Université d'Angers	Département Informatique
Faculté des Sciences	L3 Informatique 2011-2012
Module : *Architecture des Ordinateurs*	Jean-Michel Richer

Contrôle Continu
Architecture des Ordinateurs
Correction

Remarque générale : si, pour certains, vous passiez un peu plus de temps à réfléchir et travailler vous auriez de meilleurs résultats. Certains viennent en cours ou en TD et parlent pendant 2 heures de tout et surtout de n'importe quoi, mais pas de l'exercice à traiter, ou disons que c'est annexe. Vous n'êtes pas concentrés sur ce que vous devez faire. Idem en TP :

- on on a appris lors du TP n° 2, comment traiter `argv[1]`
- puis on a revu cela en TD (exercice 20 je crois)
- TP n° 8, on vous demande de récupérer cette valeur, et deux étudiants sur dix-sept sont capables de le faire

Premièrement, vous n'avez rien appris de ce point de vue, et deuxièmement, vous n'avez même pas l'idée de vous dire, j'ai déjà fait cela, je vais recopier ce que j'avais fait précédemment.

C'est donc que pour la plupart, vous ne devez pas souvent travailler vos cours, TD, TP. Cela transparait également en TD quand vous me posez des questions du style : "pouquoi vous faites cela ?". La réponse est : on a vu en cours pourquoi on faisait ainsi.

Vous êtes à bac+3 et bous n'avez toujours pas intégré, que le cours magistral donne les éléments essentiels qui seront mis en application en TD, puis mis en pratique en TP. Donc, par transitivité, si vous arrivez en TD sans connaître votre cours, vous ne pourrez pas faire grand chose et cela se répercutrera sur le TP. La solution pour réussir est donc simple : en cours, écouter, essayer de comprendre, si certaines choses suscitent quelque interrogation, n'hésitez pas à poser des questions à l'enseignant, relisez et tentez d'apprendre votre cours avant de venir en TD.

Exercice 1 -(5 pts) - cf cours et exercice que j'ai donné et que vous avez TOUS su faire ! Là ce n'est pas la même chose ! Etrange ? Non ! Certains ont simplement recopié ce qu'avaient fait les autres. Ils n'ont pas tenté de résoudre le problème, ils ont fait un simple copier-coller.

On rappelle qu'il fallait utiliser les registres SSE et faire en sorte de mettre quatre valeurs consécutives de la suite dans un registre.

Deux solutions :

```
initialization:
   xmm1 = [f(n+3) f(n+2) f(n+1) f(n)]
recursivity:
    pshufd  xmm2, xmm1, 10010001b ; xmm2 = [f(n+2) f(n+1) f(n)   f(n+1)]
    pand    xmm1, xmm3            ; xmm1 = [f(n+3) f(n+2) f(n+1) 0]
    paddd   xmm1, xmm2            ; xmm1 = [f(n+4) f(n+3) f(n+2) f(n+1)]

other solution

initialization:
    xmm0 = [f(n+3) f(n+2) f(n+1) f(n)]
    xmm1 = [f(n+4) f(n+3) f(n+2) f(n+1)]
 recursivity:
    paddd   xmm0, xmm1 ; xmm0 = [f(n+5) f(n+4) f(n+3) f(n+2)]
    paddd   xmm1, xmm0  ; xmm1 = [f(n+6) f(n+5) f(n+4) f(n+3)]
```

Exercice 2 -(5 pts) -

1. il fallait regarder l'évolution du codage des nombres et identifier les changements qui interviennent sur la mantisse et/ou l'exposant. On s'aperçoit que lorsqu'on divise par 2, la mantisse ne change pas mais que l'exposant est décrémenté de 1 !
2. soit on réalise le traitement suivant : isoler l'exposant, le décrémenter puis le remettre en place ou alors on fait une simple soustraction de 0x800000 (valeur hexa), c'est à dire que l'on soustrait 1 à l'exposant qui débute en bit 23.

   ```
   mov ebx, eax    ; on manipule l'exposant dans EBX
   shr ebx, 23     ; exposant dans bits de poids faible
   and ebx, 0xFF
   dec ebx         ; soustraire 1
   shl ebx, 23     ; remettre en place dans EBX
   and eax, 0x807FFFFF ; puis dans EAX garde signe + mantisse
   or eax, ebx        ; fixe l'exposant dans EAX
   ```

   ou simplement

   ```
   sub eax, 0x800000
   ```

   pour ceux qui m'ont dit qu'il suffisait de faire un décalage à droite de 1 bit, cela ne fonctionne qu'avec les entiers naturels. Si vous prenez $72.0 = 42.90.00.00$ et que vous réalisez un décalage à droite, vous obtenez $21.48.00.00 = 6.7762636e - 19$
3. il suffit de faire un AND avec un masque qui supprime le bit de poids fort (qui indique le signe). Certains ont la solution suivante qui fonctionne mais n'est pas la plus naturelle :

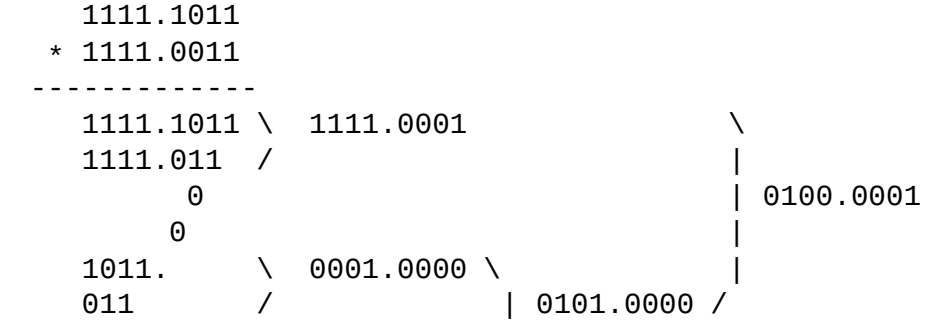
   ```
   shl eax,1
   shr eax,1
   ```

   ou plus simplement

   ```
   and eax, 0x7FFFFFFF
   ```

Exercice 3 -(4 pts) - *cf TD*, certains ont codé sur 4 bits ! ! ! alors que le sujet indique clairement 8 bits ! D'autres ont utilisé la notation en complément à 1, alors qu'il est indiqué en complément à 2. Ensuite, pour montrer que cela fonctionne il faut faire les calculs intermédiaires liés aux sommes.

1. coder la valeur -13 : 1111.0011 (binaire signé en complément à 2)
2. coder la valeur -5 : 1111.1011
3. faire le produit, inverser faire -5×-13, cela fait une ligne en moins lors de la somme :



```
   11        \  0100.0000 /
   1         /
------------
  0100.0001
```

4. oui, le résultat est correct $-13 \times -5 = 65$, car on sait que la notation binaire signée en complément à 2 donne des résultats conformes aux lois de l'arithmétique et on reste dans l'intervalle de représentation sur 8 bits [-128..+127]

Exercice 4 -(3 pts) Coprocesseur -

– *cf cours et TP 5 et 8*

instruction	ST0	ST1	ST2	explication
fld1	1			empile 1
fld1	1	1		empile 1
fadd st1,st0	1	2		st1 += st0
fld1	1	1	2	empile 1
faddp st1, st0	2	2		st1 += st0, dépile
fmul st1,st0	2	4		st1 *= st0
fmulp st1,st0	8			st1 *= st0, dépile

– *cf TP*

```
section .data
    msg: db '%f', 0

section .text
    sub     esp,8
    fstp    [esp]
    push    dword msg
    call    printf
    add     esp,12
```

Exercice 5 -(3 pts) Instructions - *cf cours, TD et TP 6, 9, 10.* On considère ici que les registres SSE sont vus comme des tableaux de 4 valeurs de 32 bits consécutives.

– **pshufd xmm0, xmm1, im8** : permet de recopier les valeurs 32 bits entières de xmm1 dans xmm0 suivant le masque de sélection im8 qui est une constante sur 8 bits

```
xmm0[0] = xmm1[ im8 &       11b ]
xmm0[1] = xmm1[ (im8 &     1100b) >> 2 ]
xmm0[2] = xmm1[ (im8 &   110000b) >> 4 ]
xmm0[3] = xmm1[ (im8 & 11000000b) >> 6 ]
```

– **paddd xmm0, xmm1** : addition vectorielle (en parallèle) des 4 valeurs entières 32 bits de xmm1 avec celles de xmm0, le résultat se trouver dans xmm0

```
xmm0[0] = xmm0[0] + xmm1[0]
xmm0[1] = xmm0[1] + xmm1[1]
xmm0[2] = xmm0[2] + xmm1[2]
xmm0[3] = xmm0[3] + xmm1[3]
```

– **addps** : comme paddd mais en considérant qu'on manipule des float