Corrigé Examen Décembre 07 - Architectures Avancées

3H – Tous documents autorisés

PIPELINES

Un processeur a les pipelines suivants :

Opérations UAL entières	LI	DI/LO	EX	MEM	RR	
Opérations flottantes SP et DP	LI	DI/LO	EX1	EX2	EX3	RR
Opérations mémoire 32 bits (LW, LF,SW,SF)	LI	DI/LO	EX	MEM	RR	
Opérations mémoire 64 bits (LD et SD)	LI	DI/LO	EX	MEM1	MEM2	RR

Les actions exécutées dans les différentes phases sont les suivantes :
LI : Lecture de l'instruction
DI/LO : Décodage de l'instruction et lecture des opérandes dans les registres entiers ou flottants.
EX : Exécution de l'opération arithmétique ou logique dans l'UAL ou calcul de l'adresse mémoire (instructions Load et Store) ou calcul de l'adresse de branchement.
MEM : lecture ou écriture dans le cache donnée (mot de 32 bits)
RR : Rangement du résultat dans le registre entier ou le registre flottant.

Q 1) Dans l'hypothèse où existent tous les "court-circuits" nécessaires, donner la latence de la première instruction (en nombre de cycles) lorsqu'elle est suivie de la seconde instructions pour toutes les configurations suivantes :

NB : la syntaxe des instructions est **code op, destination, source 1, source 2**

a) LW Ri, dep (Rj) suivi de ADD Rk, Ri, Rl

LI	DI/LO	EX	**MEM**	RR		
		LI	DI/LO	**EX**	MEM	RR

b) ADD Ri, Rk, Rl suivi de SW Ri, dep (Rj)

LI	DI/LO	**EX**	MEM	RR	
	LI	DI/LO	EX	**MEM**	RR

c) ADD Rj, Rk, Rl suivi de SW Ri, dep (Rj)

LI	DI/LO	**EX**	MEM	RR	
	LI	DI/LO	**EX**	MEM	RR

d) ADD Rj, Rk, Rl suivi de BNEZ Rj, dep

LI	DI/LO	**EX**	MEM	RR	
	LI	DI/LO	**EX**	MEM	RR

e) FADDS Fi, Fj, Fk suivi de FADDS Fl, Fm, Fi (simple précision)

LI	DI/LO	EX1	EX2	**EX3**	RR			
			LI	DI/LO	**EX1**	EX2	EX3	RR

f) FMULD Fi, Fj, Fk suivi de SD Fi , dep (Rm) (double précision)

LI	DI/LO	EX1	EX2	**EX3**	RR		
		LI	DI/LO	EX	**MEM1**	MEM2	RR

Q 2) Quel est en nombre de cycles la pénalité de mauvaise prédiction de branchement ?

OPTIMISATION DE BOUCLES

On utilise maintenant le processeur superscalaire décrit dans l'annexe 1. Les instructions ont les latences définies dans l'annexe 1 table 2.

Soit le programme P1 suivant :

```
float X[1024], Y[1024], Z[1024], A, B;
for (i=0 ; i<1024 ; i++)
      Z[i] = A*X[i] – B*Y[i] ;
```

On supposera que l'adresse de X[0] est initialement dans R1, que l'adresse de Y[0] est initialement dans R2, que l'adresse de Z[0] est initialement dans R3. F10 contient la valeur de A, F11 contient la valeur de B et R4 contient initialement le nombre d'itérations de la boucle.

Q 3 Quel est en nombre de cycles, le temps d'exécution par itération de la boucle originale sans et avec utilisation des instructions SIMD. ?

Sans SIMD : 10 cycles/itération
Avec SIMD : 10/4 = 2,5 cycles/itération

Sans déroulage, sans SIMD

	E0	E1	FA	FM
1 Boucle	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**,F10
4				FMUL **F2**,F2,F11
5				
6				
7			FSUB F1,F1,**F2**	
8				
9				
10	SF F1-4(R3)	BNE R4, Boucle		

Sans déroulage, avec SIMD

	E0	E1	FA	FM
1 Boucle	PLF S1, 0(R1)	ADDI R1,R1,16		
2	PLF S2, 0(R2)	ADDI R2,R2,16		
3	ADDI R3,R3,16	ADDI R4,R4,-4		PFMULS S1,S1,F10
4				PFMULS S2,S2,F11
5				
6				
7			PFSUB S1,S1,S2	
8				
9				
10	PSF S1, -16(R3)	BNE R4, Boucle		

Q 4) Donner par itération de la boucle initiale

- le nombre de cycles sans instructions SIMD avec un déroulage d'ordre 4
- le nombre de cycles avec instructions SIMD avec un déroulage d'ordre 4

Avec déroulage, sans SIMD : 16/4 = 4 cycles/itération
Avec déroulage, avec SIMD : 16/ 16 = 1 cycles/itération

Avec déroulage, sans SIMD

	E0	E1	FA	FM
1 Boucle	LF F1, 0(R1)	LF F2, 0(R2)		
2	LF F3, 4(R1)	LF F4, 0(R2)		
3	LF F5, 8(R1)	LF F6, 8(R2)		FMUL F1,F1,F10
4	LF F7, 12(R1)	LF F8, 12(R2)		FMUL F2,F2,F11
5	ADDI R1,R1,16	ADDI R2,R2,16		FMUL F3,F3,F10
6	ADDI R3,R3,16	ADDI R4,R4,-4		FMUL F4,F4,F11
7			FSUB F1,F1,F2	FMUL F5,F5,F10
8				FMUL F6,F6,F11
9			FSUB F3,F3,F4	FMUL F7,F7,F10
10	SF F1, -16(R3)			FMUL F8,F8,F11
11			FSUB F5,F5,F6	
12	SF F3, -12(R3)			
13			FSUB F7,F7,F8	
14	SF F5, -8(R3)			
15				
16	SF F7, -4(R3)	BNE R4, Boucle		

Avec déroulage, SIMD

	E0	E1	FA	FM
1 Boucle	PLF S1, 0(R1)	ADDI R1,R1,64		
2	PLF S2, 0(R2)	ADDI R2,R2,64		
3	PLF S3, 16(R1)	ADDI R3,R3,64		
4	PLF S4, 16(R2)	ADDI R4,R4,-16		PFMULS S1,S1,F10
5	PLF S5, 32(R1)			PFMULS S2,S2,F11

6	PLF S6, 32(R2)			PFMULS S3,S3,F10
7	PLF S7, 48(R1)		PFSUB S1,S1,S2	PFMULS S4,S4,F11
8	PLF S8, 48(R2)			PFMULS S5,S5,F10
9			PFSUB S3,S3,S4	PFMULS S6,S6,F11
10	PSF S1, -64(R2)			PFMULS S7,S7,F10
11			PFSUB S5,S5,S6	PFMULS S8,S8,F11
12	PSF S3, -48(R2)			
13			PFSUB S7,S7,S8	
14	PSF S5, -32(R2)			
15				
16	PSF S7, -16(R2)	BNE R4, Boucle		

Résumé :
Sans SIMD : 10 cycles/itération
Avec SIMD : 10/4 = 2,5 cycles/itération
Avec déroulage, sans SIMD : 16/4 = 4 cycles/itération
Avec déroulage, avec SIMD : 16/ 16 =1 cycles/itération

SIMD IA-32

Soit une image X[128][128] en niveau de gris. On applique sur cette image un seuillage par rapport à la valeur moyenne 128 pour créer une image Y[128][128] : lorsque X[i] [j] <128 alors Y[i][j] = 0 sinon Y[i][j] = 255 .

Q5) Ecrire le code C scalaire correspondant.

Q6) En utilisant les intrinsics du jeu d'instructions SIMD IA-32, écrire une version SIMD pour le seuillage. Quel accélération maximale peut on espérer par rapport à la version scalaire ?

```
/* ------------------------------------------------------------------ */
void seuil (byte **X, int i0, int i1, int j0, int j1, byte **Y)
/* ------------------------------------------------------------------ */
{int i, j;
  for(i=i0; i<=(i1); i++) {
    for(j=j0; j<=(j1)/16; j++) {
            if (X[i][j] <128) Y[i][j]=0 ;
            else Y[i][j]=255 ;}}

/* ------------------------------------------------------------------ */
void seuil_simd(byte **X, int i0, int i1, int j0, int j1, byte **Y)
/* ------------------------------------------------------------------ */
{int i, j;
  __m128i aij, tmp, zero;
  __m128i **XS, **YS;
  XS=X; YS=Y;
 zero = _mm_set1_epi16 (0);
  for(i=i0; i<=(i1); i++) {
    for(j=j0; j<=(j1)/16; j++) {
            aij=ld16(XS[i][j]); tmp=cmpgt8 (zero, aij);
            st16(YS[i][j], tmp); }}}

Accélération maximale : 16
```

Pipeline logiciel avec TMS 320C62x

Soit la boucle de pipeline logiciel suivante, qui travaille sur un tableau T[N] :

```
$C$L2:    ; PIPED LOOP KERNEL

           CMPGT   .L2X    A3,B4,B0        ; B0 = 1 si A3>B4 ;=0 sinon
||         CMPLT   .L1     A3,A5,A1        ; A1 = 1 si A3<A5 ;=0 sinon
|| [ A2]   B       .S2     $C$L2             ;

   [ B1]   SUB     .S2     B1,1,B1           ;
|| [ A1]   MV      .S1     A3,A5             ;
|| [ B0]   MV      .L2X    A3,B4             ;
|| [ A2]   SUB     .L1     A2,1,A2           ;
|| [ B1]   LDH     .D1T1   *A4++,A3          ; A4 pointe sur T[N]
```

Ce pipeline logiciel correspond à un programme C dont le début est

```
Type T[N], L, H ;
L = T[0] ;
H = T[0] ;
for (i=1 ; i<N ;i++){
      Corps de boucle ; }
```

Q 7) Quel est le type des variables T[N], L et H. Donner le corps de boucle du programme C original. Que fait le programme.

Shorts
for (i=1 ; i<N, i++)}
if (T[i] < L) L = T[i];
if (T[i] > H) H = T[i];
}

Q 8) Quel est le nombre de cycles par itération du pipeline logiciel ?

2 cycles

SIMD sur processeur NIOS.

Le code VHDL ci-dessous implante un opérateur pour une instruction spéciale ajoutée au jeu d'instructions NIOS II

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
--use IEEE.std_logic_signed.all;
library work;
--use work.float_pkg.all;

entity X is
            port
      (
            dataa       :     in    std_logic_vector(31 downto 0);
            datab       :     in    std_logic_vector(31 downto 0);
            result      :     out   std_logic_vector(31 downto 0)
                                                                  );
end X ;

-------------------------------------------------------
--      Opération à trouver --
-------------------------------------------------------
architecture comp of X  is
```

```
signal t1p,t2p,t3p, t4p,t1m, t2m,t3m,t4m, t1,t2,t3,t4 :
std_logic_vector (7 downto 0);
signal t5,t6 : std_logic_vector (8 downto 0);
signal t7 :std_logic_vector (9 downto 0);
begin
t1p <= (dataa(7 downto 0)- datab (7 downto 0))
      when (dataa(7 downto 0)> datab(7 downto 0)) else "00000000";
t2p <= (dataa(15 downto 8)- datab(15 downto 8))
      when (dataa(15 downto 8)> datab(15 downto 8)) else "00000000";
t3p <= (dataa(23 downto 16)- datab(23 downto 16))
      when (dataa(23 downto 16)> datab(23 downto 16)) else "00000000";
t4p <= (dataa(31 downto 24)- datab(31 downto 24))
      when (dataa(31 downto 24)> datab(31 downto 24)) else "00000000";
t1m <= (datab(7 downto 0)- dataa(7 downto 0))
      when (dataa(7 downto 0)< datab(7 downto 0)) else "00000000";
t2m <= (datab(15 downto 8)- dataa(15 downto 8))
      when (dataa(15 downto 8)< datab(15 downto 8)) else "00000000";
t3m <= (datab(23 downto 16)- dataa(23 downto 16))
      when (dataa(23 downto 16)< datab(23 downto 16)) else "00000000";
t4m <= (datab(31 downto 24)- dataa(31 downto 24))
      when (dataa(31 downto 24)< datab(31 downto 24)) else "00000000";
t1 <= t1p OR t1m;
t2 <= t2p OR t2m;
t3 <= t3p OR t3m;
t4 <= t4p OR t4m;
t5<=("0"&t1)+("0"&t2);
t6<=("0"&t3)+("0"&t4);
t7<=("0"&t5)+("0"&t6);
result <= "00000000000000000000"&t7;
end comp; -- end of architecture
```

Figure 1 : code VHDL de l'opération X

Q 9) Quelle est l'opération sur les registres dataa et datab effectuée par l'opérateur défini par le code VHDL de la figure 1 ?

Somme des valeurs absolues des différences des octets non signés des mots a et b. Le résultat est sur un mot de 32 bits

$S = |a_{31-24} - b_{31-24}| + |a_{23-16} - b_{23-16}| + |a_{15-8} - b_{15-8}| + |a_{7-0} - b_{7-0}|$

Soit l'extrait suivant d'un programme C (figure 2).
On appelle IX (a,b) l'instruction ajoutée processeur NIOS utilisant l'opération X définie par le code VHDL de la figure 1

Q 10) Donner une version du code à optimiser utilisant l'instruction IX. Quelle accélération maximale peut on obtenir sur ce cœur de boucle ?

```
#define N 128;

typedef union q832 {
      unsigned char char_word[N][N];
      int int_word[N][N/4];
} q832 ;

q832  X, Y;
int i, di;j;dj, k,l, s;

int main(){

// Début du main
// Début du code à optimiser
 s = 0;
```

```
   for(k=0; k<16; k++) {
       for(l=0; l<16; l++) {
        s += abs(X->char_word[i+di+k][j+dj+l]-Y->char_word[i+k][j+l]);
}}
// Fin du code à optimiser
// fin du main
}
```

Figure 2 : extrait de programme C à optimiser.

Code optimise

```
int temp;
s = 0;
for(k=0; k<16; k++) {
for(l=0; l<4; l++) {
      tmp =IX (X->int_word[i+di+k][j+dj+l],Y->int_word[i+k][j+l]);
      s+= tmp; }
```

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
- 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. Les chargements ne peuvent pas passer devant les rangements en attente.

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
FSUB	FSUB Fd, Fa, Fb	FA	Fd ← Fa - Fb
FMAX	FMAX Fd, Fa, Fb	FA	Fd ← Max(Fa, Fb)
FMIN	FMIN Fd, Fa, Fb	FA	Fd ← Min(Fa, Fb)
FMUL	FMUL Fd, Fa, Fb	FM	Fd ←Fa x Fb
FDIV	FDIV Fd, Fa, Fb	FA	Fd ← Fa/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 (données)	FADD/ FMAX/ FMIN	FMUL	FDIV	FSQRT
UAL		1	2				
LF/ST (adresses)		1	3				
SF (données)		1	2	3	3	20 (NP)	20 (NP)
Opération flottante			2	3	3	20 (NP)	20 (NP)

Table 2 : latences

Le processeur a également 32 registres de 128 bits S0 à S7 pouvant contenir chacun 4 flottants simple précision et les instructions SIMD données dans la Table 3. Cette table donne également les latences des instructions SIMD et le pipeline utilisé dans le cas superscalaire.

PLF	PLF Si, dép.(Ra)	2	E0	Charge quatre flottants simple précision à partir de l'adresse (Ra + dépl.16 bits avec ES).
PSF	PSF Si, dép.(Ra)	2	E0	Range en mémoire à partir de l'adresse (Ra + dépl.16 bits avec ES) quatre flottants simple précision
PFADD	PFADD Sd,Sa, Sb	3	FA	Sd ← Sa + Sb sur quatre « floats »
PFSUB	PFSUB Sd,Sa, Sb	3	FA	Sd ← Sa – Sb sur quatre « floats »
PFMUL	PFMUL Sd, Sa, Sb	3	FM	Sd ← Sa x Sb sur quatre « floats »
PFMULS	PMULS Sd, Sa, Fb	3	FM	SF ← Sa x Fb (chaque élément de Sa est multiplié par Fb et rangé dans l'élément correspondant de SF) sur quatre « floats »

Table 3 : Instructions SIMD

ANNEXE 2 : Instructions SIMD IA-32 utilisables

```
#define byte unsigned char
```

#define ld16(a)	_mm_load_si128(&a)	chargement aligné
#define st16(a, b)	_mm_store_si128(&a, b)	rangement aligné
#define or(a,b)	_mm_or_si128(a,b)	ou logique
#define xor(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 b2hl(a,b)	_mm_unpacklo_epi8 (a,b)	4 octets bas entrelacés vers 4 shorts
#define b2hh(a,b)	_mm_unpackhi_epi8 (a,b)	4 octets haut entrelacés vers 4 shorts
#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