TD 2 - Jeux d'instructions

Hugo Bolloré[1] and Mohammed Salah Ibnamar[2]

[1]hugo.bollore@uvsq.fr
[2]mohammed-salah.ibnamar@uvsq.fr

1 Syntaxe et convention

Chaque Instruction Set Architecture (ISA) possède sa propre syntaxe, dans le cadre de cette matière nous utiliserons la syntaxe suivante qui est une version partielle de celle du jeu d'instructions x86-64 (Intel/AMD).

1.1 Agencement

Chaque instruction peut être vue comme une fonction qui effectue un ensemble d'opérations sur une ou plusieurs valeurs données et qui retourne un ou plusieurs résultats. Une instruction est composée des éléments suivants :

1. Un nom
2. (Optionnel) un ensemble d'opérandes (l'équivalent d'un paramètre de fonction)

Les opérandes ont un rôle qui est défini pour chaque instruction (i.e le premier opérande contiendra le résultat de l'opération, les opérandes suivants doivent contenir les données d'entrées). Une même instruction peut avoir plusieurs "variantes", c.a.d un même nom mais avec un ensemble d'opérandes différent.

Listing 1: Exemple

```
MUL %Rx, %Rx, %Rx
MUL %Rx, %Rx, (%Rx, %Rx)
```

1.2 Symboles

Pour identifier visuellement les différents opérandes et leurs types, la syntaxe de l'ISA x86-64 défini les éléments suivants :

1. *,* est le séparateur des opérandes,
2. *%* indique que l'opérande est un registre,
3. *Rx* est le nom d'un registre, à remplacer dans l'assembleur,
4. *(* et *)* sont les délimiteurs des opérandes mémoires,
5. *#* est le délimiteur pour les opérandes "immédiats", c.à.d qu'ils encodent directement une valeur numérique.

1.3 Registres

Un registre est une petite case mémoire qui permet de stocker les données sur lesquelles le processeur va faire ses opérations. Chaque ISA spécifie un ensemble de registres disponibles et spécifie leurs nommages. Il existe plusieurs types de registres, certains ont des rôles spéciaux et ne peuvent pas être accédés directement par une instruction.

Dans le cas de l'assembleur x86-64, il est défini ce qui suit :

1. 16 registres d'étâts généraux de 64 bits. On peut accèder à ceux-ci avec différents nommages en fonction de la taille des données stockées :
   (a) 64 bits : RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, ..., R15
   (b) 32 bits : EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, ..., R15D
   (c) 16 bits : AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, ..., R15W
   (d) 8 bits : AL, BL, CL, DL, SIL, DIL, BPL, SPL, R8B, R9B, ..., R15B

   Attention les noms des registres en version 32/16/8 bits sont des alias et référencent la partie basse du registre 64 bits.

2. 16 registres vectoriels de 512, 256 ou 128 bits. Ceux-ci sont optionnels et ne sont pas disponibles sur tous les processeurs. Les différents nommages sont les suivants :
   (a) 512 bits : ZMM0, ZMM1, ZMM2, ..., ZMM15
   (b) 256 bits : YMM0, YMM1, YMM2, ..., YMM15
   (c) 128 bits : XMM0, XMM1, XMM2, ..., XMM15

   Attention, comme précédement les noms des registres de plus petite taille sont des alias qui référencent la partie basse des registres plus grands.

3. Un registre de flags qui stocke des informations utilisées pour contrôler le déroulement d'un programme. Celui-ci n'est pas accessible directement mais est mis à jour implicitement par des instructions comme *CMP*.

1.4 Opérandes mémoires

Le processeur peut également lire et/ou écrire des données en mémoire. Pour cela il a besoin de l'adresse mémoire de la donnée concernée. Pour simplifier le déplacement en mémoire (par exemple lorsque l'on parcourt les éléments d'un tableau) il existe une syntaxe spécifique pour les opérandes mémoires : $(base, index, scale)$. Ici *base* et *index* sont des registres, *scale* est un nombre (1/2/4 ou 8). L'adresse est obtenue par le calcul $base + index * scale$.

Listing 2: Exemple

```
MUL %RAX, %RBX, (%RCX, %RDX, 2)
;// RAX = RBX * Valeur à l'adresse (RCX + RDX * 2)
```

Exercice 2.1 (Comparaison d'architectures)

En respectant la priorité des opérateurs arithmétiques, générez efficacement le code évaluant l'expression $A = B + C * D - E$ pour les architectures suivantes.

Question 2.1.1 (Machine à pile)

Jeu d'instruction :

Instruction	Description
PUSH *%Rx*	Pousse la valeur contenue dans le registre Rx sur la pile
POP *%Rx*	Recupère la valeur au sommet de la pile et la met dans le registre Rx
MUL	Multiple les deux valeurs en haut de la pile (en les enlevant de la pile) et écrit le résultat sur la pile
ADD	Additionne les deux valeurs en haut de la pile (en les enlevant de la pile) et écrit le résultat sur la pile
SUB	Soustrait la seconde valeur sur la pile avec la première (en les enlevant de la pile) et écrit le résultat sur la pile

Les variables B, C, D et E seront fournies dans les registres RBX, RCX, RDX et RSI. Le résultat, A, devra être retourné dans le registre RAX.

Question 2.1.2 (Machine à accumulateur)

Jeu d'instruction :

Instruction	Description
LOAD *%Rx*	Charge la valeur contenue dans le registre Rx dans l'accumulateur
STORE *%Rx*	Stocke la valeur de l'accumulateur dans le registre Rx
MUL *%Rx*	Multiple la valeur contenue dans le registre Rx avec la valeur de l'accumulateur et écrit le résultat dans l'accumulateur
ADD *%Rx*	Additionne la valeur contenue dans le registre Rx avec la valeur de l'accumulateur et écrit le résultat dans l'accumulateur
SUB *%Rx*	Soustrait la valeur contenue dans l'accumulateur avec la valeur contenue dans le registre Rx et écrit le résultat dans l'accumulateur

Les variables B, C, D et E seront fournies dans les registres RBX, RCX, RDX et RSI. Le résultat, A, devra être retourné dans le registre RAX.

Question 2.1.3 (Machine à registres d'usages généraux)
Jeu d'instruction :

Instruction	Description
LOAD *%Rx, (%Ry, %Rz)*	Charge la valeur pointé par l'opérande mémoire dans le registre Rx
STORE *(%Ry, %Rz), %Rx*	Stocke la valeur du registre Rx à l'adresse pointée par l'opérande mémoire
MUL *%Rx, %Ry*	Multiple la valeur du registre Rx avec la valeur du registre Ry et écrit le résultat dans le registre Rx
ADD *%Rx, %Ry*	Additionne la valeur du registre Rx avec la valeur du registre Ry et écrit le résultat dans le registre Rx
SUB *%Rx, %Ry*	Soustrait la valeur du registre Rx avec la valeur du registre Ry et écrit le résultat dans le registre Rx

Les **adresses** des variables B, C, D et E seront fournies dans les registres RBX, RCX, RDX et RSI. Le résultat, A, devra être retourné à l'**adresse** fournie par le registre RAX.

Exercice 2.2 (Génération de code)

En utilisant le jeu d'instructions présenté à la suite, générez efficacement ces codes :

1. `DO I=1,100,1 A(I) = B(I)+1 ENDDO`
2. `DO I=1,100,1 A(I) = B(I)+C(I) ENDDO`
3. `DO I=1,100,1 A(I) = D*B(I)+E*C(I) ENDDO`

Les variables A, B et C sont des tableaux contenant des entiers (32 bits), leurs **adresses** seront fournies dans les registres RAX, RBX et RCX. Les variables D et E seront fournies dans les registres EDX et ESI.

Jeu d'instruction :

Instruction	Description
LOAD *%Rx, (%Ry, %Rz)*	Charge la valeur pointé par l'opérande mémoire dans le registre Rx
STORE *(%Ry, %Rz), %Rx*	Stocke la valeur du registre Rx à l'adresse pointée par l'opérande mémoire
MUL *%Rx, %Ry, %Rz*	Multiple la valeur du registre Ry avec la valeur du registre Rz et écrit le résultat dans le registre Rx
MULI *%Rx, %Ry, #N*	Multiple la valeur du registre Ry avec la valeur N et écrit le résultat dans le registre Rx
ADD *%Rx, %Ry, %Rz*	Additionne la valeur du registre Ry avec la valeur du registre Rz et écrit le résultat dans le registre Rx
ADDI *%Rx, %Ry, #N*	Additionne la valeur du registre Ry avec la valeur N et écrit le résultat dans le registre Rx
SUB *%Rx, %Ry, %Rz*	Soustrait la valeur du registre Ry avec la valeur du registre Rz et écrit le résultat dans le registre Rx
SUBI *%Rx, %Ry, #N*	Soustrait la valeur du registre Ry avec la valeur N et écrit le résultat dans le registre Rx
CMP *%Rx, %Ry*	Compare la valeur du registre Rx avec la valeur du registre Ry et met à jour le registre de flags
JLT *label*	Jump au label si le registre de flags a enregistré une valeur inférieure (Lesser Than)