TD2

Assembleur « Craps » : accès mémoire, pile, sous-programme

1- Structure d'un programme assembleur

On utilisera le langage assembleur défini pour le processeur « craps », dont on découvrira les instructions progressivement (plus riches que celles de « mini-craps »).
Un programme écrit en langage assembleur est constitué :

- d'instructions dans un format simple composé de deux champs séparés par un espace ou plus : ***opération*** (nom abrégé) ***opérandes*** (séparés par une virgule)
  add %r2, %r3, %r4 / set 123, %r5 / ld [%r5+%r0], %r6 / st %r6, [%r5+%r1] / beq adresse
- de données : variables, tableaux, etc. qui seront réservées en mémoire

Pour faciliter la programmation, on utilisera des noms symboliques (étiquettes, labels) pour désigner des constantes, et des adresses de variables ou d'instructions. Il est plus facile de se rappeler d'un nom qui porte un sens plutôt que d'une adresse qui peut, en plus changer.

L'exemple suivant montre le code assembleur correspondant à l'instruction :

Resultat := Variable1 + Variable2

Les variables Resultat, Variable 1 et Variable 2, de type entier, sont réservées en mémoire à des adresses que l'on va désigner par Resultat, Variable 1 et Variable2 (pour l'instant, on n'a pas besoin de connaître les valeurs exactes de ces adresses). Contrairement aux langages évolués, les noms des variables que nous utilisons désignent leur adresse ; et les valeurs de ces variables sont désignées par [variable]. Les […] indiquent un accès mémoire.

```
Debut :     set     Variable1, %r2      // r2 <- adresse de Variable1
            set     Variable2, %r3      // r3 <- adresse de Variable2
            set     Resultat, %r4       // r4 <- adresse de Resultat
            ld      [%r2], %r5          // ld = load : r5 <- valeur mémoire de Variable1
            ld      [%r3], %r6          // r6 <- valeur mémoire de Variable2
            add     %r5, %r6, %r7       // %r7 <- %r5 + %r6
            st      %r7, [%r4]          // st = store : valeur mémoire de Résultat <- r7
Fin:        ba      Fin                 // branchement sur place comme fin de programme
Variable1 : .word   123
Variable2 : .word   654
Resultat :  .word   0
```

Il est important de noter :

- pour faciliter la lisibilité du programme assembleur, on aligne les différents champs sur des colonnes séparées par une tabulation : ***label*** ***nom_instruction*** ***opérandes***
- les labels se terminent par : et nous évitent d'avoir à connaître les adresses manipulées
- le programme est un bloc d'instructions qui seront exécutées de façon séquentielle, de la première jusqu'à la dernière. Il est donc interdit d'insérer des données au milieu de ces instructions

- Le bloc de données peut être mis :
  - juste après les instructions, et sera déplacé automatiquement si des instructions sont ajoutées ou supprimées
  - à un endroit fixe dont on peut indiquer l'adresse en utilisant la directive « .org », par exemple .org 100 avant la déclaration de Variable1, mais cela présente le risque de voir les données et les instructions se chevaucher si on ajoute des instructions.
- Les directives .org, .word, etc. sont interprétées lors de la traduction du programme assembleur en langage binaire : « .word » permet de réserver et d'initialiser un mot mémoire, « .org » permet de placer le curseur d'adresse de façon absolue ou relative.

Les Instructions de branchement : b'condition' adresse

- condition : a (always), eq (égal), ne, gt (greater than), lt (less than), lu (less unsigned), leu, gu, geu, …
- la condition fait référence au résultat d'une opération précédente, dont l'état est stocké dans les indicateurs N, Z, V et C (seules les instructions se terminant par CC modifient les indicateurs).
- On utilise souvent l'instruction cmp opérande1, opérande2 suivie d'une instruction de branchement dont la condition porte sur la relation d'opérande1 par rapport à opérande2 (ou à la relation de (opérande1 – opérande2) par rapport à 0). Le tableau suivant fournit quelques exemples :

Expression	Condition	Instructions assembleur	Instructions craps	Indicateurs
A < 0	neg	Test A	subcc A, 0, %r0	N = 1
A = B	eq	cmp A, B	subcc A, B, %r0	Z = 1
A >= B (non signé)	geu	cmp A, B	subcc A, B, %r0	C = 0
A < B (signé)	lt	cmp A, B	subcc A, B, %r0	N*/V + /N*V

Les instructions de branchement nous permettront d'implanter les différentes structures de contrôle utilisées dans les langes évolués.

2- Si condition Alors Sinon

```
        Test condition                    ou          Test condition
        B'faux' Seq_sinon                             B'vrai' Seq_si
        Séquence si                                   Séquence sinon
        …                                             …
        Ba      Fin_si                                Ba      Fin_si
Seq_sinon : Séquence sinon          Seq_si :          Séquence si
        …                                             …
Fin_si :    …                       Fin_si :          …
```

Exemple : Le tableau suivant présente une séquence « Si Alors Sinon », et son implantation :

Algorithme	Choix	Séquence assembleur
Si A > B Alors A <- A – B Sinon B <- B – A FinSi	A supposé dans r1 B supposé dans r2	cmp %r1, %r2 bgu AsupB BsupA : sub %r2, %r1, %r2 ba FinSi AsupB : sub %r1, %r2, %r1 FinSi : …

3- Tant que

La boucle « Tantque » se caractérise par le test de la condition fait à l'entrée de la boucle. Elle peut être implantée de la façon suivante :

```
Tantque :     Test de la condition d'entrée
              B'condition fausse'  Fintanque
              … actions           // condition vraie : on exécute les instructions de la boucle
              Ba Tantque          // et on recommence
Fintanque :   ….
```

Exemple : soit l'algorithme de calcul du pgcd de deux nombres A et B (non signés)

Algorithme	Choix	Séquence assembleur

Tantque A /= B Faire
  Si A > B Alors
  A <- A – B
  Sinon
  B <- B – A
  FinSi
Fintantque
	A supposé dans r1
	B supposé dans r2
		Tantque : cmp %r1, %r2
		beq Fintanque
		bgu AsupB // r1 > r2
		BsupA : sub %r2, %r1, %r2
		ba FinSi // ou ba Tantque
		AsupB : sub %r1, %r2, %r1
		FinSi : ba Tantque
		Fintanque : …

4- Répéter

La boucle « répéter » se caractérise par le test de la condition qui doit être fait à la sortie de la boucle. Elle peut être implantée de la façon suivante :

```
Repeter :     … actions
              Test de la condition de sortie
              B'condition fausse'  Repeter
              …
```

Par exemple, la séquence suivante permet de réaliser une boucle « répéter » de 10 passages

```
              set     10, %r1       // index : nombre de passages restant
Repeter:      … actions
              subcc   %r1, 1, %r1   // positionne les indicateurs N, Z, V, C
              bne     boucle        // branchement à boucle si %r1 /= 0
              …                     // instruction exécutée si %r1 = 0
```

Exercice1 : Ecrire le programme qui calcule la factorielle d'un nombre stocké en mémoire.

L'instruction de multiplication possède les formes suivantes :

- umulcc %rs1, %rs2, %rdest // rdest(32 bits) <- rs1(16 bits) x rs2(16 bits)
- umulcc %rs1, constante13, %rdest // rdest(32 bits) <- rs1(16 bits) x constante13(->16 bits)

```
Factorielle :   ….
                ….
Fin :           ba Fin
Nombre :        .word 20
```

5- Sous-programmes

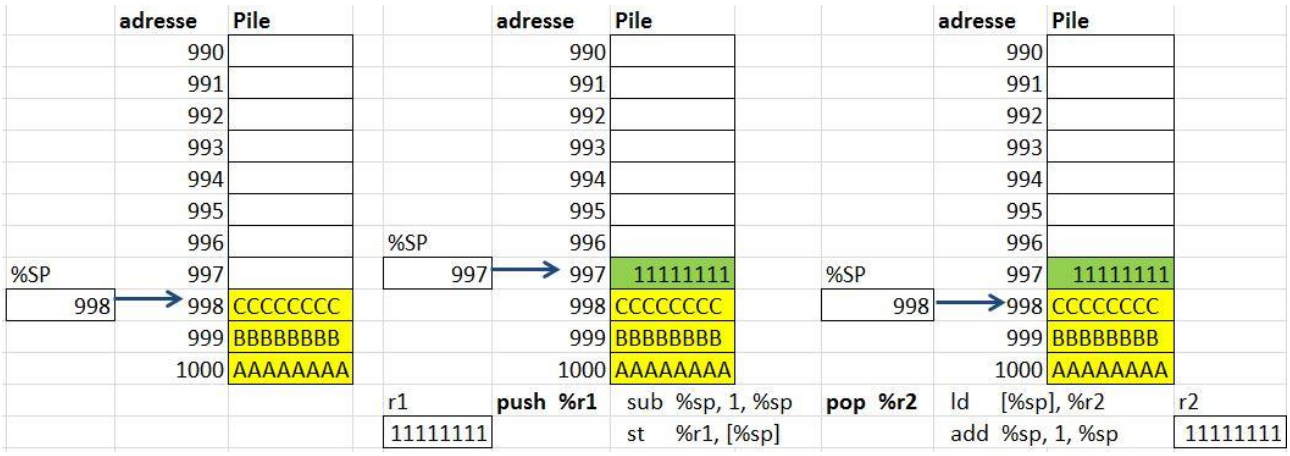
L'appel d'un sous-programme effectue les différentes opérations suivantes :

- préparation des paramètres, on parlera de passage de paramètres
- mémorisation de l'adresse de retour (adresse de l'instruction qui suit l'appel),
- branchement à la première instruction du sous-programme,
- exécution du sous-programme, puis retour au programme appelant

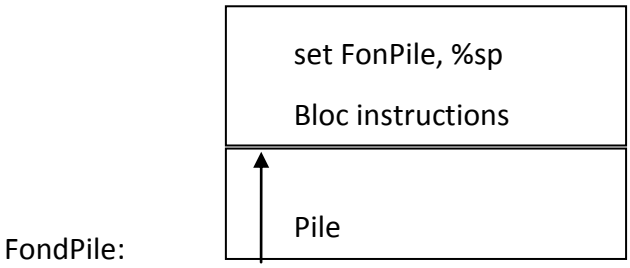
Le passage des paramètres et la mémorisation de l'adresse de retour peut s'effectuer de deux manières :

- dans des registres,
- en utilisant une pile : une pile est une structure de données, sur laquelle on peut effectuer deux opérations :
  - empiler : ajout d'une nouvelle valeur en sommet de pile : **push %ri**
  - dépiler : récupération de la valeur au sommet de pile : **pop %ri**

La pile de craps est gérée par un pointeur, qui est un registre dédié appelé %sp (%r29), et qui pointe sur l'élément au sommet de la pile. Ci-dessous, une pile implantée à l'adresse 1000, et son évolution après un push et un pop :



Chaque programme doit initialiser sa propre pile avec une adresse assez éloignée du bloc instructions pour éviter tout écrasement lorsque la pile se remplit.



En outre, la pile est utilisée pour :

- sauvegarder les registres qui sont modifiés par le sous-programme afin que ces modifications n'affectent le contexte du programme appelant
- réserver des variables locales au sous-programme : c'est ce qui est fait par les compilateurs et est appelé « mémoire automatique »

Pour des raisons d'efficacité (l'accès aux registres est plus rapide que l'accès mémoire), et de facilité de programmation, craps est doté d'un bon nombre de registres, dont plus de 20 à la disposition du programmeur. Nous les utiliserons donc en priorité pour le passage des paramètres et les variables locales.

Un sous-programme en assembleur craps prendra donc la forme :

```
// rôle :
// paramètre 1 : …… dans r1
// paramètre 1 : …… dans r2
// …
// résultat : ……. dans r3
Sous-prog :   push %r1
              push %r2      // si on modifie r1 et r2.
              …             // R3 non sauvegardé car sert pour retourner le résultat
              pop %r2       // attention : dépilement dans l'ordre inverse de l'empilement
              pop %r1
              ret           // retour au programme appelant
```

L'appel à un sous-programme s'effectue par l'instruction « ***call »***, qui sauvegarde l'adresse de l'instruction courante dans le registre **%r28**. Le retour de sous-programme s'effectue par l'instruction « ***ret*** » qui récupère l'adresse de retour dans le registre %r28. On verra plus loin que cette solution n'est pas suffisante pour des appels en cascade.

Exercice 2 : Ecrire le sous-programme qui calcule le pgcd de deux nombres fournis dans r1 et r2 ; et retourne le résultat dans r3.

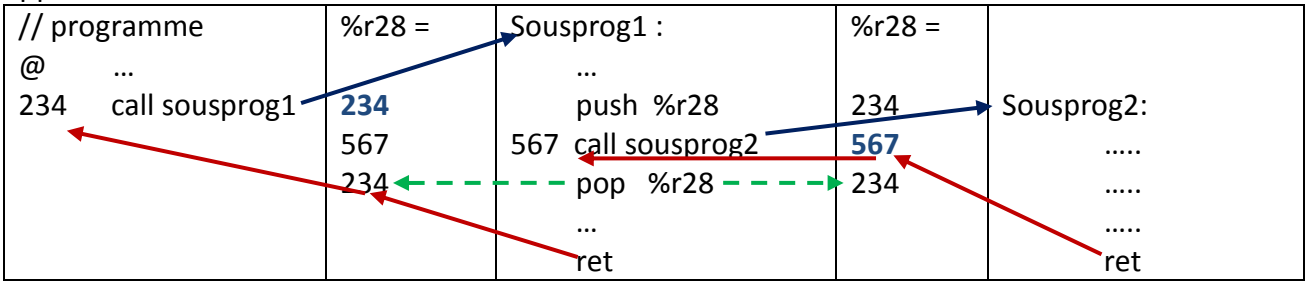
Exercice 3 : Ecrire le sous-programme qui calcule le max d'un tableau d'entiers non signés.

Un tableau est un ensemble d'éléments consécutifs en mémoire, dont on connait l'adresse (adresse du tableau = adresse du 1er élément), et le nombre d'éléments.

```
Tableau :     .word  123, 475,  255, 857, …
```

Appels en cascade

Lorsqu'un sous-programme a besoin d'appeler un autre sous-programme, le registre %r28 n'est pas disponible, car il contient déjà l'adresse de retour du premier appel. Pour ne pas perdre cette adresse, il faut la sauvegarder dans la pile avant le nouvel appel et la récupérer au retour de cet appel.



Exercice 4 : Ecrire le sous-programme qui calcule de façon récursive la factorielle d'un nombre
A- Le nombre est fourni dans r1 ; et le résultat est retourné dans r2
B- On utilisera la pile pour fournir le paramètre et retourner le résultat.