

## Corrigé Examen Janvier 06 - Architectures Avancées

3H – Tous documents autorisés

### PIPELINES

Un processeur a les pipelines suivants pour les trois types d'instructions\*

- Instructions entières  
LI1 LI2 DI LR EX1 EX2 ER
- Instructions mémoire  
LI1 LI2 DI LR EX1 AM1 AM2 ER
- Instructions de multiplication-accumulation  
LI1 LI2 DI LR M1 M2 M3 ER

Les différents étages ont la signification suivante :

- LI1 et LI2 correspondent à l'accès au cache instructions
- DI : décodage
- LR : Lecture des opérandes dans le banc de registres
- EX1 : Calculs UAL pour les opérations arithmétiques et logiques et pour les calculs d'adresse mémoire et de branchement. Si un branchement a été mal prédit, toutes les instructions dans les étages précédents du pipeline sont annulées et l'on réinitialise le pipeline à partir de la bonne adresse
- EX2 Cet étage choisit ce qui va être écrit dans le banc de registres dans les cycles ER
- AM1 et AM2 : accès au cache données
- M1, M2, M3 : multiplication entière pipelinée
- ER : Ecrire du résultat dans le banc de registres.

Tous les circuits d'anticipation (bypass) existent.

### Question 1

a) Donner les latences entre instruction producteur et instruction consommateur en nombre de cycles.

**NB : la valeur n signifie que deux instructions dépendantes peuvent se suivre aux cycles i et i+n, que ce soit pour un processeur scalaire ou superscalaire.** Une valeur 0 signifie que les deux instructions peuvent démarrer au même cycle (pour un superscalaire)

	Instruction producteur	Instruction consommateur	Latence
a	UAL Ri, -, -	UAL -, Ri, -	1
b	UAL Ri, -, -	ST Ri, ()	1
c	UAL Ri, -, -	LW Rj, (Ri)	1
d	LW Ri, ()	UAL -, Ri, -	3
e	UAL Ri	MUL -, Ri, -	1
f	LW Ri, ()	MUL -, Ri, -	3
g	MUL Ri,	ST Ri, ()	3
h	LW Ri, ()	ST Ri, ()	3

b) Quelle est la pénalité de branchement conditionnel mal prédit ?  
4 cycles

### GRAPHIQUE ET SIMD

Soit le code suivant pour un filtre 3x3 (EEMBC High-Pass Gray Filter)

```
int i, j;
unsigned char X[size][size], X[size][size];
short temp;
for(i=1; i<size-1; i++) {
  for(j=1; j<size-1; j++) {
    tmp = (short) (F11* X[i-1][j-1]+F21* X[i-1][j]+F31* X[i-1][j+1]
      + F12* X[i][j-1]+F22* X[i][j]+F32* X[i][j+1]
      + F13* X[i+1][j-1]+F23* X[i+1][j]+F33* X[i+1][j+1]);
    Y[i][j] = (unsigned char) (tmp>>8);
  }
}
```

**Figure 1 : Filtre 3 x 3**

Le coefficient F22 a la valeur 255 et tous les autres coefficients ont la valeur -28.

**Question 2 :**

Ecrire une version C scalaire optimisée du programme de la figure 1

```
int i, j;
short tmp1, tmp2, tmp3;
for(i=1; i<size-1; i++) {
  for(j=1; j<size-1; j++) {
    tmp1 = (short)((X[i][j]<<8) - X[i][j]);
    tmp2 = (short)(X[i-1][j-1]+X[i-1][j]+X[i-1][j+1]+X[i][j-1]+X[i][j+1]+X[i+1][j-1] +
      X[i+1][j] + X[i+1][j+1]);
    tmp2 = (tmp2<<2) -(tmp2<<5);
    Y[i][j] = (byte) ((tmp1+tmp2)>>8); } }
```

**Question 3 :**

On utilise maintenant le processeur NIOS II pour lequel on veut définir des instructions SIMD 4x8 ou 2x16 spéciales.

- a) donner la liste des instructions SIMD ajoutées, en précisant (informellement) leur action et en leur attribuant un mnémonique

Instruction	Mnémo	
Décalage gauche	DECG	Octets alignés j+4, j+3, j+2, j+1
Décalage droit	DECD	Octets alignés j+2, j+1, j, j-1
Octets bas à demi-mot	B2HL	Les deux octets bas d'un mot donnent deux entiers sur 16 bits
Octets haut à demi-mot	B2HH	Les deux octets haut d'un mot donnent deux entiers sur 16 bits
Décalage droite de 8 bits	SHR8	Les mots de 16 bits sont divisés par 256 (2 <sup>8</sup> )
Décalage gauche de 5 bits	SHL5	Les mots de 16 bits sont multipliés par 32
Décalage gauche de 2 bits	SHL2	Les mots de 16 bits sont multipliés par 4
Addition 2x16 bits	ADDH	Addition SIMD
Soustraction 2x16	SUBH	Soustraction SIMD
Demi-mots vers octets bas	H2BL	Conversion 2x16 vers 2x8 bas
Demi-mots vers octets hauts	H2BH	Conversion 2x16 vers 2x8 haut, avec 2x8 bas conservé

- b) Ecrire la version C SIMD optimisée utilisant les instructions définies en a)

```

t1p= DECG(XS[i][j],XS[i][j+1]); //XS[i][j+1]
t1m= DECD(XS[i][j],XS[i][j-1]); //XS[i][j-1]
t2p= DECG(XS[i-1][j],XS[i-1][j+1]); //XS[i-1][j+1]
t2m= DECD(XS[i-1][j],XS[i-1][j-1]); //XS[i-1][j-1]
t3p= DECD(XS[i+1][j],XS[i+1][j-1]); //XS[i+1][j+1]
t3m= DECD(XS[i+1][j],XS[i+1][j-1]); //XS[i+1][j-1]

t1b=B2HL(XS[i][j]);
t1h=B2HH(XS[i][j]);
t2b=B2HL(XS[i-1][j]);
t2h=B2HH(XS[i-1][j]);
t3b=B2HL(XS[i+1][j]);
t3h=B2HH(XS[i+1][j]);
t1pb=B2HL(t1p);
t1ph=B2HH(t1p);
t1mb=B2HL(t1m);
t1mh=B2HH(t1m);
t2pb=B2HL(t2p);
t2ph=B2HH(t2p);
t2mb=B2HL(t2m);
t2mh=B2HH(t2m);
t3pb=B2HL(t3p);
t3ph=B2HH(t3p);
t3mb=B2HH(t3m);
t3mh=B2HH(t3m);
tmp1=SHL8(t1b);
tmp1=SUBH(tmp1, t1b);
tmp2= ADDH(t2b,t3b);
tmp2= ADDH(tmp2, t1pb);
tmp2= ADDH(tmp2, t1mb);
tmp2= ADDH(tmp2, t2mb);
tmp2= ADDH(tmp2, t2pb);
tmp2= ADDH(tmp2, t3mb);
tmp2= ADDH(tmp2, t3pb);
tmp3 =SHL2(tmp2);
tmp2= SHL5(tmp2);
tmp3= SUBH(tmp3, tmp2);
t1b= ADDH(tmp1,tmp3);
t1b=SHR8(t1b);
tmp1=SHL8(t1h);
tmp1=SUBH(tmp1, t1h);
tmp2= ADDH(t2h,t3h);
tmp2= ADDH(tmp2, t1ph);
tmp2= ADDH(tmp2, t1mh);
tmp2= ADDH(tmp2, t2mh);
tmp2= ADDH(tmp2, t2ph);
tmp2= ADDH(tmp2, t3mh);
tmp2= ADDH(tmp2, t3ph);
tmp3 =SHL2(tmp2);
tmp2= SHL5(tmp2);
tmp3= SUBH(tmp3, tmp2);
t1h= ADDH(tmp1,tmp3);
t1h=SHR8(t1h);
t1b=H2BL(t1b);
YS[i][j]=H2BH(t1h,t1b);}}

```

## **PREDICTION DE BRANCHEMENTS**

Dans cette partie, P signifie qu'un branchement est Pris et N qu'il est non pris. On a le choix entre

- un prédicteur statique
- un prédicteur dynamique 1 bit

**Question 4 : pour chacun des branchements suivants (considérés chacun séparément), proposer un type de prédicteur en justifiant. Si plusieurs prédicteurs conviennent, on choisira celui qui utilise le moins de matériel**

- a) Branchement B1,  
P,P,P,P,P,N,N,N,N,N,N,N,N,N,P,P,P,P,P,P,P,P,P,N,N,N,N,N,P,P,P,P,P,P  
Prédicteur 1 bit : branchements pris et non pris, avec longues suites identiques.
- b) Branchement B2  
P,P,P,P,N,P,P,P,P,P,P,N,P,P,P,P,P,P,N,N,N,N,N,P,N,N,N,N,N,P,N,N,N,N,N  
Prédicteur 2 bits. Longue suite avec même comportement, interrompu par un seul branchement avec comportement opposé
- c) Branchement B3  
P,P,P,P,P,N,P,P,P,P,P,P,N,P,P,P,P,P,P,P,N,P,P,P,P,P,P,P,N,P,P,P,P,P,P  
Prédiction statique pris. Le branchement est toujours pris, sauf exception

**OPTIMISATION DE BOUCLES**

On utilise le processeur superscalaire défini dans l'annexe 1 et les programmes P1 et P2 suivants

P1	P2
float X[N], Y[N], Z[N] ; int i ;	float X[N], Y[N], Z[N], a , un=1.0; int i ;
for (i=0 ; i<N ; i++) Z[i]= X[i] + Y[i] ;	for (i=0 ; i<N ; i++) Z[i]= X[i] /a + Y[i] ;

On supposera que l'adresse de X[0] est initialement dans R1, que l'adresse de Y[0] est initialement dans R2 et que l'adresse de Z[0] est initialement dans R3. R4 contient initialement le nombre d'itérations de la boucle.

**Question 5 : On considère le programme P1**

- a) Quel est en nombre de cycles, le temps d'exécution par itération pour le programme P1 (optimisée, mais sans déroulage de boucle) ?
- b) Quel est en nombre de cycles, le temps d'exécution par itération de la boucle initiale, avec un déroulage d'ordre 4 (4 itérations initiales par itération de boucle) ?

	E0	E1	FA	FM
1Boucle :	LF F1,0(R1)	LF F2, 0(R2)		
2	ADDI R1,R1,4	ADDI R2,R2,4		
3	ADDI R3,R3,4	ADDI R4,R4,-1	FADD F3,F1,F2	
4				
5				
6	SF F3, -4(R3)	BNE R4,Boucle		

6 cycles par itération

	E0	E1	FA	FM
1Boucle :	LF F1,0(R1)	LF F2, 0(R2)		
2	LF F3,4(R1)	LF F4, 4(R2)		

3	LF F5,8(R1)	LF F6, 8(R2)	FADD F9,F1,F2	
4	LF F7,12(R1)	LF F8, 12(R2)	FADD F10,F3,F4	
5	ADDI R1,R1,16	ADDI R2,R2,16	FADD F11,F5,F6	
6	SF F9, -16(R3)	ADDI R3,R3,16	FADD F12,F7,F8	
7	SF F10, -12(R3)	ADDI R4,R4,-4		
8	SF F11, -8(R3)			
9	SF F12, -4(R3)	BNE R4,Boucle		

9 cycles pour 4 itérations soit 2,25 cycles/itération

**Question 6 : On considère le programme P2**

- Quel est en nombre de cycles, le temps d'exécution par itération pour le programme P2 (après optimisation, mais sans déroulage de boucle) ? Quel est le temps d'exécution total si  $N=100$  ?
- Peut-on utiliser le déroulage de boucle ? Si non, pourquoi ? Si oui, donner le temps d'exécution du programme pour  $N=100$  avec un déroulage de boucle d'ordre 4.

	E0	E1	FA	FM
	LF F1,a	LF F2,un		
			FDIV F0,F2,F1	
1Boucle :	LF F1,0(R1)	LF F2, 0(R2)		
2	ADDI R1,R1,4	ADDI R2,R2,4		
3	ADDI R3,R3,4	ADDI R4,R4,-1		FMUL F1,F1,F0
4				
5				
6				
7				
8			FADD F3,F1,F2	
9				
10				
11	SF F3, -4(R3)	BNE R4,Boucle		

11 cycles par itération.

16 cycles de pénalité d'initialisation

Pour  $N=100$ , on a donc 1116 cycles.

	E0	E1	FA	FM
	LF F1,a	LF F2,un		
			FDIV F0,F2,F1	
			(latence 20 cycles)	
1Boucle :	LF F1,0(R1)	LF F2, 0(R2)		
2	LF F3,4(R1)	LF F4, 4(R2)		
3	LF F5,8(R1)	LF F6, 8(R2)		FMUL F1,F1,F0
4	LF F7,12(R1)	LF F8, 12(R2)		FMUL F3,F3,F0
5	ADDI R1,R1,16	ADDI R2,R2,16		FMUL F5,F5,F0
6	ADDI R3,R3,16	ADDI R4,R4,-4		FMUL F7,F7,F0
7				

8			FADD F1,F1,F2	
9			FADD F3,F3,F4	
10			FADD F5,F5,F6	
11	SF F3, -16(R3)		FADD F7,F7,F8	
12	SF F3, -12(R3)			
13	SF F3, -8(R3)			
14	SF F3, -4(R3)	BNE R4,Boucle		

14 cycles pour 4 itérations soit 3,5 cycles par itération

16 cycles de pénalité pour l'initialisation

Total pour 100 itérations : 366 cycles

### **Optimisations logicielles**

On considère un processeur scalaire dont les instructions et les latences sont données dans les tables 1 et 2 .

Soit le programme suivant, dans lequel on suppose que R3 contient au départ l'adresse de X[0], R4 l'adresse de Y[0], R5 l'adresse de Z[0] et R2 contient N-2.

X[i], Y[i] et Z[i] sont des flottants simple précision

**Question 7 : Que fait le programme suivant ? Quel est son temps d'exécution si N=100 ? Quelle technique est employée ?**

	LF F3, (R3)	Prologue : 1
	LF F4,(R4)	2-3
	FADD F5,F3,F4	4
	LF F3, 4(R3)	5
	LF F4, 4(R4)	6
	ADDI R3, R3,8	7
	ADDI R4, R4,8	8
L1 :	SF F5, (R5)	L1 :1
	FADD F5,F3,F4	2
	LF F3, 0(R3)	3
	LF F4, 0(R4)	4
	ADDI R2,R2,-1	5
	ADDI R3,R3,4	6
	ADDI R4,R4,4	7
	ADDI R5,R5,4	8
	BNEQ R2, L1	9
	SF F5, (R5)	Epilogue :1
	FADD F5,F3,F4	2-3-4
	SF F5, 4(R5)	5

Le programme calcule

for (i=0 ;i<N ; i++)

Z[i]=X[i]+Y[i] ;

En utilisant le pipeline logiciel

Le temps d'exécution est  $98 \times 9 + 8 + 5 = 895$  cycles

### **Pipeline logiciel avec IA-64**

**Question 8 : Que font les boucles des programmes 1 et 2 ?**

Les deux programmes calculent  $Z[i]=X[i]+Y[i]$ , le programme 1 sur des entiers et le programme 2 sur des flottants simple précision.

IA-64 – Programme 1	IA-64 – Programme 2
<pre>L1 : { (p16)ld4    r36=[r8],4   (p16)ld4    r37=[r3],4     nop.i} { (p17)st4    [r2]=r39,4 ;;   (p16)lfetch.nt1 [r35]   (p16)add    r38=r36,r37    } {    nop.m 0   (p16)add    r32=12,r35     br.ctop.sptk .L1 ;;  }</pre>	<pre>L2 : { (p16)ldfs   f32=[r8],4     nop.i 0     nop.i 0} { (p16)ldfs   f38=[r3],4   (p16)lfetch.nt1 [r35]     nop.b 0 ;;} { (p23)stfs   [r2]=f46,4   (p21)fma.s   f44=f37,f1,f43   (p16)add    r32=12,r35    } {    nop.m 0     nop.m     br.ctop.sptk .L2 ;;  }</pre>

Figure 2 : Code IA-64

## CACHES

Soit le code naïf de la transposition d'une matrice  $[N][N]$  :

```
void transpose (unsigned char **X, int N ,unsigned char **Y){
int i,j ;
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        Y[j][i]=X[i][j];}
```

Figure 3 : code naïf de la transposition

On utilise un processeur Pentium 4 dont les caches ont les caractéristiques suivantes :

- Cache données L1 de 8 Ko, associatif 4 voies avec des blocs de 64 octets. L'écriture simultanée dans le cache L2 est utilisée lors des défauts de cache L1 en écriture
- Cache L2 de 512 Ko, associatif 8 voies avec des blocs de 128 octets. La réécriture est utilisée lors des défauts de cache L2 en écriture.

L'exécution de la transposition (code de la figure 3) donne les résultats suivants :

Lena512 : 14,2 CPP  
Lena1024 : 153,1 CPP

**Question 9: Est que l'augmentation spectaculaire du temps d'exécution lorsqu'on passe de  $N=512$  à  $N=1024$  provient de défauts de capacité ou de défauts de conflit ? (Justifier). Quelle expérience simple permettrait de vérifier ?**

Le cache L2 a 512 Ko, et est associatif 8 voies. Il y a 64 K ensembles de 8 blocs.

L'écriture dans  $Y[j][i]$  se fait avec un pas de 512 Octets pour Lena512 et 1024 octets pour Lena1024.

Avec un pas de 512 octets, si un élément est dans l'ensemble 0, le suivant est dans l'ensemble 2, le suivant dans l'ensemble 4, etc. 4 des 8 voies sont donc disponibles soit 256 Ko.

La transposition d'une ligne provoque  $512/128 = 4$  défauts de cache en lecture et 512 défauts de cache en écriture qui correspondent à  $512 + 512*128 = 129 * 512$  soit environ

64 Ko du cache L2. La transposition des 127 lignes suivantes provoquera des défauts de caches en lecture, mais pas en écriture.

Avec un pas de 1024 octets, si un élément est dans l'ensemble 0, le suivant est dans l'ensemble 4, le suivant dans l'ensemble 8, etc. Il y a donc uniquement 2 voies sur les 8 qui sont utilisables soit 128 Ko.

La transposition d'une ligne provoque  $1024/128 = 8$  défauts en lecture et 1024 défauts en écriture qui correspondent à  $1024 + 1024*128 = 129 * 1024$  soit 129 Ko. Il y a défaut de conflit et remplacement qui provoquent un défaut en écriture à chaque itération pour chaque nouvelle ligne.

Pour vérifier le défaut de conflit, on pourrait mesurer le temps de transposition d'une matrice  $1025 * 1025$

### **Annexe 1 : superscalaire à ordonnancement statique**

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 (R0=0) de 32 bits et 32 registres flottants (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. L'ordonnancement est statique. Les chargements ne peuvent pas passer devant les rangements en attente.
- 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**

- les instructions disponibles

- 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).

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  $i+1$ .

JEU D'INSTRUCTIONS (extrait)

LF	LF $F_i$ , dép.( $R_a$ )	E0 ou E1	$F_i \leftarrow M(R_a + \text{dépl.16 bits avec ES})$
SF	SF $F_i$ , dép.( $R_a$ )	E0	$F_i \rightarrow M(R_a + \text{dépl.16 bits avec ES})$
ADD	ADD $R_d, R_a, R_b$	E0 ou E1	$R_d \leftarrow R_a + R_b$
ADDI	ADDI $R_d, R_a, \text{IMM}$	E0 ou E1	$R_d \leftarrow R_a + \text{IMM-16 bits avec ES}$
SUB	SUB $R_d, R_a, R_b$	E0 ou E1	$R_d \leftarrow R_a - R_b$
FADD	FADD $F_d, F_a, F_b$	FA	$F_d \leftarrow F_a + F_b$
FMUL	FMUL $F_d, F_a, F_b$	FM	$F_d \leftarrow F_a \times F_b$
FDIV	FDIV $F_d, F_a, F_b$	FA	$F_d \leftarrow F_a / F_b$
BEQ	BEQ $R_i$ , dépl	E1	si $R_i=0$ alors $CP \leftarrow NCP + \text{depl}$
BNE	BNE $R_i$ , dépl	E1	si $R_i \neq 0$ alors $CP \leftarrow NCP + \text{depl}$

**Table 1 : instructions disponibles**

Latences	<i>Source</i>	UAL	LF/SF (données)	FADD	FMUL	FDIV	FSQRT
<i>Destination</i>							
UAL		1	2				
LF/ST (adresses)		1					
SF (données)		1	2	3	5	20 (NP)	20 (NP)
Opération flottante			2	3	5	20 (NP)	20 (NP)

**Table 2 : latences**

**Annexe2 :**

Dans IA-64, les registres flottants f0 et f1 sont câblés : f0=0.0 et f1=1.0.

L'instruction fma a l'action suivante :

$$fr = fa, fb, fc \text{ correspond à } fr = fa * fb + fc$$

Latence des instructions IA-64 disponibles

Consommateur (à droite) Producteur (en bas)	UAL	Adresse load store	Store données	Inst. flottantes	getf	setf
Instructions UAL	1	1	1			1
getf	5/9	6/9	5/9			6/9
setf			6/2			
Instructions flottantes			4/5	4/5	4/5	
Load entier	N	N+1	N		N	N
Load flottant	M+1		M+1	M+1	M+1	M+1

**Table 3 : Latence des instructions Itanium 2 et Itanium (sous la forme v2/v1 quand elles sont différentes)**

N=1/2 pour cache L1D, N=8/5 pour L2, N=21/12-15 pour L3, N=180-220 pour MP  
M = 5 pour L2, M=24/12-15 pour L3, M=180-220 pour MP