TP3 & TP4

Programmation en assembleur « Craps »

- ***Tous module validé doit être bloqué avant la fin de la séance***
- ***Tout module non bloqué sera considéré fait en dehors des séances encadrées***
- ***Un ou 2 modules peuvent être complétés après la séance pour un nombre de points de 5 au maximum***
- ***Les exercices doivent être réalisés et validés dans l'ordre du sujet***
- ***Tout exercice doit être testé correctement. Un commentaire en début de module doit indiquer si le code fonctionne comme demandé, ou si des erreurs persistent***
- ***Un exercice non testé ou ayant une mauvaise validation ne sera pas noté***

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. Une fois un programme valable est écrit dans l'éditeur, son fonctionnement peut être testé dans le simulateur.

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, les « boutons flèches » permettent de lancer l'exécution en mode continu (triangle) ou instruction par instruction (flèche arquée). 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          // N constante, pour éviter d'utiliser des valeurs en dur dans le programme
                set     Tab1, %r1
                set     Tab2, %r2
                clr     %r3
Tantque:        cmp     %r3, N
                bgeu    Stop            // branch if r3 greater or equal unsigned to N
                ld      [%r1+%r3], %r4      // ld = load - %r2+%r3]= adresse  de tab(r3)
                st      %r4, [%r2+%r3]
                add     %r3, 1, %r3         // index <- index + 1
                ba      Tantque
Stop :          ba      Stop
Tab1 :          .word   10, 9, 8, 7, 1, 6, 5, 4, 3, 2, 1
Tab2 :          .word   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

1- Enregistrer ce programme dans un module nommé « Tableau » et le tester dans le simulateur en mode exécution instruction par instruction (2 ou 3 passages dans la boucle), en vérifiant à chaque fois le résultat de l'instruction exécutée. On remarquera que l'instruction « set » consomme 2 mots mémoire. C'est une instruction assembleur qui est traduite par deux instructions machines.

2- Mettre un point d'arrêt (breakpoint) sur l'instruction st %r4, [%r2+%r3] (click gauche dans la colonne en gris clair à gauche de l'adresse de l'instruction).
Lancer l'exécution et verifier le résultat à chaque passage dans la boucle (l'exécution s'arrête à chaque passage sur le point d'arrêt) : En commentaire, en début de code, indiquer la valeur du register r4 à chaque point d'arrêt, et dire ce que fait le prgramme.

3- Transformer ce porgramme de manière à inverser Tab1 dans Tab2 : le premier élément de Tab1 copié dans le dernier élément de Tab2, …
Tester, et indiquer en commentaire si le code fonctionne comme demandé.

B- Dans un module nommé « min_tableau, **écrire et tester le programme qui calcule le min d'un tableau de N entiers relatifs et son indice.**

C- Soit le code suivant :

```
PILE = 0x200        // fond de pile à l'adresse 0x200
                set     PILE, %sp           // initialisation du pointeur de pile
                set     Chaine, %r1
                clr     %r2                 // %r2 <- 0 : nombre d'éléments
Repeter:        ld      [%r1], %r3
                cmp     %r3, %r0            // r3 ? 0
                beq     Stop
                push    %r3                 // %r3 -> sommet de pile
                inc     %r2                 // add %r2, 1, %r2
                inc     %r1                 // adresse du prochain élément
                ba      Repeter
Stop :          ba      Stop
Chaine :        .word   0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0     // 0 = fin d chaine
```

Dans un module nommé « inverser », enregistrer ce programme et le tester dans le simulateur en vérifiant l'état de la pile après chaque exécution de l'instruction push %r3.
On remarquera que l'instruction « push » consomme 2 mots mémoire. C'est une instruction assembleur qui est traduite par deux instructions machines.

Indiquer en commentaire les valeurs en sommet de pile après 4 exécutions de l'instruction push (en partant du sommet). Que fait ce programme ? Répondre en commentaire en début de module.

ttransformer ce programme de manière à ce qu'il inverse les éléments de la chaine (1er à la place du dernier, …, 0 non compris).

D- Soit le code suivant :

```
// programme principal
PILE = 0x200            // 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

// calcule la factorielle d'un entier naturel
// 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  à Retour si r1 <= 1
                umulcc %r1, %r2, %r2            // r2 <- r1 * r2
                dec     %r1
                ba      Tantque
Retour:         pop     %r1             // restaurer r1 qui doit retrouver  sa valeur d'entrée
                ret
```

Dans un module nommé « factorielle », enregistrer ce programme et le tester dans le simulateur en mettant un point d'arrêt sur la première instruction de Factorielle, et sur la dernière instruction de Factorielle. Quel est le contenu du registre r28 , à quoi correspond-il ? Indiquer la valeur du pointeur de pile à l'entrée et à la sortie du sous-programme.
Quelle est la valeur maximale de N possible (en hexadécimal) ? Expliquer.

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 qui

appelle un autre 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

Dans le module « factorielle », ajouter le sous-programme « Factorielle_recursive », et l'appeler depuis le programme principal.

Tester le programme dans le simulateur, en mettant des points d'arrêt de manière à vérifier le contenu de la pile durant la phase d'appel, puis durant la phase de retour.
Indiquer le contenu de la pile lors de la 1ère, 2ème et 3 entrée dans le sous-programme, et lors du 1er, 2ème et 3ème retour.

Remarques importantes :

- Dans un module, le nom d'une étiquette (label) ne peut être utilisé qu'une fois. Avec plusieurs sous-programmes, on risque d'avoir besoin d'étiquettes qui portent le même nom (boucle, fin, suite, retour, etc.). Un moyen simple sera de préfixer le nom de l'étiquette par une ou deux initiales du nom du sous-programme : par exemple FR_boucle (pour boucle de factorielle récursive).
- Il faut absolument indiquer le rôle de chaque sous-programme, et ses paramètres. Autrement tout deviendra rapidement incompréhensible.
- Il ne faut pas hésiter à mettre des commentaires dans le code : rôle de chaque registre, commentaire d'un bloc de code, etc.

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 (par sélection) :

Pour I de 0 à N-2 Faire

    Calculer Min et Indice_Min de Tab(I..N-1)

    Echanger Tab[I] et Tab[Indice_Min]

FinPour

A- Dans un module nommé « tri_selection », écrire et tester le sous-programme cal_min, d'un tableau de M entiers relatifs (signés), code déjà fait en 2B.

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

En mettant un point d'arrêt sur la dernière instruction du programme, relever le nombre de cycles consommés (en haut à droite), pour un tableau de 10 éléments, et un tableau de 20 éléments.

C- Transformer le programme précédent en sous-programme et écrire le programme qui permet de le tester

Une comparaison intéressante

Nous avons implanté le tri par sélection d'un tableau de deux façons, câblé au premier semestre, en assembleur au second. Il serait intéressant de comparer les performances de ces deux implantations, entre elles et avec des implantations en langages évolués tels que ada ou c.

La comparaison se fera sur le nombre de cycles d'horloges consommés pour le tri d'un même tableau initial de 10 entiers, par exemple : 10, 9, 8, …, 2, 1.

Pour le tri câblé :

- le calcul du max pour n éléments se fait en 2 + n cycles (voir graphe)
- le tri d'un tableau de N éléments (graphe de 5 cycles) nécessite un cycle de lancement, N+2+N+1+N+…+4 pour les calculs des max, 3*(N-1) cycles pour les échanges, soit un total de : N*(N-1)/2 + 6*(N-1), soit **90 cycles** pour le tableau de 10 éléments.

Pour le tri en assembleur, il suffit de mettre un point d'arrêt sur la dernière instruction du programme, de lancer l'exécution, et de noter le nombre de cycles affiché en haut à droite à l'arrêt du programme.

Si vous êtes intéressés, pour les versions en langages évolués (c et ada), on peut utiliser la commande time pour mesurer le temps cpu. Mais, comme le tri d'un si petit tableau va consommer moins d'une microseconde, on le répètera 1000000 de fois pour avoir une mesure exploitable :

```
Pour I de 1  à 1000000 Faire
        Initialiser le tableau
        Trier le tableau
Fin Pour
```

- la commande time ./tri_selection affichera différents temps dont on notera le temps « user ».
- pour ne garder que le temps de tri, on exécutera le même programme sans le tri, et on soustraira son temps d'exécution au temps précédent : tu_tri = tu_total – tu_initialisation
- Et pour calculer le nombre de cycles, on multipliera tu_tri par la fréquence du processeur (3,4 GHz pour certaines machines de l'N7), et on divisera par 1000000 (nombre de tris).

Pour la version ada, on devrait avoir un nombre de cycle 20 à 25 fois plus élevé que la version câblée.

Pour la version C, le nombre de cycles est similaire à celui d'ada, mais il peut être réduit de presque 50% si on compile avec l'option d'optimisation –O3 (gcc –O3 …).

5- Opérations binaires

Craps dispose d'un certains nombres d'opérations binaires :

- Les opérations logiques : or, and, xor
- 2 opérations de décalage :

- sll %rsrc, val, %rdest (shift logical left) : décalage binaire à gauche du contenu du registre rsrc, de val position (<32), avec résultat dans rdest. Val peut être une constante numérique ou dans un registre. Les « Val » bits à droite sont mis 0.
- slr %rsrc, val, %rdest (shift logical right) : décalage binaire à droite du contenu du registre rsrc, de val position (<32), avec résultat dans rdest. Val peut être une constante numérique ou dans un registre. Les « val » bits à gauche sont mis 0.

Soit le sous-programme suivant :

```
// Vérifier si un nombre est palindrome binaire
// Méthode : calcul du nombre inverse (faible poids <-> fort poids) et comparaison avec le nombre initial
// IN : r1, nombre
// OUT : r2 = 1 si palindrome, 0 sinon
palindrome_bin_inv:
                push  %r3              // sauvegarde des registres modifiés
                push  %r4
                push  %r5
                push  %r6
                set   1, %r2           // résultat <- 1
                set   32, %r3          // nombre de bits
                mov   %r1, %r4         // copie du nombre
                clr   %r5              // nombre inverse
pbi_bcle:
                andcc %r2, %r4, %r6    // isoler bit0
                sll   %r5, 1, %r5      // r5 *2
                or    %r5, %r6, %r5    // + bit0
                slr   %r4, 1, %r4      // décaler d'une position -< droite
                deccc %r3
                bne   pbi_bcle
                cmp   %r1, %r5         // nombre intial ?= nombre inverse
                beq   pbi_ret          // palindrome : r2 est déjà =1
                clr   %r2              // non palindrome
pbi_ret:        pop   %r6              // restauration des registres
                pop   %r5
                pop   %r4
                pop   %r3
                ret
```

A- Dans un module nommé « palindrome_binaire », tester ce sous-programme avec différentes valeurs, et noter le nombre de cycles consommés.

B- Ecrire une seconde version de ce sous-programme fonctionnant avec comparaison des bits symétriques (bit0 avec bit31, bit1 avec bit30, etc.).
Tester et comparer les performances avec le sous-programme précédent.

6- 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 déjà 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 tous les multiples de i
    Fin si
    i <- i + 1
Fin tant que
```

A- Dans un module nommé « eratosthene », écrire et tester le sous-programme Initialiser_tab_premiers. On peut prendre N=110

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, car cela peut devenir très, très long. Une piste : quel est le premier multiple de i ? et le deuxième multiple de i ? …

C- Ecrire et tester le sous-programme « eratosthene ». On doit trouver le résultat (en hexa) :
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109
(2, 3, 5, 7, b, d, 11, 13,17, 1d, 1f, 25, 29, 2b, 2f, 35, 3b,3d, 43, 47, 49, 4f, 53,59, 61, 65, 67, 6b, 6d)