TP3 & TP4

Programmation en assembleur « Craps »

1- environnement de travail

On travaillera sur la plateforme « shdl.fr » / « craps sandbox ». On y disposera d'un éditeur pour écrire nos programmes en assembleur, et d'un simulateur pour les tester.
Dans le simulateur, le programme est chargé à partir de l'adresse 0. On peut y voir :

- Colonne de gauche : les adresses mémoire occupées par le programme et ses données
- 2ème colonne : les codes des instructions ou les valeurs des données
- 3ème colonne : les labels (étiquettes) définis dans le programme
- 4ème colonne : les instructions en assembleur
- 5ème colonne : les 32 registres disponibles dans craps

Toutes les valeurs sont affichées en hexadécimal, sauf pour les registres où l'utilisateur peut choisir le format décimal ou binaire en plus de l'hexadécimal.

En haut à gauche de la fenêtre, les boutons flèches permettent de lancer l'exécution en mode continu ou instruction par instruction. Le bouton carré permet d'arrêter l'exécution.
Attention : les registres sont réinitialisés à 0 lorsqu'on appuie sur le bouton carré. Il vaut mieux utiliser le bouton « pause » (II) pour conserver le contenu des registres.

L'instruction courante est surlignée en bleu.
Les registres suivants jouent un rôle particulier :
R31 : IR = registre instruction
R30 : PC = compteur ordinal : contient l'adresse de l'instruction courante
R29 : SP = pointeur de pile : contient l'adresse du sommet de pile
R28 : registre qui contient l'adresse de la dernière instruction « call »

2- premiers exemples

A- Soit le code suivant :

```
N = 10
                set     Tab, %r2
                ld      [%r2], %r1          //  r1 est modifiable contrairement à mini-craps
                set     1, %r3
Tantque :       cmp     %r3, N
                bgeu    FinTque             // branch if r3 greater or equal unsigned N
                ld      [%r2+%r3], %r4      // ld = load
                add     %r4, %r1, %r1
                add     %r, 1, %r3
                ba      Tantque
FinTque:        st      %r1, [%r2+N]        // st = store
Stop :          ba      Stop
Tab :           .word   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0
```

Enregistrer ce programme et le tester dans le simulateur en mode exécution instruction par instruction, en vérifiant à chaque fois le résultat de l'instruction exécutée. On remarquera que l'instruction set consomme 2 mots mémoire. Elle correspond en réalité à deux instructions machines. Où est stocké le résultat final ?

B- Soit le code suivant :

```
// programme principal
PILE = 0x200            // le fond de pile à l'adresse 0x200
N = 5
                set     PILE, %sp       //initialisation du pointeur de pile : ABSOLUMENT NECESSAIRE
                set     N, %r1
                call    factorielle     // factorille(N) – résultat dans r2
Stop :          ba      Stop

// sous-programme qui calcule la factorielle d'un entier naturel
// paramètres :         In : r1 contient le nombre dont veut calculer la factorielle
//                      Out : r2 contient le résultat
factorielle:    push    %r1             // r1 modifié dans le sous-programme : il doit donc être
                                        // sauvegardé dans la pile à l'entrée et restauré à la sortie
                set     1, %r2          // factorielle(0) ou factorielle(1)
tantque:        cmp     %r1, 1
                bleu    retour          // branchement  retour si r1 <= 1
                umulcc %r1, %r2, %r2
                dec     %r1
                ba      tantque
retour:         pop     %r1             // restaurer r1 qui doit retrouver sa valeur d'entrée
                ret
```

Enregistrer ce programme et le tester dans le simulateur en mode exécution instruction par instruction, en vérifiant à chaque fois le résultat de l'instruction exécutée. On remarquera que certaines instructions occupent deux mots mémoire (set, call, push, pop). Elles correspondent en réalité à deux instructions machines. Vérifier bien que l'instruction call enregistre son adresse dans le registre r28. Vérifier le contenu de la pile et son pointeur.

3- appels en cascade

L'instruction call enregistre son adresse dans le registre r28 avant d'effectuer un branchement au sous-programme. Mais un problème se pose lorsque le sous-programme effectue lui-même un appel à un autre sous-programme (ou à lui-même s'il est récursif) : l'ancienne adresse dans r28 est écrasée par la nouvelle. Pour éviter de perdre cette première adresse, un sous-programme doit sauvegarder le registre r28 dans la pile avant d'exécuter l'instruction « call », et de récupérer cette adresse après le retour du call : push %r28 … call xxxxxx … pop %r28

Prendre une copie du code **2B**, et transformer le sous-programme « factorielle » en un sous-programme récursif.
Tester le programme dans le simulateur, en analysant l'évolution de la pile durant la phase d'appel, puis durant la phase de retour.

4- tri d'un tableau

L'objectif est d'écrire le sous-programme qui effectue le tri croissant d'un tableau d'entiers relatifs (signés) selon l'algorithme suivant :

```
Pour I de N-1 à 1 Pas=-1 Faire
    Calculer Max et Indice_Max de Tab(0..I)
    Echanger Tab[I] et Tab[Indice_Max]
FinPour
```

A- Ecrire et tester le sous-programme qui calcule le max, et son indice, d'un tableau de M entiers relatifs (signés), en suivant l'algorithme suivant :

```
Max <- Tab[0]
Indice_max <- 0
Pour Index de 1 à M-1 Faire
        Si Max < Tab[Index] Alors
                Max <- Tab[Index]
                Indice_max <- Index
        FinSi
FinPour
```

Il vous appartient de définir les paramètres de ce sous-programme et de choisir comment ils doivent être passés.

B- Ecrire et tester le sous-programme qui effectue le tri croissant d'un tableau d'entiers relatifs.

5- Crible d'Eratosthène

Le **crible d'Eratosthène est** un algorithme qui permet de trouver tous les nombres premiers entiers inférieur à un entier naturel *N*.

Il procède par élimination, en supprimant d'un tableau, initialement rempli des entiers allant de 2 à N, tous les multiples de n'importe quel entier présent. En supprimant tous ces multiples, il ne restera que les entiers qui ne sont multiples d'aucun entier à part 1 et eux-mêmes, et qui sont donc premiers.

On commence par éliminer les multiples de 2, puis les multiples de 3, puis les multiples de 5 (4 et les multiples de 4 ont été supprimés car multiples de 2), et ainsi de suite.

On peut s'arrêter lorsque le carré de l'entier courant 'C' est supérieur à N ; car le premier multiple de 'C' disponible est égal à C*D avec D ne pouvant être inférieur à C puisque tous les multiples des nombres inférieurs à C ont été éliminés ; Le premier multiple disponible de C est donc >= C*C > N.

À la fin du processus, il ne reste dans le tableau que les nombres premiers inférieurs à N.

L'algorithme peut donc s'écrire sous la forme suivante :

```
Initialiser Tab_premiers[0..N] à 0, 0, 2, 3, …, N
        -- 0 et 1 n'étant pas premier, on initialise Tab_premiers[0] et Tab_premiers[1] à 0
        -- par la suite, tout nombre i éliminé car non premier se traduira par Tab_premiers[i] = 0
        -- à la fin, un nombre j est premier si Tab_premiers[j] = j (/=0)
i=2
Tant que i*i < N
    Si Tab_premiers [i] /= 0
        Eliminer les multiples de i
    Fin si
    i <- i + 1
Fin tant que
```

A- Ecrire et tester le sous-programme Initialiser_tab_premiers

B- Ecrire et tester le sous-programme Eliminer_multiples. Il faut bien réfléchir à un algorithme peu couteux, et éviter de tester tous les éléments du tableau. Une piste : quel est le premier multiple de i ? et le deuxième multiple de i ?
Une solution se trouve à la fin de ce sujet, mais faites un effort avant de la consulter.

C- Ecrire et tester le sous-programme Eratosthène

6- Evaluation d'une expression arithmétique en notation post-fixée

Il existe différentes manières d'écrire une expression arithmétique. La notation classique place l'opérateur entre les opérandes. Elle nécessite des parenthèses. Dans la notation post-fixée dite polonaise inversée, l'opérateur est placé après les deux opérandes et les parenthèses deviennent inutiles.

Par exemple, l'expression (15 + (7−4)) – 2 en notation classique devient :
15 7 4 − + 2 − en notation polonaise inversée.

L'évaluation d'une expression en polonaise inversée s'effectue en utilisant une pile. Au départ, la pile est vide. Les termes sont lus de gauche à droite :

- Lorsque le terme lu est un entier (opérande), il est empilé
- Lorsque le terme lu est un opérateur, on dépile les deux dernières valeurs empilées, on leur applique l'opérateur et on empile le résultat.
- A la fin, il doit y avoir une seule valeur dans la pile, qui correspond à la valeur de l'expression.

On codera l'expression de la façon suivante : chaque terme de l'expression est codé sur un mot mémoire de 32 bits, selon le format suivant : bit N°31 = Type – bits 30 à 0 = Valeur

- Lorsque Type= 0, le terme est un opérande, avec Valeur est un entier signé sur 31 bits
- Lorsque Type= 1, le terme est un opérateur, avec :
    - Valeur= 1 représente l'opérateur +
    - Valeur= 2 représente l'opérateur –
    - Valeur= 0 représente la fin de l'expression

Ainsi, l'expression présentée plus haut sera représentée en mémoire par : 0x0000000F, 0x00000007, 0x00000004, 0x80000002, 0x80000001, 0x00000002, 0x80000002, 0x80000000

Pour implanter ce programme, on aura recours aux instructions logiques and, or, andcc, orcc, et on commencera par écrire et tester le sous-programme qui retourne le type et la valeur d'un terme de l'expression.

Tester le programme avec l'expression : 15 + (7 – 4) – (3 – (-7 – 8))

Algorithme permettant d'éliminer les multiples de i

-- Le premier multiple de i est 2*i
-- Le deuxième multiple de i est 3*i
-- … il suffit de faire une boucle avec un pas = i
-- *on commence à i*i car tous les multiples de i inférieurs à i*i ont déjà été déjà éliminés*

Pour j de i*i à N, ***pas=i***
    Tab_premiers[j] <- 0
Fin pour