

## Examen Mai 2011 - 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 vecteurs de « floats »  $X[N]$ ,  $A[N]$ ,  $B[N]$ ,  $C[N]$  et  $D[N]$  ; R1 contient la valeur 0001 0000H et R2 contient la valeur N

Les adresses des vecteurs sont

0001 0000 <sub>H</sub>	X(0)
0001 1000 <sub>H</sub>	A(0)
0001 2000 <sub>H</sub>	B(0)
0001 3000 <sub>H</sub>	C(0)
0001 4000 <sub>H</sub>	D(0)

Soit la boucle assembleur

```
Boucle :   LF F1, 1000H (R1)
           LF F2, 2000H(R1)
           LF F3, 3000H (R1)
           LF F4, 4000H(R1)
           FMUL F1,F1,F2
           FMUL F3,F3,F4
           FADD F1,F1,F3
           SF F1, 0(R1)
           ADDI R1,R1,4
           ADDI R2,R2,-1
           BNE R2, Boucle
```

**Question 1) Donner le programme C équivalent.**

**Question 2) On suppose d'abord que l'on utilise un processeur scalaire (une seule instruction démarrée par cycle). Optimisez la suite d'instructions. Quel est, en nombre de cycles, le temps d'exécution par itération de la boucle originale ?**

**Question 3) Avec le processeur superscalaire, 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 ?**

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

#### **PIPELINE LOGICIEL AVEC TMS 320C62**

Le code assembleur TMS320C62 ci-dessous donne l'itération du pipeline logiciel pour un programme C.

A4 contient au départ l'adresse d'un tableau d'entiers 16 bits  $X[N]$  et B4 contient l'adresse d'un tableau d'entiers 16 bits  $Y[N]$ .

L'instruction MPY effectue une multiplication 16 bits par 16 bits sur les parties basses des registres et MPYH effectue une multiplication 16 bits par 16 bits sur les parties hautes des registres. L'instruction CMPGT A6,0,A0 met 1 dans le registre A0 si  $A6 > 0$  et 0 sinon.

```
LOOP:
        LDW    .D1    *A4++,A2        ;
||      LDW    .D2    *B4++,B2        ;
|| [A0]  ADD    .L1    A6,A7,A7        ;
|| [B0]  ADD    .L2    B6,B7,B7        ;
|| [A1]  B      .S2    LOOP            ;
        MPY    .M1X   A2,B2,A6        ;
||      MPYH   .M2X   A2,B2,B6        ;
||      CMPGT  .L1    A6,0,A0         ;
||      CMPGT  .L2    B6,0,B0         ;
|| [A1]  SUB    .S1    A1,1,A1        ;
        ADD    .L1X   A7,B7,A4        ;
```

**Question 5 : Donner le code C correspondant au code assembleur.**

**Question 6 : Quel est l'intervalle inter-itération (II) ? Justifier la valeur.**

### **PREDICTION DE BRANCHEMENT**

Un branchement a le comportement suivant, où "P signifie pris et N signifie non pris.

PPNPPPPNPPPPNPPPPNPPPPNPPPPNPPPPN....

**Question 7 : après la phase d'initialisation, quelle est le pourcentage de bonnes prédictions ?**

- a) avec un prédicteur 1 bit
- b) avec un prédicteur 2 bits.

### **SIMD IA-32**

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 = addu(a,b);
    st16(&AS[i], c)}
```

**Question 8) Donner la version C scalaire correspondant à la version SIMD ? Que fait ce programme ?**

### **CACHES**

On considère un cache de 16Ko, associatif 2 voies (2 blocs par ensemble) avec des blocs de 64 octets. Le cache est à écriture simultanée « non allouée » (pas de défauts de cache en écriture).

Soit le programme

```
int A[1024*1024], int X = 0 ; // A[0] est implanté à l'adresse 0x0 ;
For (i=0 ; i<1024 ; i++)
X+=A[i];
```

**Question 9 : Quel est le taux d'échecs (nombre d'échecs/nombre d'accès) ?**

Le programme devient  
For (i=0 ; i<1024 ; i++)  
X+=A[i] + A[i\*1024] ;

**Question 10 : Quel est maintenant le taux d'échec ?**

**LOI D'AMDAHL**

Un programme séquentiel a 20% 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).**

**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. La table 1 donne également les latences des instructions.

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)

Instruction	Latence	Syntaxe assembleur	Pipeline	Action
LF	2	LF $F_i$ , dép.(Ra)	E0 ou E1	$F_i \leftarrow M$ (Ra + dépl.16 bits avec ES)
SF	1	SF $F_i$ , dép.(Ra)	E0	$F_i \rightarrow M$ (Ra + dépl.16 bits avec ES)
ADD	1	ADD Rd,Ra, Rb	E0 ou E1	$R_d \leftarrow R_a + R_b$
ADDI	1	ADDI Rd, Ra, IMM	E0 ou E1	$R_d \leftarrow R_a + \text{IMM}-16$ bits avec ES
SUB	1	SUB Rd,Ra, Rb	E0 ou E1	$R_d \leftarrow R_a - R_b$
FADD	3	FADD Fd, Fa, Fb	FA	$F_d \leftarrow F_a + F_b$
FMUL	3	FMUL Fd, Fa, Fb	FM	$F_d \leftarrow F_a \times F_b$
BEQ	1	BEQ $R_i$ , dépl	E1	si $R_i=0$ alors $CP \leftarrow NCP + \text{depl}$
BNE	1	BNE $R_i$ , dépl	E1	si $R_i \neq 0$ alors $CP \leftarrow NCP + \text{depl}$

Table 1 : instructions disponibles

**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_epi16 (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 srl128(a,v)	_mm_srli_si128(a,v)	décalage droite des 128 bits de a de v octets
#define sll128(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 des 8x16 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 <sub>H</sub> sinon 00 <sub>H</sub>