

Rappels de COO/POO : Conception et implémentation d'un émulateur de machine à pile

Sébastien Jean

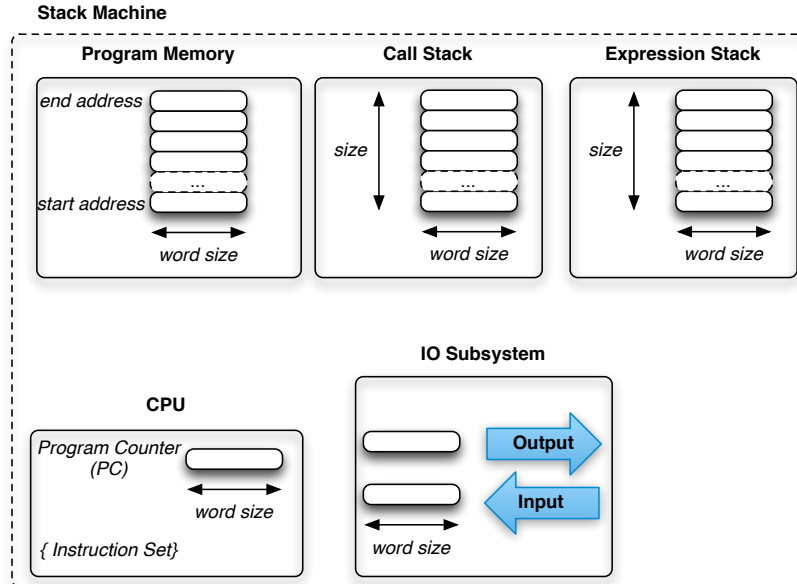
L'objectif de ce TP est de consolider les bases de COO et POO abordées dans les modules M2103 et M2104, en se ré-appropriant les outils de modélisation, de programmation et de gestion de version. Le sujet est inspiré d'un exercice original de Christophe Deleuze¹, avec son aimable courtoisie.

Contexte

On s'intéresse dans la suite à concevoir et réaliser un émulateur de machine à pile². Ces machines restent conformes à l'architecture Von Neumann³, dans la mesure où les programmes et données sont stockées dans une mémoire à laquelle accède le processeur qui connaît à chaque instant l'adresse de la prochaine instruction à exécuter (l'ensemble des instructions supportées par le processeur est appelé *jeu d'instructions*). La particularité des machines à pile réside dans l'existence d'un espace mémoire séparé pour les données et structuré sous forme de pile (dite d'expression ou *expression-stack*), dans laquelle les instructions supportées par le processeur vont puiser leurs arguments et déposer leurs résultats. Dans la plupart des cas où les machines supportent l'exécution de sous-routines (que l'on peut assimiler à des appels de fonctions), une autre pile (dite pile d'appel ou *call-stack*) permet de mémoriser les adresses de retour.

Architecture de la machine à pile

L'architecture générale de la machine à pile à émuler est présentée ci-dessous :



La **mémoire de programme** (*Program Memory*) est utilisée pour le stockage des programmes (cf. plus loin la description d'un programme). C'est une mémoire de taille fixe et à accès aléatoire, organisée sous la forme d'une suite de cellules mémoire contenant chacune un mot de taille fixe (*word size*) égale à 32 bits et possédant une adresse unique sur 32 bits (comprise dans l'intervalle $[startaddress, endaddress]$).

1. cf. <http://christophe.deleuze.free.fr/esisar05/CS410/mp.pdf>
2. cf. https://en.wikipedia.org/wiki/Stack_machine
3. cf. https://en.wikipedia.org/wiki/Von_Neumann_architecture

La **pile d'expression** (*Expression Stack*) est utilisée pour le stockage temporaire des données. C'est une mémoire de taille fixe et à accès LIFO (*Last In First Out*), organisée sous la forme d'une suite de cellules mémoire contenant chacune un mot de taille fixe (*word size*) égale à 32 bits.

La **pile d'appel** (*Call Stack*) est utilisée pour le stockage temporaire des adresses de retour lors de l'appel aux sous-routines, possédant les mêmes caractéristiques que la pile d'expression.

Le **processeur** (*CPU*) est caractérisé par un jeu d'instruction (*Instruction Set*) et possède un registre interne nommé PC (*Program Counter*) qui lui permet de maintenir l'adresse de la prochaine instruction à exécuter. A chaque cycle, le processeur :

- lit l'instruction située dans la mémoire de programme à l'adresse contenue dans le registre PC,
- décode cette instruction (cf. plus loin la description du jeu d'instructions),
- lit éventuellement les paramètres de l'instruction se trouvant aux adresses suivantes,
- exécute l'instruction, en allant chercher les opérandes et/ou empiler le résultat dans la pile d'expression et/ou en allant empiler ou dépiler une adresse de retour dans la pile d'appel,
- met à jour le contenu du registre PC (adresse de l'instruction à exécuter au cycle suivant)

Le **sous-système d'entrées-sorties** (*IO Subsystem*) permet d'échanger des données avec l'extérieur (ce qui se résumera ici à lire ou écrire une valeur (de taille *word size*) depuis ou vers la console.

L'ensemble des instructions supportées par le processeur est listé dans le tableau fourni en annexe. Les codes des mnémoniques sont des entiers sur 32 bits, les valeurs des éventuels paramètres sont également sur 32 bits.

Un programme exemple est donné également en annexe. Ce programme lit un nombre, calcule la factorielle de ce nombre et l'affiche.

N.B. : l'écriture d'un programme suppose le choix d'une adresse de base où le programme est censé être chargé (dans la mémoire de programme), ici les programmes ne sont par conséquent pas relogeables. Dans le programme exemple, cette adresse est 0x0. Les adresses spécifiées dans les instructions de saut sont absolues et doivent donc être cohérentes avec l'adresse de base du programme (ce qui est le cas dans le programme exemple).

Travail à réaliser

Dupliquer sur votre compte GitLab interne (et non cloner) le dépôt dont l'URL est la suivante : <https://gitlab.iut-valence.fr/INFO.DUT.M3105/StackMachine-UML-Java>

Dans ce dépôt se trouve un projet *Eclipse* contenant un squelette incomplet du code de l'émulateur de machine ainsi qu'une application permettant le lancement du programme de calcul de factorielle.

Terminaison de l'Implémentation de base de l'émulateur

En travaillant sur le code fourni :

- Produire le diagramme de classes complet de l'émulateur à l'aide de l'outil de votre choix.
- Compléter le code afin de pouvoir calculer 7!

Le diagramme de classes devra être placé dans un sous-répertoire UML du dépôt. Il est préférable d'avancer en parallèle sur le code et le diagramme (i.e. commencer par une rétro-conception sur la base du code fourni, puis mettre à jour le diagramme tout en complétant l'implémentation).

Amélioration de l'implémentation, élévation du niveau d'abstraction

Une fois l'étape précédente réalisée, reprendre la conception et l'implémentation, en montant d'un niveau d'abstraction pour augmenter la réutilisabilité.

Considérer tout d'abord qu'il est préférable d'exprimer les relations entre **Machine** (d'une part) et **CPU/Memory/Stack/IO** (d'autre part) sous la forme d'une composition. Essayer ensuite d'abstraire au maximum la notion de processeur, de pile, de mémoire, de sous-système d'entrée-sortie, en conservant le même niveau de fonctionnalité (i.e. l'application fournie devra produire le même résultat) mais en conservant la possibilité d'utiliser des mémoires ou des piles stockant des mots de 8, 16 bits ou 32 bits, d'utiliser un sous-système d'entrée-sortie effectuant des redirections depuis et vers des fichiers, ...

Annexe 1 : le jeu d'instructions du processeur

Mnemonic	Code Op.	Param	Effect
HALT	0x0	-	Terminate program
PUSH	0x1	<i>value</i>	Push a value on top of expression stack
ADD	0x2	-	Pop two values from expression stack, add the 1st to the 2nd, push the result on expression stack
SUB	0x3	-	Pop two values from expression stack, subtract the 1st to the 2nd, push the result on expression stack
MUL	0x4	-	Pop two values from expression stack, multiply the 1st by the 2nd, push the result on expression stack
DIV	0x5	-	Pop two values from expression stack, divide the 2nd by the first, push the result on expression stack
MOD	0x6	-	Pop two values from expression stack, compute the modulus of euclidian division of the 2nd by the 1st, push the result on expression stack
NEG	0x7	-	Pop a value from expression stack, push the opposite on expression stack
LT	0x8	-	Pop two values from expression stack, push 0x0 on expression stack if the 2nd is lower than the first or 0x1 if not
LE	0x9	-	Pop two values from expression stack, push 0x0 on expression stack if the 2nd is lower than or equal to the first or 0x1 if not
GT	0xA	-	Pop two values from expression stack, push 0x0 on expression stack if the 2nd is greater than the first or 0x1 if not
GE	0xB	-	Pop two values from expression stack, push 0x0 on expression stack if the 2nd is greater than or equal to the first or 0x1 if not
EQ	0xC	-	Pop two values from expression stack, push 0x0 on expression stack if both are equal or 0x1 if not
NE	0xD	-	Pop two values from expression stack, push 0x0 on expression stack if both are different or 0x1 if not
IN	0xE	-	Read a value from IO, push this value on expression stack
OUT	0xF	-	Pop a value from expression stack, write this value to IO
CALL	0x10	<i>address</i>	Push next PC on call stack, set PC with address given as parameter
RET	0x11	-	Pop an address from call stack, set PC with this address
JP	0x12	<i>address</i>	Set PC with address given as parameter
JZ	0x13	<i>address</i>	Pop a value from expression stack, set PC with address given as parameter only if this value is 0x0
DUP	0x14	-	Duplicate the value on top of expression stack (i.e. pop a value and push it twice)
POP	0x15	-	Pop a value from expression stack, and forget it

Annexe 2 : programme de calcul de la factorielle d'un nombre

N.B : La première colonne représente l'adresse du code opération de l'instruction (l'adresse de chargement supposée du programme est donc 0x00000000).

```
0x0 : in          // read a number n from IO
0x1 : call 0x5     // call the subroutine that computes n!
0x3 : out         // print n!
0x4 : halt        // stop the program
0x5 : dup
0x6 : jz 0xb       // if n = 0, computing n! is trivial
0x8 : call 0xf     // else n! = n * (n-1)!
0xa : ret
0xb : pop         // when n = 0 n! = 1 ...
0xc : push 0x1    // ... so n is replaced by n!
0xe : ret
0xf : dup         // ...
0x10: push 0x1    // ...
0x12: sub         // compute n-1
0x13: call 0x5    // call the subroutine that computes (n-1)!
0x15: mul         // compute n! = n*(n-1)!
0x16: ret
```