissance de l'opcode répond parfaitement au scénario nominal de l'interprétation mais cependant n'évite pas le problème de Branch Prediction où le processeur met en mémoire chaque branch et ainsi prend un temps de calcul non négligeable.

Une meilleure implémentation ou du moins plus optimale, consisterait à se placer directement à l'emplacement du code à exécuter.

1.6 Architecture de tests

Pour tester notre programme, nous l'avons confronté au benchmark de test fourni SANDMARK.UMZ où notre programme a sû produire les résultats attendus en un temps raisonnable ($\approx$ 1 min en i3 7^{th} 8GB) et donc validé notre démarche. Ci-dessous une capture des résultas : (Execution en 63 s)

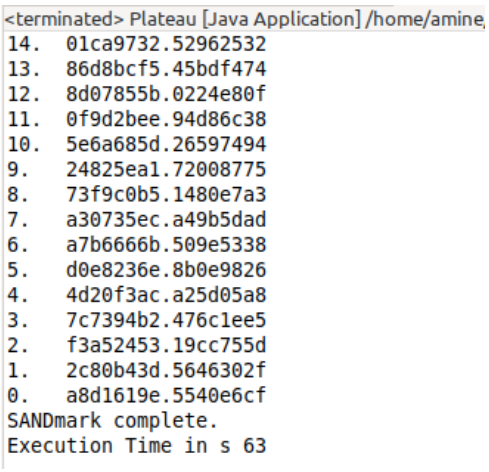

FIGURE 1.4 – Résultat de l'execution du benchmark SANDMARK.UMZ

1.7 Conclusion

La réalisation du système dégage les différents aspects "compliqués" d'une machine universelle même avec peu d'opérateurs. La compréhension du système doit être rigoureuse pour garder des performances raisonnables. Quant à la réalisation, de plus en plus que le langage est proche de la machine (bas niveau) et plus les performances seront prononcées.

Chapitre 2

Compilateur de microJS

Ce chapitre est dédié à la réalisation du compilateur microJS vers notre Machine Universelle. Faute de temps, cette partie n'a pas été implémentée.

2.1 Exemple de compilation manuelle

Nous présentons dans cette section la compilation manuelle de diverses expressions microJS que voici :

— $> 2 + 3$

```
ORTHO r1 2
ORTHO r2 3
ADD r2 r1 r2
```

— $> 2 + 3 + 4$

```
ORTHO r0 2
ORTHO r1 3
ORTHO r2 4
ADD r1 r0 r1
ADD r2 r1 r2
```

— $> 2000000000 + 2$

```
ORTHO r1 2000000000
ORTHO r2 2
ADD r1 r2
```