UNIVERSITÉ PARIS SACLAY
Master E3A – SETI

T1 Architecture avancée
TD1 Processeurs scalaires, superscalaires et VLIW

1 Prédiction de branchement

Un branchement a le comportement suivant, où N signifie non pris, P signifie pris et P*k indique une suite de k branchements pris.
PPNP*kNP*kNP*kNP*kNP*k...

1.1 Après la phase d'initialisation, quelle est le pourcentage de bonnes prédictions ?
a) avec un prédicteur 1 bit
b) avec un prédicteur 2 bits.
On considérera les cas où $k = 1$, $k = 3$, $k = 5$.

Rép. : a) 1 bit :

— *k=1*
branchement : P P N P N P N P
prédicteur : N P P N P N P N

bonnes prédictions : 0%

— *k=3*
branchement : P P N P P P N P P P N P P P
prédicteur : N P P N P P P N P P P N P P

bonnes prédictions : 1/2

— *k=5*
branchement : P P N P P P P P N P P P P P N P P P P P
prédicteur : N P P N P P P P P N P P P P P N P P P P

bonnes prédictions : 2/3

a) 2 bits (FP=**P**, fp=p, fnp=n, FNP=**N**) *:*

— *k=1 (prédicteur intialisé à FNP)*
branchement : P P N P N P N P
prédicteur : **N** n p n p n p n

bonnes prédictions : 0%

— *k=1(prédicteur intialisé à fnp — comportement similaire pour fp et FP)*
branchement : P P N P N P N P
prédicteur : n p **P** p **P** p **P** p

bonnes prédictions : 1/2

— *k=3*
branchement : P P N P P P N P P P N P P P
prédicteur : n p **P** p **P** **P** **P** p **P** **P** **P** p **P** **P**

bonnes prédictions : 3/4

— *k=5*
branchement : P P N P P P P P N P P P P P N P P P P P
prédicteur : n p **P** p **P** **P** **P** **P** **P** p **P** **P** **P** **P** **P** p **P** **P** **P** **P**

bonnes prédictions : 5/6

1.2 Même question avec un prédicteur un bit et un bit d'historique (initialisé à N).

Rép. :

On suppose l'historique et les prédicteurs N et P intialisés à N*.*
A chaque étape, l'historique détermine :

— *lequel des deux prédicteurs N ou P sera mis à jour pour l'étape suivante en fonction du comportement du branchement.*
Un prédicteur non mis à jour sera affiché en grisé et il gardera son état précédent.

— *si on utilise le prédicteur N ou le prédicteur P pour faire la prédiction courante (encadré)*

Prédiction parfaite pour $k = 1$ (l'historique correspond à la période).
Pas d'amélioration dans les autres cas (il faudrait un historique plus long)
a) 1 bit :

— *k=1*

branchement :	[P]	[P]	[N]	[P]	[N]	[P]	[N]	[P]
historique :	N	P	P	N	P	N	P	N
prédicteur N :	[N]	P	P	[P]	P	[P]	P	[P]
prédicteur P :	N	[N]	[P]	N	[N]	N	[N]	N

bonnes prédictions : 100%

— *k=3*

branchement :	[P]	[P]	[N]	[P]	[P]	[P]	[N]	[P]	[P]	[P]	[N]	[P]	[P]	[P]
historique :	N	P	P	N	P	P	P	N	P	P	P	N	P	P
prédicteur N :	[N]	P	P	[P]	P	P	P	[P]	P	P	P	[P]	P	P
prédicteur P :	N	[N]	[P]	N	[N]	[P]	[P]	N	[N]	[P]	[P]	N	[N]	[P]

bonnes prédictions : 1/2

— *k=5*

branchement :	[P]	[P]	[N]	[P]	[P]	[P]	[P]	[P]	[N]	[P]	[P]	[P]	[P]	[P]	[N]	[P]	[P]	[P]	[P]	[P]
historique :	N	P	P	N	P	P	P	P	P	N	P	P	P	P	P	N	P	P	P	P
prédicteur N :	[N]	P	P	[P]	P	P	P	P	P	[P]	P	P	P	P	P	[P]	P	P	P	P
prédicteur P :	N	[N]	[P]	N	[N]	[P]	[P]	[P]	[P]	N	[N]	[P]	[P]	[P]	[P]	N	[N]	[P]	[P]	[P]

bonnes prédictions : 2/3

2 Processeurs scalaires et superscalaires : exécution de boucles

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 court-circuits 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.

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 pour l'addition et FM le pipeline flottant pour la multiplication.
Les instructions peuvent être exécutées simultanément si elles utilisent chacune un pipeline séparé.

L'ordonnancement est statique. Les chargements ne peuvent pas passer devant les rangements en attente.

La colonne latence 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 $i + 1$.

Soit le programme C suivant (SAXPY) où `x` et `y` sont des vecteurs et `a` est un scalaire de nombres flottants simple précision.

```
float x[1000], y[1000], a;
main ()
{
  int i ;
  for (i=0; i<1000 ;i++)
    y[i] = y[i] + a*x[i] ;
}
```

On utilisera les registres suivants : `r1` pointe sur `x[i]`, `r2` pointe sur `y[i]`. `r3` a été initialisé à 1000. `f0` contient `a`. `f1` et `f2` recevront respectivement `x[i]` et `y[i]`.

2.1 On considére dans un premier temps une version scalaire du processeur et des caches parfaits (aucun cycle d'attente mémoire).
Traduire le corps du programme en assembleur.
Faire apparaître les cycles d'attentes dues aux dépendances.
Optimiser par réordonnancement ce code pour limiter les cycles d'attente. Donner l'exécution cycle par cycle de la boucle optimisée et le nombre de cycles par itération.

Rép. : *A noter :*

- *les accès mémoire sont mis en oeuvre avec un registre contenant l'adresse de l'élément courant du tableau (ce qui correspond à un pointeur en C). Le pointeur est incrémenté en fin d'itération.*
- *les itérations sont comptées par un registre initialisé à* `N` *et décrémenté à chaque cycle*
- *les boucles sont faites avec un seul branchement (boucles* `do ... while()`*)*

Le programme assembleur correspond très précisément au code C suivant :

```
main ()
{
  float x[1000], y[1000], a;
  float *px=&x[0], *py=&y[0] ;
  int i=1000 ;
  do {
    *py  = *py + a * *px ;
    px++;py++;i--
  } while (i != 0) ;
```

Version non optimisée.

JEU D'INSTRUCTIONS (extrait)				
Mnémo	Exemple	Latence	Pipeline	Effet
`lf`	`lf fi, dép.(ra)`	2	E0 ou E1	`fi` ← `mem[ra + dépl.` (16 bits avec ext. sig.)`]`
`sf`	`sf fi, dép.(ra)`	1	E0	`fi` → `mem[ra + dépl.` (16 bits avec ext. sig.)`]`
`add`	`add rd,ra, rb`	1	E0 ou E1	`rd` ← `ra + rb`
`addi`	`addi rd, ra, imm`	1	E0 ou E1	`rd` ← `ra + imm` (16 bits avec ext. sig.)
`sub`	`sub rd, ra, rb`	1	E0 ou E1	`rd` ← `ra - rb`
`fadd`	`fadd fd, fa, fb`	4	FA	`fd` ← `fa + fb`
`fmul`	`fmul fd, fa, fb`	4	FM	`fd` ← `fa × fb`
`beq`	`beq ri, dépl`	1	E1	si `ri=0` alors `cp` ← `cp+4 + dépl.`
`bne`	`bne ri, dépl`	1	E1	si `ri≠0` alors `cp` ← `cp+4 + dépl.`

TABLE 1 – Instructions disponibles

```
Boucle : lf     f1, (r1)   ; f1 <- *x=x[i]
         lf     f2, (r2)   ; f1 <- *y=y[i]
         fmul   f1, f1, f0 ; f1 <- a*x[i]
         fadd   f2, f2, f1 ; f2 <- y[i]+a*x[i]
         sf     f2, (r2)   ; *y <- f2
         addi   r1, r1, 4  ; x++
         addi   r2, r2, 4  ; y++
         addi   r3, r3, -1 ; i--
         bne    r3, r0, Boucle
```

Prise en compte des latences des opérateurs

```
Boucle : lf     f1, (r1)
         lf     f2, (r2)
         fmul   f1, f1, f0
         (nop) x 3
         fadd   f2, f2, f1
         (nop) x 3
         sf     f2, (r2)
         addi   r1, r1, 4
         addi   r2, r2, 4
         addi   r3, r3, -1
         bne    r3, r0, Boucle
```

Optimisation par réordonnancement

```
Boucle : lf     f1, (r1)
         lf     f2, (r2)
         fmul   f1, f1, f0
         addi   r1, r1, 4
         addi   r2, r2, 4
         addi   r3, r3, -1
         fadd   f2, f2, f1
         (nop) x 3
         sf     f2, -4(r2) ; car r2 incrémenté ligne 5
         bne    r3, r0, Boucle
```

12 cy/itération

2.2 Donner la version déroulée (4 itérations par corps de boucle) avec la version scalaire du processeur en supposant des caches parfaits et déterminer le nombre de cycles par itération de la boucle.

Rép. : Version déroulée ×4 optimisée

```
Boucle : lf     f1, 0(r1)
         lf     f3, 4(r1)
         lf     f5, 8(r1)
         lf     f7, 12(r1)
         lf     f2, 0(r2)
         lf     f4, 4(r2)
         lf     f6, 8(r2)
         lf     f8, 12(r2)
         fmul   f1, f1, f0
         fmul   f3, f3, f0
         fmul   f5, f5, f0
         fmul   f7, f7, f0
         fadd   f2, f2, f1
         fadd   f4, f4, f3
         fadd   f6, f6, f5
         fadd   f8, f8, f7
         addi   r1, r1, 16
         addi   r2, r2, 16
         addi   r3, r3, -4
         sf     f2, -16(r2)
         sf     f4, -12(r2)
```

```
22          sf      f6, -8(r2)
23          sf      f8, -4(r2)
24          bne     r3, r0, Boucle
```

24 cy/4 itérations

2.3 Donner l'exécution cycle par cycle de la boucle optimisée non déroulée pour la version superscalaire en considérant des caches parfaits, ainsi que le nombre de cycles par itération de la boucle.

Rép. :

cycles	E0	E1	FM	FA
1 Boucle :	**lf** f1, (r1)			
2	**lf** f2, 0(r2)			
3	**addi** r1, r1, 4	**addi** r2, r2, 4	**fmul** f1, f1, f0	
4	**addi** r3, r3, -1			
5				
6				
7				**fadd** f2, f2, f1
8				
9				
10				
11	**sf** f2, -4(r2)	**bne** r3, r0, Boucle		

11 cy/ itération

2.4 Donner l'exécution cycle par cycle de version déroulée (4 itérations par corps de boucle) dans la version superscalaire du processeur en supposant des caches parfaits, et le nombre de cycles par itération de la boucle.

Rép. :

cycles	E0	E1	FM	FA
1 Boucle :	**lf** f1, (r1)	**lf** f3, 4(r1)		
2	**lf** f5, 8(r1)	**lf** f7, 12(r1)		
3	**lf** f2, 0(r2)	**lf** f4, 4(r2)	**fmul** f1, f1, f0	
4	**lf** f6, 8(r2)	**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, -4		**fmul** f7, f7, f0	
7				**fadd** f2, f1, f2
8				**fadd** f4, f4, f3
9				**fadd** f6, f6, f5
10				**fadd** f8, f8, f7
11	**sf** f2, -16(r2)			
12	**sf** f4, -12(r2)			
13	**sf** f6, -8(r2)			
14	**sf** f8, -4(r2)	**bne** r3, r0, Boucle		

14 cy/ 4 itérations

3 Processeurs VLIW (optionnel)

L'annexe donne le schéma fonctionnel du processeur VLIW TMS320C62 de Texas Instruments, avec les instructions réparties par unité fonctionnelle et les latences des instructions. Dans cet exercice, on utilisera le TMS320C67, qui a une architecture similaire, les mêmes instructions entières et possède en plus des instructions flottantes dont la répartition dans les unités fonctionnelles et les latences sont également fournies en annexe.

3.1 Examiner l'exécution du code ci-dessous, dans lequel on suppose que `sum` est un registre initialisé à 0. Que fait cette fonction ? En étudiant les latences, proposer une méthode pour l'optimiser.

```
LOOP:           ADDSP   x,sum,sum
  ||            LDW     *xptr++,x
  || [cond]     B       cond
  || [cond]     SUB     cond,1,cond
```

Rép. : Cette fonction réalise la somme des éléments d'un vecteur de `floats`.

```
float sumvec(float a[])
{
  int i;
  float sum;
  sum=0.0f;
  for(i=0; i<N; i++)
    sum += a[i];
  return sum;
}
```

Pour que cela fonctionne, il faut un pipeline logiciel. Nous chargerons l'élément i, tout en accumulant un élément précédemment chargé pour prendre en compte la latence du `LDW`. Les données étant flottantes, la latence de l'addition est 5. Il faut donc insérer 4 `NOP` entre deux accumulations. Noter que la latence du `LDW` est identique à celle du `ADDSP`. Si ce n'était pas le cas, il faudrait que le nombre de `NOP` soit suffisant pour couvrir la plus grande latence.

```
//prologue
aa=a[0];
//nop x4
for(i=1; i<N; i++){
  sum += aa; /*accumule a[i-1]*/ aa = a[i];
  //nop
  //nop
  //nop
  //nop
}
//epilogue
sum +=aa;
```

Pour optimiser et supprimer les `NOP`, la seule solution est d'effectuer un déroulage de boucle avec plusieurs accumulateurs.

```
float sum0, sum1, sum2, sum3, sum4,
      aa0, aa1, aa2, aa3, aa4 ;
sum0=sum1=sum2=sum3=sum4=0.0f;
//prologue
aa0=a[0];
aa1=a[1];
aa2=a[2];
aa3=a[3];
aa4=a[4];
for(i=5; i<N; i+=5){
  sum0 += aa0; aa0 = a[i+0]; // en parallèle
  sum1 += aa1; aa1 = a[i+1];
  sum2 += aa2; aa2 = a[i+2];
  sum3 += aa3; aa3 = a[i+3];
  sum4 += aa4; aa4 = a[i+4];
}
//epilogue
sum0 += aa0;
sum1 += aa1;
sum2 += aa2;
sum3 += aa3;
sum4 += aa4;
return sum0+sum1+sum2+sum3+sum4;
```

On constate toutefois une mauvaise utilisation des unités de calcul.

```
LOP:        ADDSP    .S1   x,sum,sum
  ||        LDW      .D1   *xptr++,x
  || [cond] B        .L1   cond
  || [cond] SUB      .L2   cond,1,cond
```

Un seul des additionneurs flottants du processeur est vraiment utilisé. On peut facilement faire en

parallèle un calcul et un chargement additionnels avec les unités S2 et D2.
Une solution pour remédier à ce problème est d'utiliser deux accumulations simultanées à chaque instruction. Ceci permettra d'utiliser les deux ALUS.
Pour cela, il faut effectuer un déroulage de boucle d'ordre 10. (pour simplifier, nous supposons N multiple de 10)

```
float sumvec(float a[])
{
  int i;
  float sum0, sum2, sum2, sum3, sum4, ..., sum8, sum9,
        aa0, aa1, aa2, aa3, aa4, ..., aa8, aa9;
  sum1=sum2=sum3=sum4=...=0.0f;
  //prologue
  aa0 = a[0];  aa1 = aa[1]; // en parallèle
  aa2 = a[2];  aa3 = aa[3]; // en parallèle
  ...
  aa8 = a[8];  aa9 = a[9]; // en parallèle
  for(i=10; i<N/10; i+=10){
    sum0 += aa0; aa0 = a[i];      sum1 += aa1; aa1 = a[i+1]; // en parallèle
    sum2 += aa2; aa2 = a[i+2];    sum3 += aa3; aa3 = a[i+3]; // en parallèle
    ...
  }
  //epilogue et accumulation finale
  return aa0+aa1+...+aa9+sum0+sum1+...+sum9;
}
```

3.2 En utilisant le pipeline logiciel, écrire le code assembleur pour la fonction `dotp` « flottante ». Quel est le temps d'exécution de la fonction ?

```
float dotp(float a[], float b[])
{
  int i;
  float sum;
  sum=0.0f;
  for(i=0; i<100; i++)
    sum += a[i] * b[i];
  return sum;
}
```

Rép. : *Code scalaire*

```
#  A1 : @a[i], B1 : @b[i], A5 cptr boucle, A7 sum
Boucle:
        LDW   .D1 *A1++,A0  ; A0=a[i]
        LDW   .D2 *B1++,B0  ; B0=b[i]
        NOP 4
        MPYSP .M1 A0, B0, A3; A3=a[i]*b[i]
        NOP 4
        ADDSP .L1 A3, A7, A7; sum += A3
        SUB   .S1 A5,1,A5   ; cptr--
   [A5] B     .S1 Boucle
```

La grande différence par rapport à la fonction entière vue en cours est la latence des opérations flottantes.
La latence des multiplications induit un retard, mais n'est pas génante coté performances, car on peut faire toutes les multiplications indépendamment.
Par contre, comme l'addition fait une accumulation, il faut attendre le résultat précédent (5 cycles de latence) avant de commencer une nouvelle addition.
Le seul moyen pour accélérer est de dérouler les boucles et d'utiliser plusieurs accumulateurs.
En faisant cela, on arrive à une solution proche de la question précédente.

3.3 En utilisant le pipeline logiciel, écrire le code assembleur pour la fonction IIR. Quel est le temps d'exécution de la fonction ?

```
void iir(short x[], short y[], short c1, short c2, short c3)
{
  int i;

  for(i=0; i<100; i++)
    y[i+1]=(c1*x[i] + c2*x[i+1] + c3*y[i]) >> 15 ;

}
```

4 Annexes

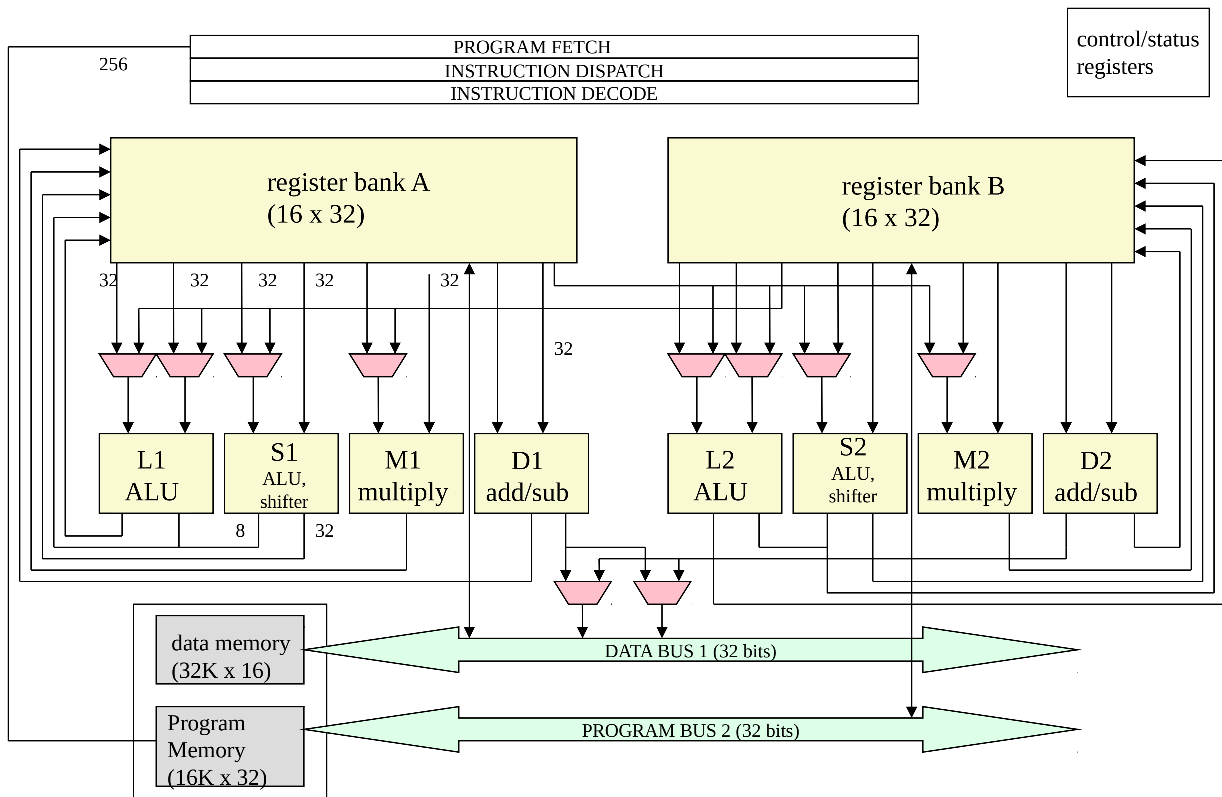

FIGURE 1 – Schéma fonctionnel du TMS320C62

Type instruction	Latence
NOP (no operation)	1
Store	1
Instr. Simple	1
Multiplication (16x16)	2
Load	5
Branchement	6
Addition SP	5
Multiplication SP	5

TABLE 2 – Latence des instructions

.L Unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset)‡
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset)‡
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset)‡
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SSHL	LDB	SUBAB
CMPGT	MPYHU	B IRP†	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP†	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB (15-bit offset)‡	
MV	MPYHULS	EXTU		LDBU (15-bit offset)‡	
NEG	MPYHSLU	MV		LDH (15-bit offset)‡	
NORM	MPYLH	MVC†		LDHU (15-bit offset)‡	
NOT	MPYLHU	MVK		LDW (15-bit offset)‡	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

† S2 only
‡ D2 only

.L Unit	.M Unit	.S Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
DPTRUNC		CMPGTDP	
INTDP		CMPGTSP	
INTDPU		CMPLTDP	
INTSP		CMPLTSP	
INTSPU		RCPDP	
SPINT		RCPSP	
SPTRUNC		RSQRDP	
SUBDP		RSQRSP	
SUBSP		SPDP	

Table 3 – Répartition des instructions entières et flottantes dans les unités fonctionnelles