TP3 : Simulateur de processeur MIPS

Auteur : Alain Mérigot (Université Paris Saclay)

Compte rendu de TP par équipe de 4 à déposer sur ecampus au plus tard le 5 décembre sous forme de fichier pdf tp3-nom1-nom2-nom3-nom4.pdf)

1 Objectifs du TP

Ce TP a pour objectif d'étudier le comportement d'un processeur pipeline simple sur des programmes écrits en assembleur. Nous utiliserons le simulateur de processeur MIPS eduMIPS http://www.edumips.org/
Installer l'archive dans un répertoire et lancer le simulateur par
`java -jar edumips64-1.2.9.jar`

2 Etude du pipeline sur des programmes élémentaires

Exécuter les programmes `sum.s`, `mul.s`, `sumf.s`, `mulf.s`, `divf.s` et `abs.s`.
Relever le nombre de cycles, le CPI et le nombre de cycles d'attente. On considèrera les cas où il y a ou non des mécanismes d'envoi (*forwarding*) dans le processeur (modificable par le menu `Configure->Settings`).
Quelles sont les latences des différentes opérations entières et flottantes ?
Exécuter le programme `double.s` pour ces deux configurations du processeur. Le comportement de la multiplication et de la division est-il identique ?
Pourrait-on optimiser ces programmes assembleur ?

3 Comportement de programmes complexes

Exécuter les programmes `fibs.s` (calcul des 12 premiers nombres de Fibonacci tels que $f_{n+1} = f_n + f_{n-1}$) et `daxpy.s` (calcul de $y_i = a \times x_i + y_i$ pour les tableaux x et y).
Avec les deux configurations possibles du processeur, évaluer le nombre de cycles, le CPI et le nombre de suspensions.
Chercher des optimisations simples des programmes `fib.s` et `daxpy.s`.

4 Ecriture de programmes assembleur

Ecrire un programme assembleur correspondant au code C suivant :

```
1 #define N 12
2 double A[N], B[N];
3 for(int i=0; i<N; i++){
4   if(i&0x1) {       // i impair
```

```
5     B[i]=B[i]+A[i];
6   } else {          // i pair
7     B[i]=B[i]-A[i];
8   }
9 }
```

Exécuter le code et évaluer le nombre de suspensions. Montrer que l'on peut fortement optimiser le code par un déroulage de boucle d'un facteur 2.
Même question pour le programme suivant :

```
1 #define N 12
2 double A[N], somme=0.0;
3 for(int i=0; i<N; i++){
4   somme += A[i];
5 }
```

Comment pourrait-on accélérer le temps de traitement ?
[optionnel] Ecrire un programme pour calculer x^n. Dans un premier temps faire une version simple correspondant au code :

```
1 double x=2.0, w=1.0;
2 int n=5;
3 for(int i=0; i<n; i++)
4   w *= x;
```

Puis optimiser l'exécution en calculant les x^{2^k} successifs.

```
1 double x=2.0, w=1.0;
2 int n=5;
3 while(n){
4   if(n&0x01) // n impair ?
5     w *= x;
6   x *= x ;   // calcul de x^(2^k)
7   n>>=1;   // n = n/2
8 }
```

5 Assembleur MIPS supporté par eduMIPS

The following assembler directives are supported

```
.data                  - start of data segment
.text                  - start of code segment
.code                  - start of code segment (same as .text)
.org    <n>            - start address
.space  <n>            - leave n empty bytes
.asciiz <s>            - enters zero terminated ascii string
.ascii  <s>            - enter ascii string
.align  <n>            - align to n-byte boundary
.word   <n1>,<n2>..    - enters word(s) of data (64-bits)
.byte   <n1>,<n2>..    - enter bytes
.word32 <n1>,<n2>..    - enters 32 bit number(s)
.word16 <n1>,<n2>..    - enters 16 bit number(s)
```

```
.double <n1>,<n2>..   - enters floating-point number(s)
```

where <n> denotes a number like 24, <s> denotes a string like "fred", and <n1>,<n2>.. denotes numbers separated by commas. The integer registers can be referred to as r0-r31, or R0-R31, or $0-$31 or using standard MIPS pseudo-names, like $zero for r0, $t0 for r8 etc. Note that the size of an immediate is limited to 16-bits. The maximum size of an immediate register shift is 5 bits (so a shift by greater than 31 bits is illegal). Floating point registers can be referred to as f0-f31, or F0-F31
Flow control instructions (branches, jump, etc) are delayed. The following instruction is always executed.
The following instructions are supported. Note reg is an integer register, freg is a floating-point (FP) register, and imm is an immediate value.

```
lb reg,imm(reg)      - load byte
lbu reg,imm(reg)     - load byte unsigned
sb reg,imm(reg)      - store byte
lh reg,imm(reg)      - load 16-bit half -word
lhu reg,imm(reg)     - load 16-bit half word unsigned
sh reg,imm(reg)      - store 16-bit half-word
lw reg,imm(reg)      - load 32-bit word
lwu reg,imm(reg)     - load 32-bit word unsigned
sw reg,imm(reg)      - store 32-bit word
ld reg,imm(reg)      - load 64-bit double-word
sd reg,imm(reg)      - store 64-bit double-word
l.d freg,imm(reg)    - load 64-bit floating-point
s.d freg,imm(reg)    - store 64-bit floating-point
halt                 - stops the program
daddi reg,reg,imm    - add immediate
daddui reg,reg,imm   - add immediate unsigned
andi reg,reg,imm     - logical and immediate
ori reg,reg,imm      - logical or immediate
xori reg,reg,imm     - exclusive or immediate
lui  reg,imm         - load upper half of register immediate
slti reg,reg,imm     - set if less than immediate
sltiu reg,reg,imm    - set if less than immediate unsigned
beq reg,reg,imm      - branch if pair of registers are equal
bne reg,reg,imm      - branch if pair of registers are not equal
beqz reg,imm         - branch if register is equal to zero
bnez reg,imm         - branch if register is not equal to zero
j imm                - jump to address
jr reg               - jump to address in register
jal imm              - jump and link to address (call subroutine)
jalr reg             - jump and link to address in register
dsll reg,reg,imm     - shift left logical
dsrl reg,reg,imm     - shift right logical
```

```
dsra reg,reg,imm     - shift right arithmetic
dsllv reg,reg,reg    - shift left logical by variable amount
dsrlv reg,reg,reg    - shift right logical by variable amount
dsrav reg,reg,reg    - shift right arithmetic by variable amount
movz reg,reg,reg     - move if register equals zero
movn reg,reg,reg     - move if register not equal to zero
nop                  - no operation
and reg,reg,reg      - logical and
or reg,reg,reg       - logical or
xor reg,reg,reg      - logical xor
slt reg,reg,reg      - set if less than
sltu reg,reg,reg     - set if less than unsigned
dadd reg,reg,reg     - add integers
daddu reg,reg,reg    - add integers unsigned
dsub reg,reg,reg     - subtract integers
dsubu reg,reg,reg    - subtract integers unsigned
dmul reg,reg,reg     - signed integer multiplication
dmulu reg,reg,reg    - unsigned integer multiplication
ddiv reg,reg,reg     - signed integer division
ddivu reg,reg,reg    - unsigned integer division
add.d freg,freg,freg - add floating-point
sub.d freg,freg,freg - subtract floating-point
mul.d freg,freg,freg - multiply floating-point
div.d freg,freg,freg - divide floating-point
mov.d freg,freg      - move floating-point
cvt.d.l freg,freg    - convert 64-bit integer to a double FP form
cvt.l.d freg,freg    - convert double FP to a 64-bit integer form
c.lt.d freg,freg     - set FP flag if less than
c.le.d freg,freg     - set FP flag if less than or equal to
c.eq.d freg,freg     - set FP flag if equal to
bc1f imm             - branch to address if FP flag is FALSE
bc1t imm             - branch to address if FP flag is TRUE
mtc1 reg,freg        - move data from integer register to FP regi
mfc1 reg,freg        - move data from FP register to integer regi
```