IFT-209

Programmation système

Automne 2019

Devoir #6

Thèmes : manipulation de chaînes de bits, interface avec le langage C, nombres à virgule flottante, modes d'adressage, architecture.

Mise en situation

Un domaine qui prend une place de plus en plus importante en informatique est celui de l'émulation ou la simulation d'anciennes architectures matérielles. Un émulateur est un programme construit pour imiter un ou un ensemble de composants matériels et/ou logigiciels. On trouve plusieurs utilités aux émulateurs, que ce soit de revisiter d'anciennes consoles, de supporter du vieux matériel qu'on ne fabrique plus avec du nouveau, ou bien de relever le défi d'imiter une architecture pour laquelle on dispose de peu d'information.

Dans les années 90, les émulateurs pour les consoles de jeux vidéo et pour les anciens jeux d'arcade ont commencé à apparaître plus fréquemment et à fonctionner assez bien pour permettre aux amateurs de revisiter leurs anciens jeux préférés. Tout a commencé avec la distribution via internet de documentation sur les architectures (nestech.txt en est un fameux exemple), pour que des programmeurs voient la possibilité de coder des émulateurs intéressants pour les consoles les plus populaires. De fil en aiguille, les consoles et jeux d'arcade contenant des composants plus simples furent émulés d'abord, avec des techniques plutôt simples. Suivirent des consoles de deuxième génération, qui commencèrent à être émulées difficilement pendant qu'elles étaient toujours en vente, pour qu'apparaîsse finalement un émulateur qui allait faire réfléchir sérieusement les grands fabriquants de consoles : UltraHLE.

Cet émulateur hors du commun faisait fidèlement fonctionner des jeux de Nintendo64 alors qu'ils étaient sortis peu de temps auparavant. Le programme simulait une architecture pour laquelle l'information technique était la propriété de Nintendo, sans que les auteurs aient eu accès à ce matériel. Bien que depuis ce temps, le mur entre les consoles de jeux et les ordinateurs personnels se soit amenuisé (ils utilisent le même médium de stockage, par exemple, ou des composants matériels similaires), le domaine de l'émulation représente

encore un ensemble de défis difficiles à relever, autant pour les auteurs de ces logiciels, que pour les manufacturiers qui désirent se protéger du piratage.

De nos jours, les émulateurs nous entourent de plusieurs façons et nous les utilisons pour beaucoup de raisons diverses. Pensez seulement à VMWare pour construire des bacs à sable, Wine pour régler une partie du problème de compatibilité entre les systèmes d'exploitation, la Wii Virtual Console pour donner un accès légal et fidèle aux anciennes bibliothèques de jeux classiques, sans oublier des émulateurs fidèles et majeurs pour les anciens systèmes IBM 360/370 mainframe, sur lesquels on programme encore des applications de gestion bancaire! Pour tous ces systèmes, un équilibre entre la performance et la compatibilité (fidélité à l'ancien système) est recherché, et le gros du combat se déroule au niveau de la programmation système.

Fabrication d'un émulateur rudimentaire

Plusieurs types d'émulateurs existent, utilisant diverses techniques d'émulation, la plus répandue étant l'interprétation de code.

Cette technique est toute simple et représente l'approche la plus intuitive à l'émulation.
Comment imiter le comportement d'une autre architecture?
Il faut tout simplement lire les instructions d'un programme compilé destiné à cette architecture, les décoder et imiter le comportement de l'architecture pour chacune des instructions. À l'aide de structures de données représentant les différentes parties de l'architecture (registres, mémoire, etc), on peut représenter l'état de la machine virtuellement et tenter de simuler son comportement.

Pour réaliser cette tâche, on doit d'abord disposer d'un document technique qui explique le comportement exact de chaque instruction (**voir la spécification en annexe**). À partir de cette spécification, on peut concevoir des structures pour représenter l'architecture en question et commencer à coder les segments de code qui représentent le comportement des instructions.

Tâche à réaliser

Pour réussir ce devoir, vous devez réaliser en assembleur de ARM v8 le décodage des instructions de la machine à pile décrite en annexe, ainsi que les fonctions qui serviront à imiter les instructions suivantes : **PUSH, ADD, MUL, SUB, DIV, WRITE.**
Vous devrez également supporter le fonctionnement de ces instructions avec des ***nombres à virgule flottante.***

Vous devez supporter, en gros, toutes les instructions pouvant être générées par l'exécutable du devoir 5, en plus de leur version qui supporte les nombres à virgule flottante.

Pour réaliser cette tâche, vous devez modifier le fichier **tp6.as** (disponible sur le répertoire public) en y ajoutant le code manquant.
Ce fichier contient un ensemble de fonctions qui seront appelées par le fichier **machine.cc**.

Voici une description des fichiers du répertoire :

machine.cc/h

Ces fichiers définissent les structures utilisées par l'émulateur. De plus, ils contiennent du code qui implante le fonctionnement général de l'émulateur. Les fonctions **step** et **run** sont les plus intéressantes à comprendre. Lors de l'exécution d'un programme, on répète tout simplement le comportement suivant : on décode l'instruction suivante; on l'exécute, on vérifie si la machine doit s'arrêter et on recommence.
Pour exécuter une instruction, on utilise la structure produite par la fonction **Decode** pour décider quelle fonction d'émulation on va appeler. Chaque instruction a sa fonction d'émulation, elles sont toutes déclarées en haut de machine.cc et en haut de tp6.as.

Le fichier machine.h définit deux structures importantes **: machineState** et **Instruction**.

La structure **machineState** contient toute l'information importante sur l'état de la machine qu'une fonction d'émulation risque de modifier, ou que la fonction de décodage peut utiliser.
Ses champs sont les suivants :

-PC = la valeur courante du compteur ordinal, il représente une position dans la mémoire de la machine virtuelle.

-SP = la valeur du pointeur de pile de la machine virtuelle, il représente aussi une position dans la mémoire de la machine virtuelle.

-PS = la valeur des indicateurs d'état de la machine virtuelle. Seuls les bits 0 et 1 de PS sont utilisés, représentant respectivement les bits Z et N.

-memory = un pointeur sur une chaîne d'octets représentant la mémoire de la machine virtuelle.

Par exemple, si vous avez une instance de machineState se nommant state,

-state.memory[PC] est la valeur du premier octet de l'instruction courante.
-state.memory[SP] est la valeur de l'octet se trouvant sur le dessus de la pile.

La structure **Instruction** représente les champs d'une instruction décodée. Ces champs s'apparentent à ceux décrits en annexe.

-**mode** = les bits 6 et 7 de toutes les instructions. Représente le type d'instruction (spéciale, immédiate, indirecte ou branchement). Ce champ est utilisé pour déterminer une partie du comportement des instructions. Les modes que vous devez absolument supporter sont 00 (pour WRITE) et 01 (PUSH, ADD ,SUB, MUL, DIV).

-**operation** = le champ Oper de la plupart des instructions, représente le type de branchement, le type d'opération arithmétique, etc.

-**operand** = l'opérande de l'instruction. Dans le cas de PUSH et POP, cet opérande peut avoir 16 bits (entiers) ou 32 bits (floats). Dans le cas des branchements, c'est le champ déplacement (13 bits), dans le cas des instructions système, c'est le champ format. Les instructions ADD, SUB, DIV et MUL n'ont pas d'opérande.

-**cc** = l'instruction modifie-t-elle les codes condition? Seulement utilisé pour ADD, SUB, DIV et MUL. Vous n'êtes pas tenus d'implanter cette fonctionnalité.
-**fl** = l'instruction manipule-t-elle un nombre à virgule flottante ou non? Pour l'instruction PUSH, ce champ détermine la grandeur de l'opérande (16 ou 32 bits). Pour les instructions arithmétiques, ce champ détermine combien d'octets consommer sur la pile pour récupérer les opérandes (2 octets chacun si fl=0, 4 octets chacun si fl=1). Il détermine également quelles instructions seront utilisées pour simuler le calcul.

-**size** = le nombre d'octets utilisés par l'instruction. Par exemple, PUSH avec un entier (fl=0) en mode immédiat (mode=01) prend 3 octets. Avec mode = 01 et fl = 1, on occupe 5 octets. L'instruction ADD, par exemple, occupe toujours un octet, tandis que les branchements (BZ et BN) en prennent toujours 2.

Normalement, tous ces champs sont remplis par la fonction ***Decode***.

tp6.as

Ce fichier doit contenir le code de la fonction Decode, ainsi que celui de toutes les fonctions d'émulation implantées.

La fonction **Decode** reçoit deux paramètres : un pointeur sur une structure ***Instruction*** dans %i0 et un pointeur sur une structure ***machineState*** dans %i1.

Cette fonction doit utiliser la structure ***machineState*** pour récupérer l'instruction courante (mémoire, compteur ordinal…). Ensuite, elle doit décoder l'information se trouvant dans l'instruction selon la spécification fournie en annexe et remplir les champs de la structure ***Instruction*** correctement.

Les fonctions d'émulation reçoivent les mêmes paramètres que la fonction Décode. Cependant, dans leur cas, la structure Instruction contient déjà l'information qui décrit l'instruction

courante. Chacune de ces fonctions doit implanter le comportement de l'instruction correspondande.

Par exemple, le comportement de l'instruction SUB est le suivant :

```
Si (instruction.fl == 0)
        operande2 = Depile (2 octets)
        operande1 = Depile (2 octets)
        resultat = operande1 – operande 2
        Empile (resultat, 2 octets)
Sinon
        operande2 = Depile (4 octets)
        operande1 = Depile (4 octets)
        /* Ici, la soustraction se fera avec l'unité virgule flottante; il faudra lui transférer les
                opérandes */
        resultat = operande1 – operande2
        Empile (resultat, 4 octets)
```

Vous **DEVEZ** implanter deux fonctions supplémentaires dans tp6.as : Empile et Depile.
Ces deux fonctions vont vous servir à travailler avec la pile de la machine virtuelle de façon stricte.

Empile

Entrées : %i0, adresse de la structure machineState
%i1 : nombre d'octets à empiler
%i2 : les octets à empiler (placés côte à côte sur 32 bits).
Sorties : aucune.

Empile doit récupérer les octets dans le registre %i2 et le placer sur la pile, sans oublier de modifier SP de la structure machineState.

Depile

Entrées : %i0, adresse de la structure machineState
%i1 : nombre d'octets à dépiler

Sorties : %o0 : les octets dépilés (placés côte à côte sur 32 bits).

Dépile doit récupérer les octets sur le dessus de la pile et les placer côte à côte dans un registre, pour ensuite les retourner via %o0, sans oublier de modifier SP de la structure machineState.

NOTE : Pour l'instruction WRITE, vous ne devez implanter que les formats %d et %f.

Overdrive 110%

Afin de rendre les programmes interactifs, vous pouvez implanter l'instruction READ. Son implantation ressemble beaucoup à celle de l'instruction WRITE. Cependant, prenez garde à la redirection d'entrées à l'aide de fichiers de données (i.e. tp6 < test), vous devrez inclure les valeurs entrées dans le fichier, à la suite des instructions quand vous effectuez des tests.

Overdrive 120%

Afin de supporter l'utilisation de variables en mémoire, vous pouvez implanter le mode 10 (direct) pour l'instruction PUSH et l'instruction POP. L'instruction POP en format immédiat n'existe pas. Dans ce format, la valeur de l'opérande dans l'instruction est l'adresse où se trouve ce que l'on doit empiler (pour PUSH), ou l'adresse où l'on doit dépiler (pour POP).

Overdrive 125%

Afin de faire des programmes intéressants, il faut pouvoir boucler. Vous pouvez implanter les branchements (BZ et BN) ainsi que les codes condition. Faites bien attention! Le compteur ordinal (PC) est incrémenté de la longueur de l'instruction après son exécution dans le fichier **machine.cc**. Vous devez contrer cette incrémentation SANS MODIFIER **machine.cc**.

Overdrive 130%

Afin d'avoir un architecture minimalement utilisable pour coder, il faut ajouter le support pour l'appel et retour de sous-programmes. Implantez les instructions JMPL (jump and link) et RET, qui sauvegardent et récupèrent respectivement la valeur du compteur ordinal sur la pile.

Quelques conseils pratiques

Ne modifiez pas machine.cc et machine.h !!!

Commencez tout d'abord par essayer la machine avec la suite de code la plus simple :

(sur la console)
machine
Custom Pile Dumper v0.4b
Taille de la section de code: 1

Code:
0
Machine demarree a l'adresse 0x0
!! PC=0x0 SP=0xbb8 PS=0x0 !! HALT
Machine arretee a l'adresse 0x1

Ce programme fait tout simplement HALT. Cette instruction est déjà implantée.

Ensuite, codez une partie de la fonction **Decode** pour supporter les versions les plus simples de PUSH et WRITE, pour pouvoir placer un entier sur la pile et ensuite l'afficher.
Un test de cette implantation donnerait ceci :
Custom Pile Dumper v0.4b
Taille de la section de code:5

Code:
40
0
7
21
0
Machine demarree a l'adresse 0x0
!! PC=0x0 SP=0xbb8 PS=0x0 !! PUSH 7
!! PC=0x3 SP=0xbba PS=0x0 !! WRITE %d
7
!! PC=0x4 SP=0xbb8 PS=0x0 !! HALT
Machine arrêtee a l'adresse 0x5

Les valeurs de PC, SP et PS sont celles contenues dans ces champs juste avant d'exécuter l'instruction décrite. Vous pouvez facilement savoir si vous décodez correctement une instruction en passant le même code à l'exécutable du répertoire public.

Ensuite, implantez une instruction arithmétique et vous pourrez construire un petit test à l'aide de l'exécutable du devoir 5.

Finalement, une fois les instructions arihtmétiques implantées en virgule flottante également, vous pourrez utiliser les tests du répertoire public. Ils sont en ordre croissant de complexité. Pour savoir ce que font ces tests, utilisez l'exécutable du répertoire public.

Afin de compiler votre devoir, faites comme suit :

make

Il vous est demandé de faire une soumission électronique du devoir. Pour cela, mettez vos fichiers dans un répertoire de nom *tp6.* Puis, faire exactement la commande suivante:

```
turnin  -cift209  -ptp6  tp6
```

(Cette commande doit être faite depuis le répertoire parent du répertoire "tp6")
Votre répertoire doit contenir le fichier suivant :

-tp6.as
Pour vérifier partiellement votre soumission, faire exactement la commande suivante:

```
turnin  -v  -cift209  -ptp6
```

(Cette commande doit, au moins, vous lister le répertoire et les fichiers soumis)

Prenez grand soin de respecter toutes ces directives, car toute soumission erronée entraînera probablement la note zéro. Tant que la date limite n'est pas atteinte, vous pouvez soumettre autant de fois que vous le désirez : seule la dernière soumission est conservée.

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.

La machine stocke les données en mémoire et dans les instructions en format Big-Endian.

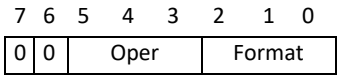
Les instructions ont quatre formats de base, selon le mode d'adressage utilisé. Les bits 6 et 7 du premier octet spécifient ce format.

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 (entiers) ou 32 bits (floats);
10 : mode d'adressage direct : l'opérande est une adresse de 16 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)

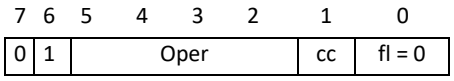
000 : « %c »
001 : « %d »

010 : « %s »
011 : « %f »
100 : « %u »
101 : « %x »

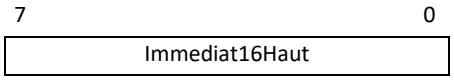
L'instruction HALT arrête l'exécution. Les instructions READ et WRITE fonctionnent comme printf et scanf, les formats étant identiques. Les valeurs à imprimer se trouvent sur le dessus de la pile.

Format 01 :

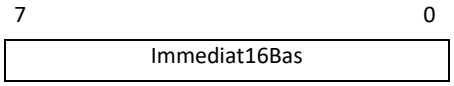
Octet 0



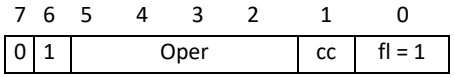
Octet 1



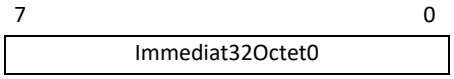
Octet 2



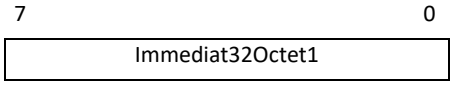
Octet 0



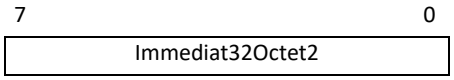
Octet 1



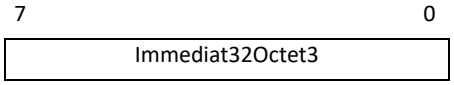
Octet 2



Octet 3

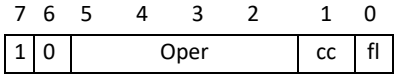


Octet 4

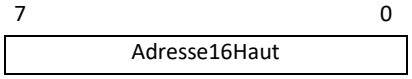


Format 10 :

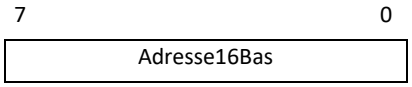
Octet 0



Octet 1



Octet 2



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

0000 : PUSH (modes 01 et 10)
0001 : POP (mode 10 seulement)
0010 : ADD (mode 01 seulement)
0011 : SUB (mode 01 seulement)
0100 : MUL (mode 01 seulement)
0101 : DIV (mode 01 seulement)
1101 : RET (mode 01 seulement)
1110 : JMPL (mode 01 seulement)
1111 : JMP (mode 01 seulement)

Seule les instruction PUSH et JMP prennent 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 16 bits dans le format 10.
Les autres instructions auront donc uniquement l'octet 0 d'utilisé et ***leur longueur sera donc de 1 octet.***.

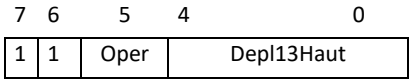
Les champs immédiat16, immédiat32 et adresse16 sont divisés en plaçant d'abord les bits les plus significatifs (8 à 15 ou 24 à 31 selon le cas) dans l'octet 1, jusqu'aux bits les moins significatifs (0 à 7) dans l'octet 2 ou 4, respectivement.

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

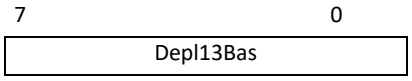
Le champ ***fl*** spécifie si l'on travaille avec des nombres en virgule flottante ou avec des nombres entiers (0 = entier, 1 = virgule flottante).

Format 11 :

Octet 0



Octet 1



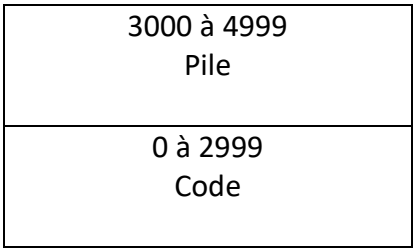
Ce format est utilisé pour les instructions de branchement.
La valeur du bit 5 indique la condition :

0 : BZ (branchement si zéro: bit 0 de PS)
1 : BN (branchement si négatif : bit 1 de PS)

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.

La mémoire de la machine virtuelle comporte 10000 octets.
Ils sont réservés comme suit :

5000 à 9999 Données et variables locales



Ce schéma implique que le pointeur de pile part à l'adresse 3000 et que le compteur ordinal part à l'adresse 0. Les adresses utilisées avec les instructions PUSH et POP en mode direct devraient être supérieures à 5000.