Université Paris Sud 2013-2014
Informatique L313

ARCHITECTURE DES ORDINATEURS
Corrigé Examen Décembre 2013
3 H – Tous documents autorisés – Parties indépendantes

OPTIMISATIONS DE PROGRAMME

Cette partie utilise le sous-ensemble du jeu d'instructions MIPS donné en annexe.

On suppose une version pipelinée du processeur utilisant les instructions MIPS.
La latence des instructions est donnée dans la deuxième colonne de la Table 2. On rappelle qu'une latence de **n** signifie que si une instruction **I** démarre au cycle **c**, une instruction qui utilise le résultat de I ne peut démarrer qu'au cycle c+n. (une latence de 1 signifie qu'elle peut démarrer au cycle suivant).
La Table 1 présente un programme C et le programme assembleur MIPS correspondant. Les tableaux A, B, C, D, E sont rangés successivement à partir de l'adresse 0x10000000, qui est contenue au démarrage dans le registre R3.

Question 1) Quel est le temps d'exécution (en cycles par itération) de la boucle de la table 1. Optimiser la boucle sans déroulage et donner le nouveau temps d'exécution.

float A[100], B[100],C[100],
D[100],E[100];
int i ;
for (i=0; i<100; i++)
A[i]=B[i]*C[i] + D[i]*E[i];
	ADDI R5, R3, 400
	Boucle :LWC1 F1,400(R3)
	LWC1 F2,800 (R3)
	LWC1 F3,1200(R3)
	LWC1 F4,1600(R3)
	MUL.S F1,F1,F2
	MUL.S F3,F3,F4
	ADD.S F1,F1,F3
	SWC1 F1,(R3)
	ADDI R3,R3,4
	BNE R3,R5, Boucle

Table 1 : Programme C et programme assembleur

	Avant optimisation	Après optimisation	Déroulage d'ordre 2 (V1)	Déroulage de boucle 2 (V2)
1 Boucle	LWC1 F1,400(R3)	LWC1 F1,400(R3)	LWC1 F1,400(R3)	LWC1 F1,400(R3)
2	LWC1 F2,800 (R3)	LWC1 F2,800 (R3)	LWC1 F5,404(R3)	LWC1 F2,800 (R3)
3	LWC1 F3,1200(R3)	LWC1 F3,1200(R3)	LWC1 F2,800 (R3)	LWC1 F3,1200(R3)
4	LWC1 F4,1600(R3)	LWC1 F4,1600(R3)	LWC1 F6,804(R3)	LWC1 F4,1600(R3)
5	MUL.S F1,F1,F2	MUL.S F1,F1,F2	LWC1 F3,1200(R3)	MUL.S F1,F1,F2

6	MUL.S F3,F3,F4	MUL.S F3,F3,F4	LWC1 F7,1204 (R3)	MUL.S F3,F3,F4
7		ADDI R3,R3,4	LWC1 F4,1600(R3)	LWC1 F5,405(R3)
8			LWC1 F8,1604 (R3)	LWC1 F6,804(R3)
9			MUL.S F1,F1,F2	LWC1 F7,1204 (R3)
10	ADD.S F1,F1,F3	ADD.S F1,F1,F3	MUL.S F5,F5,F6	LWC1 F8,1604 (R3)
11			MUL.S F3,F3,F4	MUL.S F5,F5,F6
12			MUL.S F7,F7,F8	MUL.S F7,F7,F8
13			ADDI R3,R3,8	ADD.S F1,F1,F3
14	SWC1 F1,(R3)	SWC1 F1,-4(R3)		ADDI R3,R3,8
15	ADDI R3,R3,4	BNE R3,R5, Boucle	ADD.S F1,F1,F3	
16	BNE R3,R5, Boucle		ADD.S F5,F5,F7	ADD.S F5,F5,F7
17				SWC1 F1,(R3)
18				
19			SWC1 F1,-8(R3)	
20			SWC1 F5,-4(R3)	SWC1 F5,-4(R3)
21			BNE R3,R5, Boucle	BNE R3,R5, Boucle

Avant optimisation : 16 cycles
Après optimization : 15 cycles

Question 2) Donner le code assembleur d'une version optimisée après déroulage d'ordre 2.

Les deux versions donnent 10,5 cycles/itération

Question 3) Quel serait le temps d'exécution (en nombre de cycles de la boucle initiale) avec un déroulage de boucle d'ordre 4 et optimisation?

Dans ce cas, il n'y a plus de cycles de suspension.
Il y a

- 16 LWC1
- 8 MUL.S
- 4 ADD.S
- 4 SWC1
- ADDI + BNE

Soit 34 instructions (cycles) pour 4 itérations = 8,5 cycles/itération

CACHES (1ère partie)

Soit les programmes suivants :

Programme P1 :

```
float X[128], Y[128] ; S ;
S=0.0 ;
for (i=0 ; i<128 ; i++)
      S+=X[i]*Y[i]; // produit scalaire
```

Programme P2

Dans la version assembleur, R1 contient initialement l'adresse de X[0] et R2 contient l'adresse de Y[0].

```
float X[128], Y[128] ; S ;
S0=0.0 ; S1=0.0,;S2=0.0;
S3=0.0
for (i=0 ; i<128 ; i+=4){
      S0+=X[i]*Y[i];
      S1+=X[i+1]*Y[i+1];
      S2+=X[i+2]*Y[i+2];
      S3+=X[i+3]*Y[i+3];}
S=S0+S1+S2+S3;
```

```
          SUB.S F0,F0,F0
          SUB.S F1,F1,F1
          SUB.S F2,F2,F2
          SUB.S F3,F3,3
          ADDI R3,R1,512
Boucle :  LWC1 F5,0(R1)
          LWC1 F6,0(R2)
          MUL.S F5,F5,F6
          ADD.S F0,F0,F5
          LWC1 F7,4(R1)
          LWC1 F8,4(R2)
          MUL.S F7,F7,F8
          ADD.S F1,F1,F7
          LWC1 F9,8(R1)
          LWC1 F10,8(R2)
          MUL.S F9,F9,F10
          ADD.S F2,F2,F9
          LWC1 F11,12(R1)
          LWC1 F12,12(R2)
          MUL.S F11,F11,F12
          ADD.S F3,F3,F11
          ADDI R1,R1,16
          BNE R1,R3,Boucle
          ADD.S F0,F0,F1
          ADD.S F2,F2,F3
          ADD.S F0,F0,F2
```

On considère un cache de données de 8 Ko, à correspondance directe, avec des lignes de 16 octets.
On suppose que les deux tableaux X[N] et Y[N] de flottants simple précision sont aux adresses :
Adresse de X[0] : 0xA000 0000
Adresse de Y[0] : 0xB000 0000
Les variables scalaires sont en registre.

Question 4) Quel est le nombre de défauts de caches pour exécuter le programme P1 ? (on donnera le nombre total de défauts de caches)

En correspondance directe, il y a 2 défauts de cache par itération (X[0] et Y[0] vont dans la ligne 0.
Nombre total de défauts : 256

Question 5) Optimiser le code assembleur du programme P2 pour minimiser le nombre de défauts de cache. Quel est alors le nombre total de défauts de cache du programme P2 ?

Boucle :	LWC1 F5,0(R1) LWC1 F7,4(R1) LWC1 F9,8(R1) LWC1 F11,12(R1) LWC1 F6,0(R2) LWC1 F8,4(R2) LWC1 F10,8(R2) LWC1 F12,12(R2) MUL.S F5,F5,F6 MUL.S F7,F7,F8 MUL.S F9,F9,F10 MUL.S F11,F11,F12 ADD.S F0,F0,F5 ADD.S F1,F1,F7 ADD.S F2,F2,F9 ADD.S F3,F3,F11 ADDI R1,R1,16 BNE R1,R3,Boucle

Les registres sont chargés selon les lignes de cache. Il y a maintenant 1 défaut/4 itérations pour X et 1 défaut/4 itérations pour Y.
Total : 64 défauts

Question 6) Proposer une modification matérielle du cache pour que le programme P1 ait le même nombre de défauts de cache que le programme P2 après optimisation du code assembleur.
Utiliser un cache associatif 2 voies

CACHES (2ème partie)

Soit le programme C suivant qui calcule un vecteur Z[N] dont les composantes sont la somme des colonnes d'une matrice Y[N][N].

```
#define N 100
float Z[N], Y[N][N];

void main(){
int i,j;
for (j=0;j<N;j++) {
      S=0.0f;
      for (i=0;i<N;i++)
            S+=Y[i][j];
      Z[j]=S;}
}
```

Question 7) Donner une nouvelle version du programme C avec déroulage de boucle qui permette de réduire le nombre de défauts de cache (on peut supposer que l'on utilise un cache avec des lignes de 16 octets, à correspondance directe et écriture allouée, provoquant des défauts de cache en écriture).

Il faut un déroulage de boucle d'ordre 4 de la boucle externe.

```
#define N 100
float Z[N], Y[N][N];

void main(){
int i,j;
for (j=0;j<N;j+=4) {
      S=0.0f; S1=0.0f; S2=0.0f; S3=0.0f;
      for (i=0;i<N;i++){
            S+=Y[i][j];
            S1+=Y[i][j+1];
            S2+=Y[i][j+2];
            S3+=Y[i][j+3];}
      Z[j]=S;
      Z[j+1]=S1;
      Z[j+2]=S2;
      Z[j+3]=S3;
      }
}
```

TEMPS D'EXECUTION

Soit le programme C suivant :

```
float X[N][N], Y[N][N], Z= 4.5 ;
for (j=0 ; j<N ;j++)
      for (i=0; i<N;i++)
      Y[i][j] = X[i][j]/Z;
```

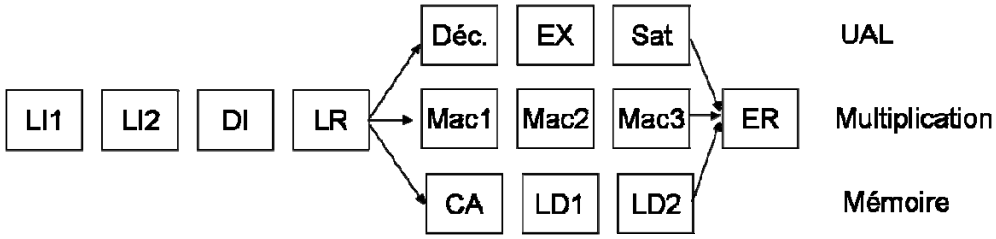
Question 8) Donner une nouvelle version (sans utiliser d'instructions SIMD) du programme C permettant de réduire le temps d'exécution (sans tenir compte d'éventuelles optimisation du compilateur)

Deux modifications
Remplacer la division par une multiplication par l'inverse de la constante
Permuter les boucles.

```
float X[N][N], Y[N][N], Z1= 1/4.5 ;
for (i=0; i<N;i++)
      for (j=0 ; j<N ;j++)
      Y[i][j] = X[i][j]/Z;
```

PIPELINE

La figure ci-dessous donne le pipeline du processeur ARM11 (jeu d'instructions ARM).



La signification des étapes du pipeline sont

- LI1 et LI2 : lecture de l'instruction
- DI : décodage de l'instruction
- LR : lecture des registres
- Déc : décalage ou rotation du 2ème opérande
- EX : exécution
- Sat : saturation pour les instructions de l'arithmétique saturée
- Mac 1, 2 et 3 : étapes de la multiplication
- CA : calcul d'adresse
- LD1 et LD2 : accès cache de données
- ER : écriture du résultat.

Question 9) Donner les latences des instructions suivantes, en supposant que tous les court-circuits nécessaires sont implantés.

	Instruction	Producteur	Consommateur
a)	ADD	ADD **R1**,R2,R3	SUB R5,R6,**R1**
b)	ADD	ADD **R1**,R2,R3	STR **R1**,[R4 +4] !
c)	ADD	ADD **R1**,R2,R3	STR R4,[**R1** +4] !
e)	ADD	ADD **R1**,R2,R3	MUL R5,R6,**R1**
f)	LDR	LDR **R1,** [R2], #4	SUB R5,R6,**R1**
d)	MUL	MUL **R1**,R2,R3	SUB R5,R6,**R1**

ADD **R1**,R2,R3						SUB R5,R6,**R1**			
LI1	LI2	DI	LR	DEC	**EX**	SAT	ER		
	LI1	LI2	DI	LR	DEC	**EX**	SAT	ER	

1 cycle

ADD **R1**,R2,R3						STR **R1**,[R4 +4] !			
LI1	LI2	DI	LR	DEC	**EX**	SAT	ER		
	LI1	LI2	DI	LR	CA	**LD1**	LD2	ER	

1 cycle

ADD **R1**,R2,R3						STR R4,[**R1** +4] !			
LI1	LI2	DI	LR	DEC	**EX**	SAT	ER		
	LI1	LI2	DI	LR		**CA**	LD1	LD2	ER

2 cycles

ADD **R1**,R2,R3			MUL R5,R6,**R1**						
LI1	LI2	DI	LR	DEC	**EX**	SAT	ER		
	LI1	LI2	DI	LR		**M1**	M2	M3	ER

2 cycles

LDR	LDR **R1,** [R2], #4			SUB R5,R6,**R1**					
LI1	LI2	DI	LR	CA	LD1	**LD2**	ER		
	LI1	LI2	DI	LR	DEC		**EX**	SAT	ER

2 cycles

MUL **R1**,R2,R3			SUB R5,R6,**R1**						
LI1	LI2	DI	LR	M1	M2	**M3**	ER		
	LI1	LI2	DI	LR	DEC		**EX**	SAT	ER

2 cycles

PROGRAMMATION ASSEMBLEUR

Soit le programme assembleur ARM ci-dessous

```
      MOV r0,#5
      BL PROC
      LDR r2,=RES
      STR r0,[r2]
      SWI 0x11 @ Stop program execution
PROC: STR r0,[sp],#4
      STR lr,[sp],#4
      SUBS r0,r0,#1
      BNE SUITE
      MOV r0,#1
      SUB sp,sp,#4
      B FIN
SUITE: BL PROC
FIN:  LDR lr,[sp],#-4
      LDR r1,[sp], #-4
      MUL r0,r1,r0
      MOV pc,lr
```

Question 10) Que fait le programme assembleur ARM ? Quel est le contenu de la case mémoire d'adresse RES après exécution du programme ?

La réponse doit tenir en moins de 5 lignes.

Ce programme calcule factorielle 5 de manière récursive. En fin d'exécution, on a 5 ! = 120
Ce programme est identique au programme vu dans le TD-TP 4 (https://www.lri.fr/~de/TD4L3A8.s) à deux instructions près modifiant une somme récursive en un produit récursif.

- MOV r0, #1 (au lieu de 0)
- MUL r0,r1,r0 (au lieu de ADD)

SIMD

Soit le programme ci-dessous utilisant des intrinsics SIMD, comme dans le TP n°10.

```
#define N 500
#define PADD(a,b) _mm_add_ps(a,b)   // addition SIMD 4 floats
#define PMUL(a,b) _mm_mul_ps(a,b)   // multiplication SIMD 4 floats
#define MOVSS(a,b) _mm_store_ss(&a,b) // b31-0 ( b 128 bits)→ float a
#define HADDPS(a,b) _mm_hadd_ps(a,b)   //addition horizontale #define
#define DUP4(a)  _mm_set1_ps(a)     // 4 fois le float a dans 128 bits

typedef union VEC{
      float float_word[N];
      __m128 quad_word[N/4];
} VEC ;
VEC X,Z;
float S;
__m128 SS;

main(){
 int j;
      SS = DUP4(0.0f);
      for (j=0;j<N/4;j++)
            SS=PADD (SS, PMUL (X.quad_word[j],Z.quad_word[j]));
      SS=HADDPS(SS,SS);
      SS=HADDPS(SS,SS);
      MOVSS(S, SS);
}
```

Question 11) Quel calcul effectue le programme sur les vecteurs X[N] et Z[N] ? Que contient la variable S en fin d'exécution si les deux vecteurs X et Z contiennent N fois la valeur 1.0f ?

La réponse doit tenir en moins de 5 lignes.

Le programme SIMD calcule le produit scalaire des vecteurs X[N] et Z[N].
Si les deux vecteurs X et Z contiennent N fois la valeur 1.0f , la variable S contient N=500.0 en fin d'exécution.

Ce programme est très proche du programme produit matrice vecteur SIMD vu dans le TD-TP 10 (programme 3)

ANNEXE MIPS

Les figures donnent la liste des instructions MIPS disponibles.
La signification des abréviations est la suivante :
IMM correspond aux 16 bits de poids faible d'une instruction.
SIMM est une constante sur 32 bits, avec 16 fois le signe de IMM, suivi de IMM (extension de signe). ADBRANCH est l'adresse de branchement, qui est égale à NCP+ SIMM (NCP est l'adresse de l'instruction qui suit le branchement)

Mnémonique	Latence	Syntaxe	Action
ADDI	1	ADDI rt, rs, IMM	rt ← rs + SIMM avec exception sur débordement

BNE	1	BNEQ rs,rt, IMM.	si rs ≠ rt, branche à ADBRANCH
LWC1	2	LWC1 ft, IMM(rs)	rt ← MEM [rs +SIMM]
SWC1	1	SWC1 ft, IMM.(rs)	ft → MEM [rs + SIMM]
ADD.S	4	ADD.S fd, fs,ft	fd ← fs + ft (addition flottante simple précision)
MUL.S	4	MUL.S fd, fs,ft	fd ← fs * ft (multiplication flottante simple précision)
SUB.S	4	SUB.S fd, fs,ft	fd ← fs - ft (soustraction flottante simple précision)
DIV.S	12	DIV.S fd,fs,ft	fd ← fs / ft (division flottante simple précision)

Table 2 : Instructions entières et flottantes MIPS utilisées (NB : les branchements ne sont pas retardés)

ANNEXE ARM

On rappelle que le processeur ARM a 15 registres de 32 bits. Les immédiats sont signés.
R15 est le compteur de programme.

Instruction	Assembleur	Effet
MOV	MOV Ri,Rj	Ri ← Rj
SUBS	SUBS Ri, Rj, #N	Ri ← Rj - N ; positionne Code condition
B	B adresse cible	Branchement inconditionnel
BNE	BNE adresse cible	Branchement si le résultat si le résultat de SUBS différent de zéro
BL	BL adresse cible	Branchement et adresse de retour dans R14
MUL	MUL Ri, Rj, Rk	Ri reçoit Rj * Rk
LDR	LDR Rd, [Rs], #N	Rd reçoit Mem (Rs) et Rs = Rs +N
STR	STR Rd, [Rs], #N	Mem (Rs) reçoit Rd et Rs = Rs +N

Table 3 : Instructions ARM utilisées