université angers
FACULTÉ DES SCIENCES
Unité de formation et de recherche

Département Informatique
L3 Informatique
Jean-Michel Richer
Architecture des Ordinateurs
2024/2025[1]

jean-michel.richer@univ-angers.fr

Correction Contrôle Continu 2025

Aucun document (excepté **une** feuille A4 recto/verso), ni outil électronique n'est autorisé. Vous devez soigner votre écriture (la copie ne doit pas être un brouillon) et expliquer clairement votre raisonnement. L'absence de raisonnement ou des étapes de calcul conduit à l'absence de point. En outre, la propreté de la copie, l'orthographe et la grammaire sont prises en compte ainsi que le fait que vous ne rendiez pas votre copie quand cela vous est démandé.

Certains ont perdu 1 point pour avoir rendu une copie avec de nombreuses ratures.

Exercice 1 - 4 pts, 20 min - Soit l'expression suivante : `(x + y) / (x - y)`.

1. donnez une traduction simple en assembleur x86 32 bits de cette expression en utilisant la FPU et en considérant que x et y sont des floats stockés en mémoire :

En passant en notation polonaise inverse : $xy + xy - /$.

```
1 fld   dword [x] ; x
2 fld   dword [y] ; y x
3 faddp st1, st0  ; x+y
4 fld   dword [x] ; x x+y
5 fld   dword [y] ; y x x+y
6 fsubp st1, st0  ; x-y x+y
7 fdivp st1, st0  ; (x+y)/(x-y)
```

2. donnez une traduction simple en assembleur x86 32 bits utilisant la partie basse des registres SSE

On utilise les instructions comme `movss`, `addss`, `subss`. Attention `movss` ne permet que de lire ou écrire vers la mémoire.

```
1 movss xmm1, [x]   ; xmm1 = x
2 movss xmm2, [y]   ; xmm2 = y
3 movss xmm0, xmm1  ; xmm0 = x
4 addss xmm0, xmm2  ; xmm0 = x+y
5 subss xmm1, xmm2  ; xmm1 = x-y
6 divss xmm0, xmm1  ; xmm0 = (x+y)/(x-y)
```

3. optimisez le calcul en ne chargeant qu'une seule fois x et y dans la FPU, on rappelle que `fld st2`, par exemple, permet de dupliquer la valeur dans **`st2`** en la rechargeant dans **`st0`** et que l'on peut écrire, par exemple, `fadd st4, st0` qui additionne le contenu de **`st0`** à **`st4`**. Ou encore `fadd st0, st4` qui additionne le contenu de **`st4`** à **`st0`**. Donnez un tableau qui montre ce que contient chaque registre de la FPU après exécution de chaque instruction.

On utilise les instructions comme `fadd sti, st0` ou `fadd st0, sti`, où i varie entre 0 et 7. Attention, on ne peut pas faire `fadd st2, st1`, cela n'est pas autorisé.

Première solution donnée par gcc :

```
1 fld dword [x]   ; x
2 fld dword [y]   ; y   x
3 fld st1         ; x   y    x
4 fadd st0, st1   ; x+y y    x
5 fxch st2        ; x   y    x+y : echanger st0 et st2
6 fsubrp st1, st0 ; -(y-x) x+y => x-y x+y
7 fdivp st1, st0  ; (x+y)/(x-y)
```

Deuxième solution :

```
1 fld dword [y]   ; y
2 fld dword [x]   ; x y
3 fld st1         ; y x y
4 fld st1         ; x y x y
5 faddp st3, st0  ; y x x+y
6 fsubp st1, st0  ; x-y x+y
7 fdivp st1, st0  ; (x+y)/(x-y)
```

Exercice 2 - 2 pts, 20 min - Soit le code C suivant où x est l'abscisse d'une balle (représentée par un caractère) qui rebondit sur un écran texte de 40 caractères (indices 0 à 39). La variable dx est la direction dans laquelle se déplace la balle ; dx vaut donc 1 ou -1. La fonction new_dx donne donc la nouvelle valeur de dx en fonction du rebond sur le bord gauche ou droit de l'écran.

```
1 int new_dx(int x, int dx) {
2     return (x == 0) - (x == 39) + ((x >0) and (x<39))*dx;
3 }
```

Donnez une traduction **optimisée** en assembleur x86 32 bits de la fonction en indiquant au préalable quelles associations variables / registres vous utilisez.

Il s'agit d'une exclusion mutuelle :

- soit $x = 0$ et on retourne 1, on rebondit vers la droite

- soit $x = 39$ et on retourne -1, on rebondit vers la gauche
- soit x est compris entre 1 et 38 et on retourne dx

La fonction peut donc être réinterprétée si on commence :

- soit avec $x > 0$ et $dx = 1$
- soit avec $x < 39$ et $dx = -1$

```
1  int new_dx(int x, int dx) {
2      if (x == 0) return 1;
3      if (x == 39) return -1
4      return dx;
5  }
```

Pour optimiser la fonction on peut utiliser `cmov` :

```
1  ; ecx = x
2  ; eax = dx, initialement, puis resultat
3  ; edx = valeur a remplacer si condition remplie
4  push  ebp
5  mov   ebp,esp
6  mov   ecx, [ebp+8] ; x
7  mov   eax, [ebp+12]; dx, dernier cas
8  mov   edx, 1
9  test  ecx, ecx     ; si x == 0
10 cmovz eax, edx     ; alors resultat = 1, aller a droite
11 mov   edx, -1
12 cmp   ecx, 39      ; si x == 39
13 cmove eax, edx      ; alors resultat =-1, aller a gauche
14 mov   esp, ebp
15 pop   ebp
16 ret
```

Autre version donnée par gcc :

```
1  push  ebp
2  mov   ebp, esp
3  mov   edx, [ebp+8]
4  xor   eax, eax
5  test  edx, edx
6  sete  al
7  xor   ecx, ecx
8  cmp   edx, 39
9  sete  cl
10 sub   edx, 1
11 sub   eax, ecx
12 cmp   edx, 37
13 mov   edx, 0
14 cmovbe  edx, [ebp + 12]
15 add eax, edx
16 mov   esp, ebp
17 pop   ebp
18 ret
```

Exercice 3 - 8 pts, 50 min - Codage assembleur

```
1  float func(float *x, float *y, float *z, int size) {
2    float prod = 1, sum = 0;
3
4    for (int i = 0; i < size; ++i) {
5      z[i] = x[i] * y[i];
6      prod *= z[i];
7      sum += x[i] + y[i];
8    }
9
10   prod = prod / sum;
11   printf("%f\n", prod);
12
13   return sqrt(prod);
14 }
```

code/cc_mar_exo1.cpp

Note : on considère que les vecteurs `x`, `y` et `z` sont alignés sur des adresses multiples de 16. Le paramètre `size` **n'est pas** multiple de 4

1. donnez le code assembleur x86 32 bits de la fonction `func` en utilisant la FPU. On indiquera au préalable l'association variables / registres.

Traduction classique en assembleur 32 bits

La difficulté réside dans le fait qu'on a deux valeurs à calculer sum et $prod$ que l'on peut stocker dans **`st0`** et **`st1`**, ou alors, si on ne voit pas comment faire, on les stocke dans la pile en `[ebp-4]` et `[ebp-8]`

```
1  ; esi = x, edi =y, ebx = z, edx=size, ecx=i
2  ; sum en [ebp-4] et prod en [ebp-8]
3
4  push   ebp
5  mov    ebp,esp
6  sub    ebp,8      ; reserve sum et prod
7  push   esi
8  push   edi
9  push   ebx
10 mov    esi, [ebp+8]
11 mov    edi, [ebp+12]
12 mov    ebx, [ebp+16]
13 mov    edx, [ebp+20]
14 fldz
15 fstp   dword [ebp-4] ; sum = 0
16 fld1
17 fstp   dword [ebp-8] ; prod = 1
18 xor    ecx, ecx
19 .for:
20     cmp    ecx, edx
21     jge    .endfor
22     fld     dword [ebp-8]         ; prod
23     fld     dword [esi + ecx*4]   ; x prod
24     fmul    dword [edi + ecx*4]   ; x*y prod
25     fst     dword [ebx + ecx*4]   ; z[i] <- x*y
```

```
26      fmulp   st1, st0            ; prod*(x*y)
27      fstp    dword [ebp-8]       ; prod <- prod*(x*y)
28      fld     dword [ebp-4]       ; sum
29      fld     dword [esi + ecx*4] ; x sum
30      fadd    dword [edi + ecx*4] ; x+y sum
31      faddp   st1, st0            ; sum+x+y
32      fstp    dword [ebp-4]       ; sum <- sum+(x+y)
33      inc     ecx
34      jmp     .for
35  .endfor:
36  fld     dword [ebp-8]
37  fdiv    dword [ebp-4]
38  ; printf
39  sub     esp, 8
40  fst     qword [esp]
41  push    dword msg ;  msg: db "%f", 10, 0
42  call    printf
43  add     esp, 12
44  fsqrt
45  pop     ebx
46  pop     edi
47  pop     esi
48  mov     esp, ebp
49  pop     ebp
50  ret
```

2. donnez le code C/C++ avec un dépliage par 4 (lignes 2 à 10)

Dépliage classique, attention dépLIAge et non dépILAge comme je peux le lire

```
1  int i; // sortir le 'i' de la boucle for
2  for (i = 0; i < (size & ~3); i += 4) {
3    z[i] = x[i]*y[i]; ... z[i+3] = x[i+3] * y[i+3];
4    prod *= z[i]; ... prod *= z[i+3];
5    sum += x[i] + y[i]; ... ; sum += x[i+3] + y[i+3]
6  }
7  while (i < size) {
8    z[i] = x[i]*y[i];
9    prod *= z[i];
10   sum += x[i] + y[i];
11   ++i;
12 }
```

3. donnez une version assembleur vectorisée (SSE) de la fonction en utilisant les instructions `mulps`, `addps`, `subps`, `divps`, `haddps`, `movss` et `shufps` ou `pshufd`. On stockera $prod$ et sum respectivement dans **`xmm0`** et **`xmm1`**, $x[i : i + 3]$, $y[i : i + 3]$ dans **`xmm2`**, **`xmm3`**. On utilisera **`edx`** pour stocker $size$ ou un multiple de $size$.

Traduction classique

Traduction classique avec difficulté pour le produit en fin de vectorisation. On ne peut pas faire du `haddps` pour le produit, ni de `hmulps`, instruction qui n'existe pas

```
; sum en [ebp-4] et prod en [ebp-8]
push   ebp
mov    ebp,esp
sub    esp, 8      ; reserve sum et prod
push   esi
push   edi
push   ebx
mov    esi, [ebp+8]
mov    edi, [ebp+12]
mov    ebx, [ebp+16]
mov    edx, [ebp+20]
; prod = [1,1,1,1]
fld1
fstp   dword [ebp-8]
movss  xmm0, [ebp-8]
pshufd xmm0, xmm0, 0 ; ou shufps xmm0=[1,1,1,1]

; sum = [0,0,0,0]
xorps  xmm1, xmm1
and    edx, ~3
; ===== depliage par 4 =====
xor    ecx, ecx
.for:
  cmp     ecx, edx
  jge     .endfor
  movaps  xmm2, [esi + ecx*4] ; ou movdqa
  movaps  xmm3, [edi + ecx*4] ; ou movdqa
  movaps  xmm4, xmm2
  addps   xmm2, xmm3  ; x+y
  mulps   xmm4, xmm3  ; x*y
  movaps  [ebx + ecx*4], xmm3
  mulps   xmm0, xmm4
  addps   xmm1, xmm2
  add     ecx, 4
  jmp     .for
.endfor:
; ===== derniere iterations ======
haddps   xmm1, xmm1
haddps   xmm1, xmm1
movss    [ebp-4], xmm1; sum
; il n'existe pas de hmulps
sub      esp, 16
movdqu   [esp], xmm0  ; prod[,,,]
fld      dword [esp]
fmul     dword [esp+4]
fmul     dword [esp+8]
fmul     dword [esp+12]
fstp     dword [ebp-8]
add      esp, 16

mov      edx, [ebp+20]
; code precedent de la question 1

pop    ebx
pop    edi
pop    esi
```

```
57 ||mov    esp, ebp
58 ||pop    ebp
59 ||ret
```

Remarque : pour positionner `prod=[1,1,1,1]`, étant donné que $1.0 = 3F\_80\_00\_00_{16}$, on aurait pu écrire :

```
1 ||pcmpeqd xmm1, xmm1; 1111_1111.1111_1111.1111_1111.1111_1111 = FF.FF.FF.FF
2 ||psrld   xmm1, 25  ; 0000_0000.0000_0000.0000_0000.0111_1111 = 00.00.00.7F
3 ||pslld   xmm1, 23  ; 0011_1111.1000_0000.0000_0000.0000_0000 = 3F.80.00.00
```

Exercice 4 - 6 pts, 30 min - Répondre aux questions de cours en expliquant et détaillant votre réponse :

1. rappelez quelles sont les 5 étapes de traitement d'une instruction assembleur
2. qu'est-ce que la prédiction de branchement et à quoi sert-elle ?
3. à quoi sert un registre de segment ?
4. à quoi sert la mémoire cache et comment fonctionne t-elle ?
5. qu'est-ce qu'un nombre auto-descriptif ? Donnez un exemple de nombre auto-descriptif.
6. quelles sont les commandes à utiliser pour produire l'exécutable `a.exe` à partir de `a.asm` ?

cf. Cours et TP