# *Architecture des ordinateurs*

*TP2 : arithmétique signée et non signée*

## 1. Additionneur 4 bits

Réutiliser le module vu au TP1 pour construire un additionneur 4 bits, d'interface :

```
module adder4(a[3..0], b[3..0], cin : s[3..0], cout)
```

## 2. Additionneurs 8 bits, 16 bits, 32 bits

Construire de façon modulaire des additionneurs de tailles croissantes et d'interfaces :

```
module adder8(a[7..0], b[7..0], cin : s[7..0], cout)
module adder16(a[15..0], b[15..0], cin : s[15..0], cout)
module adder32(a[31..0], b[31..0], cin : s[31..0], cout)
```

## 3. Additionneur-soustracteur 32 bits

En réutilisant le module adder32 développé précédemment, concevoir un module additionneur soustracteur 32 bits ayant l'interface suivant :

```
module addsub32(a[31..0], b[31..0], sub : s[31..0], V, C)
```

Lorsque sub = 0 le module calcule la somme s[31..0] = a[31..0] + b[31..0] ; lorsque sub = 1, il calcule la différence. Les bits V et C ont le sens habituel des bits de débordement et de retenue-emprunt pour les opérations d'addition et de soustraction.

Tester avec le vecteur de tests fourni.

## 4. Comparateurs non signés

Concevoir modulairement des comparateurs non-signés d'interface :

```
module ucmp1(a, b : sup, eq)
module ucmp2(a[1..0], b[1..0] : sup, eq)
module ucmp4(a[3..0], b[3..0] : sup, eq)
module ucmp8(a[7..0], b[7..0] : sup, eq)
module ucmp16(a[15..0], b[15..0] : sup, eq)
```

`sup` indique que A est strictement supérieur à B, `eq` qu'ils sont égaux

## 5. Décaleur à barillet

Concevoir de façon modulaire des décaleurs de tailles croissantes :

```
module shift1(r, en, e[31..0] : s[31..0])
module shift2(r, en, e[31..0] : s[31..0])
module shift4(r, en, e[31..0] : s[31..0])
module shift8(r, en, e[31..0] : s[31..0])
module shift16(r, en, e[31..0] : s[31..0])
```

Si en=0, s[31..0] = e[31..0] ; sinon s[31..0] est égal à e[31..0] décalé vers la droite (resp. gauche) si r = 1 (resp. 0), d'un nombre fixe de bits. Les bits sortants sont perdus, les bits entrants sont des zéros.

Concevoir finalement un module de décalage générique, d'interface :

```
module barrelshifter32(r, nb[4..0], e[31..0] : s[31..0])
```

s[31..0] est égal à e[31..0] décalé vers la droite (resp. gauche) si r = 1 (resp. 0), d'un nombre de bits égal à nb[4..0]. Les bits sortants sont perdus, les bits entrants sont des zéros.

## 6. Unité arithmétique et logique

Créer une unité arithmétique et logique ayant l'interface suivante :

```
module ual(a[31..0], b[31..0], cmd[5..0] : s[31..0], N, Z, V, C)
```

L'opération est spécifiée par les bits de commande cmd[5..0] de la façon suivante :

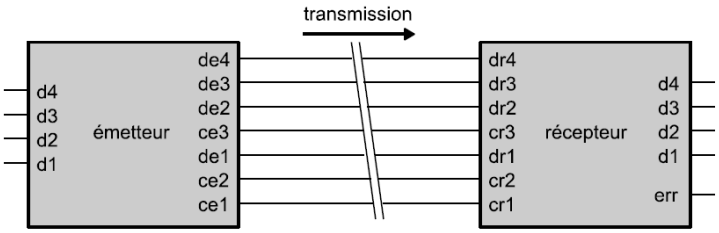
| **cmd[5..0]** | **opération** |
|---|---|
| 000000 (0) | ADD, addition entre a[31..0] et b[31..0] |
| 000100 (4) | SUB, soustraction entre a[31..0] et b[31..0] |
| 000001 (1) | AND, et logique bit à bit entre a[31..0] et b[31..0] |
| 000010 (2) | OR, ou logique bit à bit entre a[31..0] et b[31..0] |
| 000011 (3) | XOR, xor logique bit à bit entre a[31..0] et b[31..0] |
| 001101 (13) | Décalage à droite de b[4..0] bits de a[31..0] |
| 001110 (14) | Décalage à gauche de b[4..0] bits de a[31..0] |
| 011010 (26) | UMULCC, multiplication non signée entre a[15..0] et b[15..0] |

- N indique que le résultat est négatif, Z qu'il est nul, V et C sont les bits de retenue et d'overflow qui sortent de l'additionneur/soustracteur
- un multiplieur déjà câblé dans le FPGA sera recruté par utilisation du module prédéfini d'interface :

```
umult16x16(a[15..0], b[15..0]: s[31..0])
```

## 7. Code autocorrecteur de Hamming (optionnel)

On utilise un code de Hamming lors d'une transmission de données, afin de détecter l'existence d'une erreur, mais aussi sa position si l'erreur est unique ; dans ce cas elle peut alors être automatiquement corrigée. Le principe consiste à transmettre un message construit à partir des bits de données à transmettre, dans lequel sont insérés des bits de parité.




Les modules suivants implémentent le code de Hamming pour la transmission de 4 bits de données ; le calcul des bits de parité a été caché. Analyser le fonctionnement des modules, et retrouver le calcul de parité qui permet le bon fonctionnement.

```
 // 7  6  5  4  3  2  1
 // d4 d3 d2 c3 d1 c2 c1
 module emetteur(d4,d3,d2,d1 : de4,de3,de2,c3,de1,c2,c1)
    de4 = d4; de3 = d3; de2 = d2; de1 = d1;
    // calcul des bits de contrôle qui sont ajoutés au message
    bits_ctrl(d4,d3,d2,d1 : c3,c2,c1);
 end module

 module recepteur(dr4,dr3,dr2,cr3,dr1,cr2,cr1 : err, d4,d3,d2,d1)
    // même calcul des bits de contrôle sur le message reçu
 bits_ctrl(dr4,dr3,dr2,dr1 : c3,c2,c1);
    // la position de l'err est la différence entre (c3,c2,c1) et
(cr3,cr2,cr1)
    poserr[2] = cr3*/c3+/cr3*c3;
    poserr[1] = cr2*/c2+/cr2*c2;
    poserr[0] = cr1*/c1+/cr1*c1;
    // décodage du numéro de l'erreur
    decoder3to8(poserr[2..0] : derr[7..0]);
    // il y a une erreur si poserr est non nul
    err = /derr[0];
    // s'il y a une erreur en position 7, 6, 5, 3 (donnée), elle est
corrigée
    d4 = dr4*/derr[7] + /dr4*derr[7];
    d3 = dr3*/derr[6] + /dr3*derr[6];
    d2 = dr2*/derr[5] + /dr2*derr[5];
    d1 = dr1*/derr[3] + /dr1*derr[3];
 end module

 module bits_ctrl(d4,d3,d2,d1 : c3,c2,c1)
    ouex3(-,-,- : c1);
    ouex3(-,-,- : c2);
    ouex3(-,-,- : c3);
 end module
```

Ecrire le module ouex3 référencé dans bits_ctrl.

Ecrire un module de test qui permette de suivre la transmission d'une donnée et sa correction ; faire en sorte qu'une ou plusieurs erreurs puissent être introduites dans le message.

Si vous deviez transmettre plus de données, quelles serait la structure du message à envoyer ? (nombre de bits de données, de contrôle, positions).

***Notation***:

- Additionneurs 4 bits, 32 bits
- Additionneur / soustracteur
- Comparateurs non-signés
- Décaleur à barillet
- UAL