IFT-209

# Programmation système

Automne 2019
Devoir #5

*Thèmes : manipulation de chaînes de bits, récursivité, utilisation de la pile, interface avec le langage C.*

## *Mise en situation*

Un des derniers domaines où la connaissance des langages d'assemblage est cruciale est l'écriture de compilateurs. Bien que les compilateurs modernes soient fabriqués avec des outils flexibles permettant de passer facilement d'une architecture à une autre, il n'en demeure pas moins que le comportement et l'utilité de chaque instruction de l'architecture ciblée doivent être bien connus de ceux qui concoivent le dit compilateur.

Plusieurs types d'énoncés peuvent se retrouver dans les langages évolués, certains servant à construire des structures de contrôle (if, while, for, switch), d'autres, déclarer des types données, écrire des expressions booléennes ou réaliser de simples affectations.

## *Compilation d'expressions arithmétiques*

Un de ces types d'énoncés concerne l'arithmétique entière avec les opérateurs usuels (+,-,*,/), en notation infixée (l'écriture habituelle). Dans la majorité des langages évolués, on peut écrire des expressions telles que : 4+5-6, ou bien, (4-5)*7+9, par exemple.

Transformer ces expressions en code machine n'est cependant pas une tâche simple. Tout d'abord, la notation infixée habituelle se prête mal à une génération directe d'instructions machine. La priorité des divers opérateurs doit être connue et prise en compte lors de l'évaluation de l'expression. Cependant, si l'expression est écrite dans la notation dite ***polonaise inverse***, ce problème disparaît. De plus, la majorité des compilateurs, avant de passer à la génération d'instructions machine, construisent un ***arbre syntaxique***. Cette structure représente l'ordonnancement des opérations que l'on peut traduire en code machine et facilite la génération de code. Un parcours de l'arbre permet de générer le code dans le bon ordre et de calculer la position des divers énoncés par rapport aux autres facilement (pour les branchements).

***Tâche à réaliser***

Pour réussir ce devoir, vous devez réaliser en assembleur de ARM v8 la partie génération de code d'un compilateur rudimentaire pour une machine à pile virtuelle.

Votre programme compilera uniquement des expressions arithmétiques contenant des **constantes positives** et les quatre opérateurs +,-,* et /, plus les parenthèses pour indiquer la priorité.
L'expression ne peut pas contenir d'espaces ou de nombres négatifs. Elle peut contenir des nombres positifs à plusieurs chiffres, jusqu'à concurrence de 32767 ($2^{15}$-1).

Afin de transformer votre expression en un arbre syntaxique, on vous fournit un programme principal qui utilise quelques fonctions déjà écrites. Une fois l'arbre syntaxique généré, votre sous-programme se nommant ***Compile*** est appelé. L'arbre syntaxique est représenté à l'aide d'un struct en langage C.

## La structure **Nœud**

Cette structure représente un nœud de l'arbre syntaxique. Chaque nœud peut contenir soit un opérateur, soit un nombre. Les opérateurs ont toujours deux enfants, les branches gauche et droite de l'arbre sous eux-mêmes. Les nombres n'ont jamais d'enfants.
La définition de la structure Noeud est écrite dans **analyse.h**.

La structure Nœud est organisée comme suit en mémoire :
octets 0 à 3 : un entier désignant le type de nœud (0=nombre, 1=opérateur).
octets 4 à 7 : un entier contenant la valeur associée au nombre ou à l'opérateur (entier signé ou numéro d'opérateur, respectivement).
octets 8 à 15 : un mot étendu contenant un pointeur de 64 bits, l'adresse du premier nœud du sous-arbre de gauche du nœud courant. La valeur 0 indique le pointeur NULL (aucun sous-arbre).
octets 16 à 23 : un mot contenant un pointeur de 64 bits, l'adresse du premier nœud du sous-arbre de droite du nœud courant. . La valeur 0 indique le pointeur NULL (aucun sous-arbre).
Les numéros d'opérateurs inscrits dans la structure Nœud sont : 0 = ADD, 1 = SUB, 2 = MUL, 3 = DIV. ***Ces numéros sont différents des valeurs du champ « oper » des instructions de la machine virtuelle. Vous devez opérer une conversion pour générer correctement le code.***

Pour compléter le compilateur, vous devez produire un fichier **tp5.as** contenant la fonction de génération de code. Cette fonction (et les sous-programmes dont vous pouvez avoir besoin) se trouveront dans le fichier tp5.as, et devra s'appeler **Compile**.

Votre fonction Compile doit répondre aux critères suivants :

1- Retourner la longueur du code machine produit en octets. Pour retourner une valeur qui se représente sur un mot, vous pouvez utiliser le registre x0.

2- Parcourir l'arbre syntaxique et fabriquer les instructions machine, en suivant l'algorithme ci-après, en **respectant la spécification du code machine fournie en annexe**. Le code machine devra être écrit dans un tableau d'octets, (le deuxième paramètre de Compile).

### *Génération de code*

Afin de générer le code machine, vous pouvez suivre l'algorithme en pseudo-code suivant :

```
Compile (Nœud)
{
    si Nœud contient un nombre
    alors
        produire l'instruction PUSH nombre. Le nombre est un immédiat de 16
        bits. La taille du code produit est la taille de l'instruction PUSH.

    sinon
        Compiler le sous-arbre de gauche de Nœud;
        Compiler le sous-arbre de droite de Nœud;
        selon le numéro d'opération, produire ADD, SUB, MUL ou DIV.
        La taille du code produit est la taille du code produit par le sous-arbre de
        gauche, plus celle du code produit par le sous-arbre de droite, plus la taille
        de l'instruction pour l'opérateur se trouvant dans Noeud
}
```

Dans cet algorithme, le calcul de la longueur du code et la façon d'ajouter les instructions à la chaîne d'octets ne sont pas parfaitement expliqués. Ne les oubliez pas! Ces tâches s'ajoutent facilement au squelette d'algorithme plus haut et sont indispensables à la production de résultats corrects.

De plus, à la suite de l'expression, vous devez afficher le résultat du calcul à la fin à l'aide de l'instruction WRITE, et stopper la machine virtuelle à l'aide de l'instruction HALT.

Ces ajouts ne sont pas faciles à inclure dans une fonction récursive, il vous est donc conseillé de faire une coquille (wrapper, adaptateur) pour réaliser cette étape.
Pour réaliser la production des instructions, référez-vous à la spécification du code machine fourni en annexe. Les modes d'adressage que vous allez utiliser sont le mode immédiat (PUSH, ADD, SUB, MUL, DIV) et le mode système (WRITE, HALT).

Pour tester vos résultats, vous disposez de deux moyens :

1- Comparez vos résultats à ceux fournis avec le devoir, dans /tests.

2- Redirigez votre affichage vers un fichier (par exemple, tp5 > test) et ensuite, passez ce fichier comme entrée à l'exécutable ***machine*** (machine <test). Cet exécutable est un simulateur rudimentaire pour la machine spécifiée en annexe. Il sera utilisé pour tester le code produit par votre compilateur. Si vous avez des problèmes à exécuter machine (permission denied), alors, exécutez la commande : **chmod +x machine**

***Quelques conseils pratiques***

Tout d'abord, lorsque vous voulez coder l'algorithme de la fonction Compile, vous pouvez vous concentrer uniquement sur le parcours de l'arbre syntaxique, comme vu en classe. Si vous voulez voir si le parcours se déroule correctement, commencez par faire de appels à *printf* qui affichent les instructions (PUSH, ADD, SUB, etc) et leurs opérandes. Si le parcours affiche les instructions dans le bon ordre, vous pouvez passer à la production d'instructions, que vous pouvez aussi afficher en format hexadécimal à l'aide de *printf.* Finalement, concentrez-vous à écrire les instructions au bon endroit dans la chaîne d'octets et à calculer le nombre d'octets que votre programme occupe. Chaque instruction a une certaine longueur en octets. ***Ne calculez pas la longueur du code une fois qu'il est complètement généré, mais bien durant la génération***. ***Retirez vos printf du code avant de terminer!***

## Ne modifiez pas analyse.h, tp5.cc et analyse.o !!!

Le simulateur pour la machine à pile virtuelle n'implante pas toutes les instructions mentionnées en annexe, mais bien celles que vous allez utiliser! Ne vous amusez pas à fabriquer des instructions non-implantées, vous risquez de trouver le comportement de la machine plutôt étrange!

Pour compiler votre travail, importez d'abord les fichiers du répertoire public dans votre répertoire de travail.
Une fois le fichier tp5.as écrit, procédez comme suit :

**make**

Bon Travail!

# Annexe 1
# Spécification des instructions pour une machine à pile virtuelle

Les instructions peuvent avoir plusieurs octets de long. Le premier octet de la suite (et parfois le seul) contient les informations sur l'instruction (mode d'adressage, type d'opérande, etc), les octets suivants contiennent l'opérande.

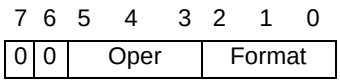
Les instructions ont quatre formats de base, selon le mode d'adressage utilisé. Les bits 6 et 7 du premier octet spécifient ce format.

00 : mode d'adressage de type « système », instructions spéciales et entrées/sorties;
01 : mode d'adressage immédiat : l'opérande est un immédiat de 16 bits;
10 : mode d'adressage direct : l'opérande est une adresse de 24 bits;
11 : mode d'adressage relatif : l'opérande est un déplacement de 13 bits.

Les instructions dans les modes 01 et 10 n'utilisent pas toutes l'opérande mentionné. Les instructions arithmétiques, par exemple, n'utilisent pas l'opérande; leurs opérandes sont déjà sur la pile d'exécution.

Format 00 :

Octet 0




Valeurs pour le champ Oper :

000 : HALT (Format=000)
001 : SAVE (Format=000)
010 : RESTORE (Format=000)
011 : READ
100 : WRITE

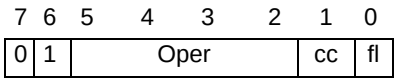
Valeur pour le champ Format (READ et WRITE)

000 : « %c »
001 : « %d »
010 : « %s »
011 : « %f »
100 : « %u »
101 : « %x »

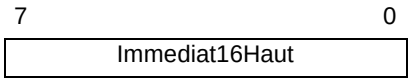
L'instruction HALT arrête l'exécution. Les instructions READ et WRITE fonctionnent comme printf et scanf, les formats étant identiques. Les valeurs à imprimer se trouvent sur le dessus de la pile.

Format 01 :

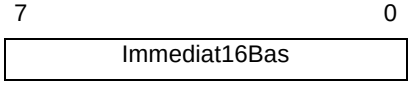
Octet 0




Octet 1

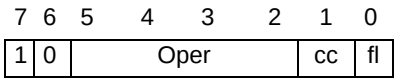



Octet 2

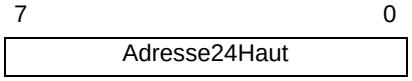



Format 10 :

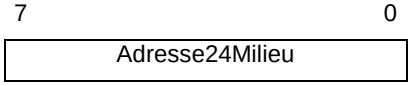
Octet 0



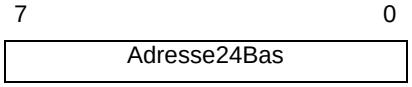

Octet 1




Octet 2




Octet 2




Ces deux formats sont utilisés pour les mêmes opérations.
Les instructions suivantes correspondent aux valeurs du champ Oper :

0000 : PUSH
0001 : POP
0010 : ADD
0011 : SUB
0100 : MUL
0101 : DIV
1111 : JMP

Seule l'instruction PUSH prend un immédiat de 16 bits dans le format 01.
***Les valeurs entières placées dans cet immédiat sont en représentation complément à 2 sur un demi-mot.***
Seules les instructions PUSH et POP prennent une adresse de 24 bits dans le format 10.
Les autres instructions auront donc uniquement l'octet 0 d'utilisé et ***leur longueur sera donc de 1 octet.***

Les instructions ADD, SUB, MUL et DIV et JMP en format 10 appliquent un mode d'adressage indirect. Les opérandes sur la pile sont considérés comme les adresses où se trouvent les nombres à calculer.

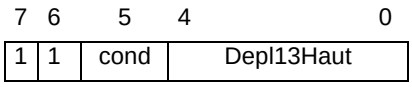
Les champs immédiat16 et adresse24 sont divisés en plaçant d'abord les bits les plus significatifs (8 à 15 ou 16 à 23 selon le cas) dans l'octet 1, jusqu'aux bits les moins significatifs (0 à 7) dans l'octet 2 ou 3, respectivement.

Le champ ***cc*** spécifie si l'instruction va modifier les codes condition ou non (1 = cc modifié, 0 = cc inchangé).

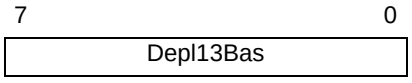
Le champ ***fl*** spécifie si l'on travaille avec des nombres en virgule flottante ou avec des nombres entiers (0 = entier, 1 = virgule flottante).

Format 11 :

Octet 0




Octet 1




Ce format est utilisé pour les instructions de branchement.
La valeur du bit 5 indique la condition :

0 : BZ (branchement si zéro)
1 : BNEG (branchement si négatif)

Le champ Depl13 est un déplacement en octets dans le code.  Ses bits 8 à 12 sont dans l'octet 0 et ses bits 0 à 7 sont dans l'octet 1.

Seules les instructions suivantes sont implantées dans le simulateur partiel disponible pour ce devoir :

HALT
PUSH (format 01)
ADD, SUB, DIV, MUL (format 01)
WRITE (seul le format "%d" est supporté)

Seuls les nombres entiers (positifs et négatifs) sont supportés.