Représentation des nombres : signe + partie entière

Le but de ce TP est de manipuler la représentation binaire de nombres sous la forme *signe + partie entière*. Plus précisément, vous devrez implémenter différentes méthodes afin de réaliser des opérations de base (telles que la transformation d'un nombre entier en binaire de la forme signe + partie entière,[1] affichage de la représentation binaire, décalage de bit, ...) et de plus haut niveau (addition, soustraction, multiplication et division).

Représentation

Dans la suite les nombres binaires seront représentés dans un tableau de `NB_BITS` entiers.

```
#define NB_BITS 8   // constante permettant de fixer le nombre de bit
...

int t[NB_BITS];     // déclaration d'un nombre binaire
```

Par exemple le tableau suivant représente le nombre -9 codé sur 8 bits en binaire avec la représentation signe + partie entière :

1	0	0	0	1	0	0	1

Opérations de base

Implémentez les fonctions permettant de réaliser les opérations suivantes :

1. Afficher un nombre en binaire à l'écran.

```
/* entête de fonction */
void print(int t[NB_BITS]){
  // affiche t suivant le format : 1000 1001
}
```

2. Convertir un nombre entier en binaire.

```
/* entête de fonction */
void intToBin(int v, int t[NB_BITS]){
  // ex: si v = -9 alors t = 10001001
}
```

3. Convertir un nombre binaire en entier.

```
/* entête de fonction */
int toInt(int t[NB_BITS]){
  // ex: si t = 10001001 alors la fonction retourne -9
}
```

4. Assigner un nombre binaire à un autre.

```
/* entête de fonction */
void assign(int v[NB_BITS], int o[NB_BITS]){
  // ex: copier le tableau v dans o, ainsi si v = 10001001 alors o = 10001001
}
```

[1] par la suite lorsqu'il sera fait référence à un nombre binaire nous supposerons toujours qu'il est sous la forme : signe + partie entière

5. Implémenter les relations de comparaisons >, < et == sur la partie entière de deux nombres binaires (c'est-à-dire que le bit de signe n'est pas pris en compte dans la comparaison).

```
/* entêtes de fonctions */
bool gt_abs(int a1[NB_BITS], int a2[NB_BITS]){
  // true si abs(a1) > abs(a2), faux sinon
}

bool lt_abs(int a1[NB_BITS], int a2[NB_BITS]){
  // true si abs(a1) < abs(a2), faux sinon
}

bool eq_abs(int a1[NB_BITS], int a2[NB_BITS]){
  // true si abs(a1) == abs(a2), faux sinon
}
```

6. Implémenter les relations de comparaisons >, < et == sur deux nombres binaires.

```
/* entêtes de fonctions */
bool gt(int a1[NB_BITS], int a2[NB_BITS]){
  // true si a1 > a2, faux sinon
}

bool lt(int a1[NB_BITS], int a2[NB_BITS]){
  // true si a1 < a2, faux sinon
}

bool eq(int a1[NB_BITS], int a2[NB_BITS]){
  // true si a1 == a2, faux sinon
}
```

7. Implémenter les décalages de bits à gauche et à droite.

```
/* entêtes de fonctions */
void lShift(int v[NB_BITS]){
  // si v = 10001001 alors lShift(v) = 00010010
}

void rShift(int v[NB_BITS]){
  // si v = 10001001 alors lShift(v) = 01000100
}
```

Opérations de haut niveau

Implémentez les fonctions permettant de réaliser les opérations suivantes :

1. Addition de deux nombres binaires.

```
/* entête de fonction */
void add(int a1[NB_BITS], int a2[NB_BITS], int r[NB_BITS]){
  // r = a1 + a2
}
```

2. Soustraction de deux nombres binaires.

```
/* entête de fonction */
void sub(int a1[NB_BITS], int a2[NB_BITS], int r[NB_BITS]){
  // r = a1 - a2
}
```

3. Multiplication de deux nombres binaires.

```
/* entête de fonction */
void mul(int a1[NB_BITS], int a2[NB_BITS], int r[NB_BITS]){
  // r = a1 * a2
}
```

4. Division de deux nombres binaires (attention vous devez gérer le cas de la division par zéro).

```
/* entête de fonction */
void div(int a1[NB_BITS], int a2[NB_BITS], int d[NB_BITS], int r[NB_BITS]){
  // d = a1 / a2 et r = a1 % a2
}
```