Devoir d'architecture

Le devoir sera rendu : par binôme, aux enseignants de TD, dans la semaine du 11 Décembre, date limite STRICTEMENT IMPERATIVE.

Première partie : Architecture logicielle

Les programmes non commentés ne seront PAS PRIS EN COMPTE. En outre, les instructions seront numérotées et on donnera la séquence d'exécution des instructions pour les tests indiqués dans les questions.

On considère l'architecture logicielle définie en annexe.

Exercice 1. Conversion

Le but de l'exercice est d'écrire un programme de transformation d'une chaîne de caractères alphanumériques codée en ASCII en une autre chaîne, où les caractères alphabétiques majuscules sont convertis en minuscules, et les caractères numériques et minuscules sont inchangés. La chaîne est terminée par le caractère codé 0. Le registre R1 contient l'adresse de la chaîne de départ et le registre R2 l'adresse de la chaîne convertie.

Les codes sont les suivants :

Caractères de 0 à 9 : codes de 0x30 à 0x39
Caractères de a à z : codes de 0x61 à 0x7A
Caractères de A à Z : codes de 0x41 à 0x5A

a) Ecrire le fragment de code qui convertit un caractère alphanumérique (majuscule vers minuscule, autres caractères inchangés) placé dans l'octet bas du registre R5 (autres octets à 0) et retourne le résultat dans R5. La séquence d'instructions ne doit pas utiliser d'autre registre que R5.

b) Ecrire le code complet. Tests : chaîne vide ; chaîne « De0 ».

c) Modifier ce code en supposant que les instructions d'accès mémoire disposent en outre du mode d'adressage base + index.

d) On veut limiter le nombre d'accès mémoire, en chargeant en registre les caractères par groupe de 4. Quelle hypothèse faut-il faire sur la chaîne initiale ? Ecrire le code en mode base + déplacement et le tester sur la chaîne « Dev2 ». Comparer le nombre total d'instructions pour une chaîne de longueur N (caractère de terminaison non compris) entre le code de la question b) et ce code.

Exercice 2. Palindromes

Le but de l'exercice est d'écrire un programme *récursif* qui teste si une chaîne de caractères est un palindrome (chaîne qui peut se lire indifféremment de gauche à droite ou de droite à gauche). L'algorithme utilisé est :

```
int palin (int i, int n, char x[]) {
        if (n – 2*i <= 1) return (TRUE) ;
        if (x[i] != x [n – i –1] return (FALSE) ;
        return (palin (i+1, n, x) ;
}
```

x[n] est le tableau de caractères, *n* sa longueur ; le premier appel est *palin (0, n x)* . On suppose que dans le programme appelant, le registre R1 contient *i*, le registre R2 la valeur de *n* et le registre R3 l'adresse de *x* (*&x[0]*).

Les conventions logicielles sont les suivantes : le pointeur de pile d'exécution est le registre R30 ; l'adresse de retour est sauvegardée dans R31 ; les paramètres sont passés dans les registres R1 à R7, et la valeur de retour d'une fonction est retournée dans R1.

Ecrire le programme et le tester sur les chaînes « aba », « abc » et « abba ».

Deuxième partie : caches

On utilise les programmes de simulation de caches vus en TP, et les spécifications du sujet de TP.

Exercice 3. Produit matrice - vecteur

On considère la version suivante du produit matrice–vecteur :

```
for (j = 0 ; j < N ; j++) {
        s = 0 ;
        for (i = 0 ; i < N ; i++)
                Z[i] = Z[i] + X[i][j]*Y[j] ;
        }
```

a) Montrer qu'on peut supposer *Y[j]* en registre pendant la boucle *i* . Modifier Prog32.c en conséquence (on donnera uniquement le fragment de code modifié).

b) Donner une représentation graphique et une table des résultats pour $N = 32$ et $N = 128$ et toutes les configurations du cache.

c) On se place dans le cas idéal d'un cache de taille non bornée complètement associatif. Calculer le nombre d'échecs (en fonction de N et L) pour $j = 0$. Calculer le nombre total d'échecs pour les accès à X, Y et Z et le taux d'échec.

d) Analyser les résultats expérimentaux pour $N = 32$, et $N = 128$, $M = 8K$, correspondance directe.

Exercice 4. Produit de matrices

Prog33.c correspond au produit de matrices carrées $Z = X * Y$.

```
for (i = 0 ; i < N ; i++)
        for (j = 0 ; j < N ; j++) {
                s = 0 ;
                for (k = 0 ; k < N ; k++)
                        s = s + X[i][k]*Y[k][j] ;
                Z[i][j] = s ;
        }
```

a) Donner une représentation graphique et une table des résultats pour $N = 128$ et toutes les configurations du cache.

b) Quels résultats généraux sur les performances des caches sont confirmés ou infirmés par les résultats pour N = 128 ?

Annexe : Jeu d'instructions

L'architecture est 32bits, possède 32 registres 32 bits, r0 est câblé à 0. La mémoire est organisée par octets, en Big Endian, et n'admet que l'accès aligné sur 32 bits.

Dans la table suivant, ES désigne l'extension de signe, et || la concaténation des chaînes de bits.

Mnémonique	Syntaxe	Description
ADD	ADD rd, ra, rb	rd <- ra + rb
ADDI	ADDI rd, ra, imm16	rd <- ra + ES(imm16)
SUB	SUB rd, ra, rb	rd <- ra - rb
AND	AND rd, ra, rb	rd <- ra and rb
ANDI	ANDI rd, ra, imm16	rd <- ra and (0x0000 || imm16)
XOR	XOR rd, ra, rb	rd <- ra xor rb
XORI	XORI rd, ra, imm16	rd <- ra xor (0x0000 || imm16)
OR	OR rd, ra, rb	rd <- ra or rb
ORI	ORI rd, ra, imm16	rd <- ra or (0x0000 || imm16)
LUI	LUI rd, imm16	Place imm16 dans les 16 bits de poids fort de rd, et met les 16 bits de poids faible à 0
SLL	SLL Rd,imm5, Ra	Rd reçoit Ra décalé à gauche de imm5 (non signé) bits.
SAR	SAR Rd,imm5, Ra	Rd reçoit Ra décalé à droite de imm5 (non signé) bits. C'est un décalage arithmétique.
SLR	SLR Rd,imm5, Ra	Rd reçoit Ra décalé à droite de imm5 (non signé) bits. C'est un décalage logique.
SL	SL Rd, Ra, Rb	Rd reçoit Rb décalé de Ra positions, Ra signé (> 0 décalage à droite, < 0 décalage à gauche). C'est un décalage logique.
LD	LD Rd, imm16(Ra)	Rd <- Mem32[Ra +ES (imm16)]
LDH	LDH Rd, imm16(Ra)	Rd <- ES(Mem16[Ra +ES (imm16)])
LDUH	LDUH Rd, imm16(Ra)	Rd <- 0x0000 || (Mem16[Ra +ES (imm16)])
LDB	LDB Rd, imm16(Ra)	Rd <- ES(Mem8[Ra +ES (imm16)])
LDUB	LDUB Rd, imm16(Ra)	Rd <- 0x000000 || (Mem8[Ra +ES (imm16)])
ST	ST Rd, imm16(Ra)	Mem32[Ra +ES (imm16)] <- Rd
STH	STH Rd, imm16(Ra)	Mem16[Ra +ES (imm16)] <- Rd
STB	STB Rd, imm16(Ra)	Mem8[Ra +ES (imm16)] <- Rd
B*cond*	Bcond imm30	Si cond vraie PC <- PC + ES(imm30*4)
JMPL	JMPL rd, ra, imm16	rd <- PC ; PC <- ra +ES(imm16*4)

cond	Condition testée
A	Toujours
EQ	Egal
N	Négatif
LE	≤ signé
G	> signé
LEU	≤ non signé
GU	> non signé
O	Overflow
C	Retenue

Dans la syntaxe assembleur, les constantes peuvent être notées :

- en décimal, exemple ADDI R1, R2, 10
- en hexadécimal, exemple ADDI R1, R2, 0xA

Pour les instructions Bcond et JMPL, on utilisera des étiquettes symboliques.