CC-TD-M - LIF6 Architecture matérielle et logicielle

Lundi 23 avril 2012 - durée 1h30 - aucun document autorisé

I Représentation à virgule flottante (5 pts)

On travaille sur une machine où les flottants binaires sont codés sur 12 bits. On considère le code :

$c =$

11	10	9	8	7	6	5	4	3	2	1	0
s	e_c				m_c						

Le nombre représenté est $f(c) = (-1)^s \times (1, m_c)_2 \times 2^{e(c)}$. Si $1 \leq N(e_c) \leq 14$, alors $e(c) = N(e_c) - 2^3$ (on ne se préoccupe pas du codage des sous-normaux ni des valeurs exceptionnelles ici).

Q.I.1) - (1,5 pts) Quel nombre est représenté par le code octal $c = 7257_8$, si on le regarde comme le codage d'un flottant ? Donnez votre résultat sous la forme d'une représentation positionnelle à virgule en décimal.

Le code $c = 7257_8$ représente $111\ 010\ 101\ 111_2 = 1\ 1101\ 0101111_2$. L'exposant du flottant considéré est donc $e(c) = N(e_c) - 8 = (1101)_2 - 8 = 13 - 8 = 5$. La mantisse du flottant est $(1, 0101111)_2$. On a donc

$$\begin{aligned} f(c) &= -(1, 0101111)_2 \times 2^5 \\ &= -(101011, 11)_2. \end{aligned}$$

On trouve que $(101011)_2 = (43)_{10}$ et $(0, 11)_2 = 0.5 + 0.25 = (0, 75)_{10}$, donc $f(c) = -(43, 75)_{10}$.

Q.I.2) - (1,5 pts) Comment est représenté le nombre $(3, 2)_{10}$ en représentation positionnelle à virgule en binaire ? Détaillez un peu vos calculs.

On trouve après calculs que $(0, 2)_{10} = (0, \underline{0011})_2$, donc $(3, 2)_{10} = (11, \underline{0011})_2$.

Q.I.3) - (2 pts) Comment peut-on représenter le nombre $(3, 2)_{10}$ dans le format flottant binaire décrit ci-dessus ? Vous effectuerez un arrondi au plus proche, et exprimerez votre résultat sous la forme d'un code hexadécimal.

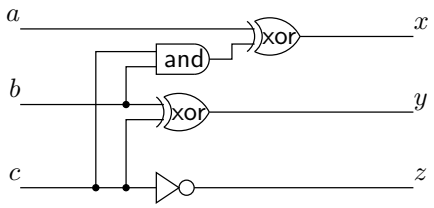
On travaille avec en tout 8 bits de précision :

$$\begin{aligned} (3, 2)_{10} &= (11, 0011001100110\underline{0011})_2 \times 2^0 \\ &= (1, 1001100\ 110\underline{0011})_2 \times 2^1 \\ &\approx (1, 1001101)_2 \times 2^1, \end{aligned}$$

après arrondi au plus proche. On a donc $N(e_c) = e(c) + 8 = 1 + 8 = 9 = (1001)_2$ et $m_c = 1001101_2$. Le codage c de $(3, 2)_{10}$ au format flottant, en arrondi au plus proche, sera donc $c = 0\ 1001\ 1001101_2 = 0100\ 1100\ 1101_2 = 4CD_H$.

II Un circuit combinatoire (4 pts)

On considère le circuit combinatoire suivant, dont les entrées sont a, b et c, et les sorties x, y et z :



Q.II.1) - (1 pt) Donnez des expressions booléennes pour x, y et z en fonction de a, b et c.

$x = a \oplus bc$, $y = b \oplus c$ et $z = \overline{c}$.

Q.II.2) - (1 pt) Complétez la table de vérité suivante :

a	b	c	x	y	z
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

a	b	c	x	y	z
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Q.II.3) - (2 pts) Quelle est la fonction réalisée par le circuit considéré (exprimez $(xyz)_2$ en fonction de $(abc)_2$) ?

$(xyz)_2 = (abc)_2 + 1 \mod 8$.

III Décompte des bits non-nuls (6 pts)

En C, le type `unsigned char` permet de coder des entiers naturels sur 8 bits. A titre d'exemple, on donne la fonction C suivante qui retourne le nombre de bits non-nuls de son argument (rappelons que `n/2` et `n%2` désignent respectivement le quotient et le reste dans la division euclidienne de l'entier `n` par 2) :

```
int nzbcount(unsigned char m) {
  n=m; i=0;
  while(n != 0) {
    if(n%2 == 1) i++;
    n=n/2;
  }
  return(i);
}
```

On va mettre au point une autre fonction pour compter le nombre de bits non-nuls d'un `unsigned char`. Si `a` et `b` sont deux entiers naturels, on rappelle que `a&b` calcule en C le « ET bit-à-bit » entre `a` et `b`.

Q.III.1) - (1 pt) Pour `a` $= (00000001)_2$, `b` $= (10111100)_2$, `c` $= (10000000)_2$, calculer `(a-1)&a`, `(b-1)&b` et `(c-1)&c`. Détaillez vos calculs (posez à chaque fois la soustraction en binaire, puis effectuez le « ET bit-à-bit »).

On trouve que `(a-1)&a` donne 0 :

$$\begin{array}{cccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ - & & & & & & & & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \quad \Rightarrow \quad \begin{array}{cccccccc} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \& & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

On trouve que `(b-1)&b` donne $(10111000)_2$:

$$\begin{array}{cccccccc} & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ - & & & & & & & & 1 \\ \hline & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{array} \quad \Rightarrow \quad \begin{array}{cccccccc} & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ \& & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$$

On trouve que exttt(d-1)&d donne à nouveau 0 :

$$\begin{array}{cccccccc} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ - & & & & & & & & 1 \\ \hline & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \quad \Rightarrow \quad \begin{array}{cccccccc} & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \& & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Q.III.2) - (2 pts) Si `n` est un `unsigned char`, que permet de faire l'opération `(n-1)&n` ?

L'opération `n&(n-1)` permet d'annuler le bit non-nul de poids le plus faible dans `n`.

Q.III.3) - (2 pts) Déduisez de la question précédente une nouvelle fonction C pour calculer le nombre de bits non nuls d'un `unsigned char`.

Il suffit, dans le programme C donné dans l'énoncé, de remplacer la boucle while par :
`while(n != 0) { i++; n=(n-1)&n; }`
A chaque itération, on annule le bit de poids le plus faible de `n`, et on incrémente `i`. On sort de l'itération quand `n` est nul.

Q.III.4) - (1 pt) Avec la fonction initiale `nzbcount`, combien d'itérations de la boucle `while` sont nécessaires pour compter le nombre de bits non-nuls d'un `unsigned char` ? Même question pour votre fonction.

Soit k le rang du bit non nul de poids le plus fort dans `n` (les poids vont de 0 à 7) : avec la fonction initiale, il faut exactement $k+1$ itérations de la boucle. Avec la nouvelle fonction, il faut exactement autant d'itérations qu'il y a de bits non-nuls dans `n`.

IV Mémoire cache (5 pts)

On considère le programme C suivant :

```
int main(void) {
  double A[8], B[8], C[8];
  int i;
  for(i=0; i<8; i++) C[i] = A[i] + B[i];
  ...
}
```

On suppose que l'on compile et que l'on exécute ce programme sur un ordinateur qui ne dispose que d'un seul niveau de cache de données direct : ce cache comporte 4 entrées de 32 octets (on ne se préoccupe pas du chargement des instructions, ni de la variable `i` dans cet exercice). On note les lignes de cache $\ell_0, \ell_1, \ell_2, \ldots$, et les entrées du cache e_0, e_1, e_2, e_3. Les flottants de type `double` sont d'une taille de 64 bits. Les tableaux sont stockés de manière contigüe en mémoire, dans l'ordre de leur déclaration.

Q.IV.1) - (1 pt) Combien de flottants de type `double` peut contenir une ligne de cache ?

Un `double` occupe 8 o. Une ligne de cache de 32 o peut donc contenir 4 `double`s.

Q.IV.2) - (1 pt) Combien de lignes de caches consécutives occupe chacun des tableaux `A`, `B` et `C` ?

Chaque ligne de cache compte 4 double, donc un tableau de 4 `double`s occupe 2 lignes de cache consécutives.

Q.IV.3) - (1,5 pt) En supposant que `A[0]` est aligné sur le début de la ligne de cache ℓ_0, indiquez à quelles lignes de cache appartiennent les élément de `A`, `B` et `C`. Indiquez également dans quelles entrées du cache seront rangés ces élements lorsqu'ils seront accédés.

- `A[0...3]` appartiennent à ℓ_0, stockés dans e_0
- `A[4...7]` appartiennent à ℓ_1, stockés dans e_1
- `B[0...3]` appartiennent à ℓ_2, stockés dans e_2
- `B[4...7]` appartiennent à ℓ_3, stockés dans e_3
- `C[0...3]` appartiennent à ℓ_4, stockés dans e_0
- `C[4...7]` appartiennent à ℓ_5, stockés dans e_1

Q.IV.4) - (1,5 pts) Dans un tableau de la forme indiquée ci-dessous, indiquez quelle ligne de cache contient chaque entrée du cache à la fin de chaque itération de la boucle `for(i=0; i<8; i++) C[i] = A[i] + B[i]` ; (Vous placerez un « ? » quand on ne peut pas savoir). Indiquez également le nombre de *cache miss* qui ont eu lieu au cours de chaque itération.

`i`	0	1	2	3	4	5	6	7
e_0								
e_1								
e_2								
e_3								

`i`	0	1	2	3	4	5	6	7
e_0	ℓ_4	ℓ_4	ℓ_4	ℓ_4	ℓ_4	ℓ_4	ℓ_4	ℓ_4
e_1	?	?	?	?	ℓ_5	ℓ_5	ℓ_5	ℓ_5
e_2	ℓ_2	ℓ_2	ℓ_2	ℓ_2	ℓ_2	ℓ_2	ℓ_2	ℓ_2
e_3	?	?	?	?	ℓ_3	ℓ_3	ℓ_3	ℓ_3
cache miss	3	2	2	2	3	2	2	2