Corrigé Examen Novembre 2010 - Architectures Avancées

3H – Tous documents autorisés

Parties indépendantes

OPTIMISATION DE BOUCLES

On utilise le processeur superscalaire défini dans l'annexe 1

Soit la boucle suivante écrite en assembleur, qui travaille sur des tableaux de « floats » X[N], Y[N] et Z[N]. F0 contient la valeur 0.5 ; R1 est contient l'adresse de X[i], R2 l'adresse de Y[i], R3 l'adresse de Z[i] et R4 la valeur de N. :

```
Boucle :   LF F1, 0(R1)
           LF F2, 0(R2)
           FADD F1,F1,F2
           FMUL F1,F1,F0
           SF F1, 0(R3)
           ADDI R1,R1,4
           ADDI R2,R2,4
           ADDI R3,R3,4
           ADDI R4,R4,-1
           BNE R4, Boucle
```

Question 1) Donner le programme C équivalent.

```
float x[N], y[N], z[N], a=0.5;

main ()
{
int i ;
for (i=0;i<256;i++)
      z[i] = (x[i]+ y[i])*a;
```

Question 2) Donner l'exécution cycle par cycle de la boucle en plaçant les instructions dans les différents pipelines E0, E1, FA et FM. Quel est, en nombre de cycles, le temps d'exécution par itération de la boucle originale ?

	E0	E1	FA	FM
Boucle	LD F1, (R1)	LD F2, (R2)		
2	ADDI R1,R1,4	ADDI R2,R2,4		
3	ADDI R3,R3,4	ADDI R4,R4,-1	FADD F1,F1,F2	
4				
5				
6				FMUL F1,F1,F0
7				
8				
9				
10	SD F1, -4(R3)	BNE R4,Boucle		

10 cycles

Question 3) Refaire la question 2) avec un déroulage de boucle d'ordre 4.

	E0	E1	FA	FM
Boucle	LD F1, (R1)	LD F2, (R2)		
2	LD F3,4 (R1)	LD F4, 4(R2)		
3	LD F5, 8(R1)	LD F6, 8(R2)	FADD F1,F1,F2	
4	LD F7, 12(R1)	LD F8, 12(R2)	FADD F3,F3,F4	
5	ADDI R1,R1,16	ADDI R2,R2,16	FADD F5,F5,F6	
6	ADDI R3,R3,16	ADDI R4,R4,-4	FADD F7,F7,F8	FMUL F1,F1,F0
7				FMUL F3,F3,F0
8				FMUL F5,F5,F0
9				FMUL F7,F7,F0
10	SD F1, -16(R3)			
11	SD F3, -12(R3)			
12	SD F5, -8(R3)			
13	SD F7, -4(R3)	BNE R4,Boucle		

13 cycles/4 = 3,25 cycles/itération

PIPELINE LOGICIEL AVEC TMS 320C62

Le code assembleur TMS320C62 ci-dessous donne l'itération du pipeline logiciel pour un programme C qui travaille sur des tableaux de mots de 16 bits signés X[N], Y[N] et Z[N].

A4 est le pointeur sur X[i]
B4 est le pointeur sur Y[i]
A5 est le pointeur sur Z[i]

Initialisation

```
            MVK .S1 100,A1
```

Prologue // non donné

```
Pipeline logiciel

LOOP : LDW .D1 *A4++, A2
      || LDW .D2 *B4++, B2
      || SHR2   .S1  A3,1,A3       // Décalage SIMD à droite d'un bit
      || [A1] B.S2  LOOP          // de mots de 16 bits signés

         ADD2 A2,B2, A3 // ADD2 : addition SIMD de mots de 16 bits signés
      || STW.D1 *A5++, A3
      ||[A1] SUB .S2 A1,2,A1
```

Question 4) Donner le code C correspondant au code assembleur.

```
short x[100], y[100], z[100];

main ()
{
int i ;
for (i=0;i<50;i+=2){
      z[i] = (x[i]+ y[i])/2;
      z[i+1] = (x[i+1]+ y[i+1])/2;
```

Question 5) Quel est le nombre de cycles par itération de la boucle initiale ?

2 cycles pour 2 itération soit 1 cycle/itération

PREDICTION DE BRANCHEMENT

Dans cette partie, on considère le comportement d'un branchement qui est pris (P) ou non pris (N).
On dispose de plusieurs types de prédicteurs
- toujours pris
- toujours non pris
- prédicteur dynamique 1 bit
- prédicteur dynamique 2 bits

Question 6) Choisir un prédicteur pour chacun des branchements suivants, en choisissant celui nécessitant le moins de ressources matérielles si plusieurs prédicteurs ont les mêmes performances

a) Branchement 1
P, P, P, P, P, P, N, P, P, P, P, P, P, P, P, N, P, P, P, P, P, P, P, P, P, N, P, P, P, P, P, P, P, P, P, P, P, N
Prédicteur statique « toujours pris »
b) Branchement 2
P, P, P, P, P, N, N, N, N, N, N, N, N, P, P, P, P, P, P, P, P, P, P, N, N, N, N, N, P, P, P, P, P, P
Prédicteur 1 bit
c) Branchement 3
P, P, P, P, N, P, P, P, P, P, P, P, N, P, P, P, P, P, P, P, P, N, N, N, N, N, P, N, N, N, N, N, N, P, N, N, N, N
Prédicteur 2 bits

SIMD IA-32 (première partie)

Soit le programme C suivant utilisant des instructions SIMD. Il travaille sur des tableaux d'octets non signés X[N], Y[N] et A[N]. XS, YS et AS sont des tableaux de mots de 128 bits. Les trois tableaux d'octets ont la même adresse de début que les trois tableaux de mots de 128 bits correspondants et tous les tableaux sont alignés sur des frontières de mots de 128 bits (les « loads » sont alignés).

```
_m128i a, b, c, d
   for(i=0; i<32; i++) {
      a = ld16(&XS[i]);
      b = ld16(&YS[i]);
      c = subbu (a,b);
      d = subbu (b,a);
      c= maxbu (c,d);
      st16(&AS[i], c)}
```

Question 7) Donner la version C scalaire correspondant à la version SIMD ? Que fait ce programme ?

```
#define vabs(a,b) a>b ? a-b : b-a // valeur absolue (a-b)
unsigned char A[512], X[512], Z[512];
for (i=0; i<512;i++)
        A[i] = vabs (X[i], Y[i]);
```

SIMD IA-32 (deuxième partie)

Le jeu d'instruction IA-32 contient des instructions SIMD de conversion d'entiers 32 bits (int) en flottants 32 bits simple précision (float) : CVTDQ2PS .
On considère des variables 128 bits (_mm128i) v0, v1, v2, v3, v4, v5, v6, v7 pour les entiers et des variables 128 bits (_mm128) f0, f1, f2 et f3 pour les flottants simple précision.

Question 8) En supposant que la variable v0 contient 16 octets (unsigned char) correspondant à des pixels (niveaux de gris), donner la suite des instructions SIMD nécessaires pour convertir les 16 données 8 bits en 16 données de type float dans les variables f0 à f3. Quelle opération faut-il alors effectuer pour avoir des données flottantes comprises entre 0 et 1 ? Quel est le nombre total d'instructions nécessaires pour faire la conversion ?

On pourra fournir la réponse sous forme d'un programme avec des intrinsics (annexe 2), soit sous forme de schémas explicitant le travail de chacune des instructions successives.

```
zero = _mm_xor_si128(zero,zero)
v1= b2hl (v0,zero) // Huit pixels bas de v0 étendus sur 16 bits
v2 = b2hh(v0,zero) // Huit pixels hauts de v0 étendus sur 16 bits
v4 = h2intl(v1, zero) // Quatre pixels bas de v1 étendus sur 32 bits
v5 = h2inth (v1, zero) // Quatre pixels haut de v1 étendus sur 32 bits
v6 = h2intl (v2, zero) // Quatre pixels bas de v2 étendus sur 32 bits
v7 = h2inth (v2, zero) // Quatre pixels haut de v2 étendus sur 32 bits
f0= int2float(v4)          // int converti en float
f1= int2float (v5)         // int converti en float
f2= int2float (v6)         // int converti en float
f3= int2float (v7)         // int converti en float
w = 1/255.0                    // constante 1/255.0
c = _mm_set_ps1(w)             // constante 4 * (1/255.0)
f0 = _mm_mul_ps (f0, c)        // valeurs des pixels entre 0.0 et 1.0
f1 = _mm_mul_ps (f1, c)        // valeurs des pixels entre 0.0 et 1.0
f2 = _mm_mul_ps (f2, c)        // valeurs des pixels entre 0.0 et 1.0
f3 = _mm_mul_ps (f3, c)        // valeurs des pixels entre 0.0 et 1.0
```

CACHES

La figure 1 donne le temps, exprimé en cycles par pixel, de la copie d'une image de N x N pixels avec un octet ou un float par pixel sur un processeur Pentium 4. Dans les deux cas, la copie utilise les instructions SIMD.

Question 9) Expliquer les courbes obtenues.

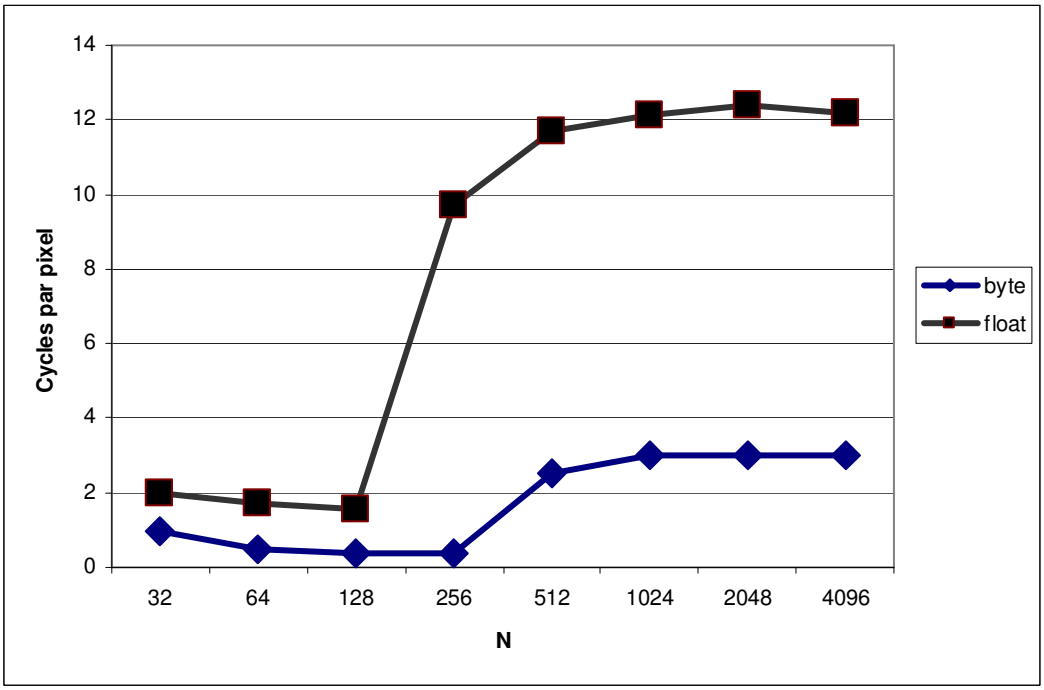


Figure 1) Temps (en cycles par pixel) de la copie d'une image N x N pixels en fonction de N.

On constate que les valeurs de CPP pour les floats (32 bits) sont environ 4 fois celles pour les octets, ce qui est normal puisqu'il faudra transférer quatre fois plus d'octets.
La cassure dans les deux courbes commence à N=128 (floats) et N=256 (octets), ce qui correspond à nouveau à un rapport 4 (copie d'images N * N)
256 * 256 octets = 64 Ko.
512 * 512 octets = 256 Ko
Effet d'un cache L2 du Pentium 4 de 256 Ko

PROGRAMMATION OPENMP

Soit le code C ci-dessous.

```
float X[128], Y[128], Z[128] ;
for (i = 0 ; i<128 ; i++)
      Z[i]= (X[i]+Y[i])*0.5 ;
```

Question 10) Donner une version OpenMP de ce programme pour 4 processeurs qui minimise les défauts de cache.

```
float X[128], Y[128], Z[128] ;
omp_set_num_thread (4);
#pragma omp parallel
{int id;
id = omp_get_thread_num();
sum[id]=0.0;
for (i =id*32 ; i<(id+1)*32 ; i++)
      Z[i]= (X[i]+Y[i])*0.5 ;
}
```

LOI D'AMDAHL

Un programme séquentiel a 10% de son temps d'exécution qui ne peut être parallélisé. On veut l'accélérer à taille constante.

Question 11) Quel est le nombre de processeurs nécessaire pour obtenir une efficacité parallèle de 50% ? (Efficacité parallèle = Accélération / Nombre de processeur).

$$EP = Acc/n = \frac{1}{n*(0{,}1+\frac{0{,}9}{n})} = 0{,}5$$

0,1 n + 0,9 = 2
0,1 n = 1,1
n =11

Annexe 1

Soit un processeur superscalaire à ordonnancement statique qui a les caractéristiques suivantes :
- les instructions sont de longueur fixe (32 bits)
- Il a 32 registres entiers (dont R0=0) de 32 bits et 32 registres flottants (de F0 à F31) de 32 bits.
- Il peut lire et exécuter 4 instructions par cycle.
- L'unité entière contient deux pipelines d'exécution entière sur 32 bits, soit deux additionneurs, deux décaleurs. Tous les bypass possibles sont implantés.
- L'unité flottante contient un pipeline flottant pour l'addition et un pipeline flottant pour la multiplication. - L'unité Load/Store peut exécuter jusqu'à deux chargements par cycle, mais ne peut effectuer qu'un load et un store simultanément. Elle ne peut effectuer qu'un seul store par cycle.
- Il dispose d'un mécanisme de prédiction de branchement qui permet de "brancher" en 1 cycle si la prédiction est correcte. Les sauts et branchements ne sont pas retardés.

La Table 1 donne les instructions disponibles et le pipeline qu'elles utilisent : E0 et E1 sont les deux pipelines entiers, FA est le pipeline flottant de l'addition et FM le pipeline flottant de la multiplication. Les instructions peuvent être exécutées simultanément si elles utilisent chacune un pipeline séparé.
L'addition et la multiplication flottante sont pipelinées. La division flottante n'est pas pipelinée (une division ne peut commencer que lorsque la division précédente est terminée).
L'ordonnancement est statique.

JEU D'INSTRUCTIONS (extrait)

LF	LF Fi, dép.(Ra)	E0 ou E1	Fi ← M (Ra + dépl.16 bits avec ES)
SF	SF Fi, dép.(Ra)	E0	Fi -> M (Ra + dépl.16 bits avec ES)
ADD	ADD Rd,Ra, Rb	E0 ou E1	Rd ←Ra + Rb
ADDI	ADDI Rd, Ra, IMM	E0 ou E1	Rd ←Ra + IMM-16 bits avec ES
SUB	SUB Rd,Ra, Rb	E0 ou E1	Rd ←Ra - Rb
FADD	FADD Fd, Fa, Fb	FA	Fd ←Fa + Fb
FMUL	FMUL Fd, Fa, Fb	FM	Fd ←Fa x Fb
BEQ	BEQ Ri, dépl	E1	si Ri=0 alors CP ← NCP + depl
BNE	BNE Ri, dépl	E1	si Ri≠0 alors CP ← NCP + depl

Table 1 : instructions disponibles

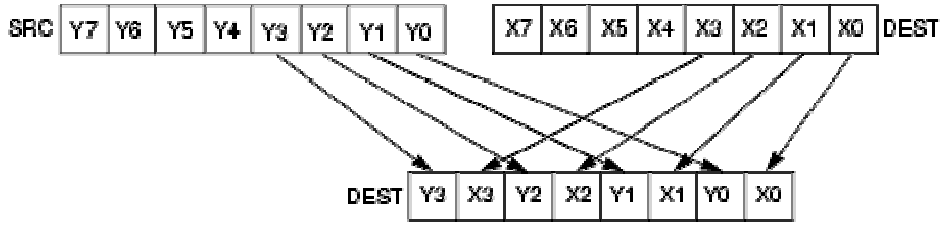
La Table 2 donne la latence entre une instruction source et une instruction destination, dans le cas de dépendances de données. La valeur 1 est le cas où les deux instructions peuvent se succéder normalement, d'un cycle i au cycle i+1.

Latences ***Source*** / ***Destination***		UAL	LF/SF	FADD/	FMUL
UAL		1	2		
LF/ST(adresses)		1	3		
SF(données)		1	2	3	4
Opération flottante			2	3	4

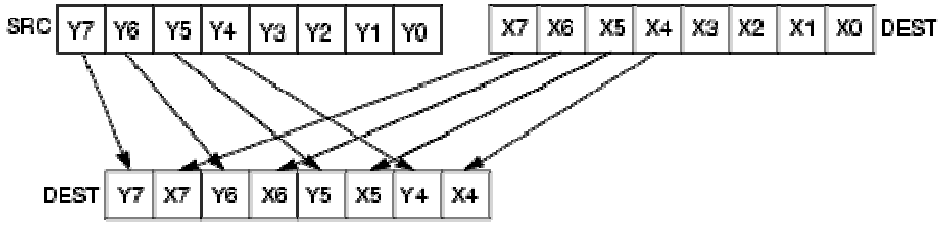
Table 2 : Latences

ANNEXE 2 : Instructions SIMD IA-32 utilisables

#define int2float(a)	_mm_cvtepi32_ps (a)	Convertit 4 entiers 32 bits signés en 32 bits flottants
#define ld16(a)	_mm_load_si128(&a)	chargement aligné
#define st16(a, b)	_mm_store_si128(&a, b)	rangement aligné
#define por(a,b)	_mm_or_si128(a,b)	ou logique
#define pxor(a,b)	_mm_xor_si128(a,b)	ou exclusif
#define maxbu(a,b)	_mm_max_epu8(a,b)	max 8 bits non signés
#define minbu(a,b)	_mm_min_epu8(a,b)	min 8 bits non signés
#define addh(a,b)	_mm_add_epi16(a,b)	addition 16 bits signée
#define addhu(a,b)	_mm_add_epu16(a,b)	addition 16 bits non signée
#define subh(a,b)	_mm_sub_epi16(a,b)	soustraction 16 bits signée
#define subbu (a,b)	_mm_subs_epu8(a,b)	Soustraction 8 bits non signés avec saturation
#define b2hl(a,b)	_mm_unpacklo_epi8 (a,b)	8 octets bas entrelacés vers 8 shorts
#define h2intl(a,b)	_mm_unpacklo_epi116 (a,b)	4 shorts bas entrelacés vers 4 int
#define b2hh(a,b)	_mm_unpackhi_epi8 (a,b)	8 octets haut entrelacés vers 8 shorts
#define h2inth(a,b)	_mm_unpackhi_epi16 (a,b)	4 shorts haut entrelacés vers 4 int
#define h2b(a,b)	_mm_packus_epi16(a,b)	8 shorts (a) dans 8 octets bas - 8 shorts (b) dans 8 octets haut
#define srli128(a,v)	_mm_srli_si128(a,v)	décalage droite des 128 bits de a de v octets
#define slli128(a,v)	_mm_slli_si128(a,v)	décalage gauche des 128 bits de a de v octets
#define srai16(a,v)	_mm_srai_epi16(a,v)	déc. droite des8x 16 bits de a de v bits
#define slli16(a,v)	_mm_slli_epi16(a,v)	déc. gauche des 8x16 bits de a de v bits
#define cmpgt8(a,b)	_mm_cmpgt_epi8(a,b)	comparaison octets signés. Si octet (a)> octet(b) octet résultat = FF_H sinon 00_H



PUNPCKLBW (B2HL)



PUNPCKHBW (B2HH)