La Structure de Données Lofig

La structure de données `Lofig` est conçue pour représenter les *netlists* en mémoire. Les attributs des différents objets constituants cette structure ont été présentés en cours. Nous allons maintenant compléter avec les déclarations des fonctions membres.

Fonctions membres de la class Locon

Les constructeurs :

- Un `Locon` peut appartenir, soit à un modèle (`Lofig`), soit à une instance (`Loins`). Il y aura donc deux constructeurs correspondants à chacune de ces possibilités. On fournit en outre son nom et sa direction. Le type sera déduit à partir du constructeur appelé.
    - `Locon ( Lofig*, const std::string& name, unsigned int dir );`
    - `Locon ( Loins*, const std::string& name, unsigned int dir );`

Les accesseurs :

- `std::string getName ();`
- `Losig* getSignal ();`
- `Lofig* getModel ();`
- `Loins* getInstance ();`
- `unsigned int getDirection();`

Les modificateurs (*mutators*) :

- Un `Locon` peut (doit) être associé à un signal. On pourra le faire de deux façons différentes, soit en donnant explicitement un pointeur sur le `Losig`, soit en indiquant simplement le nom du signal.
    - `void setSignal ( Losig* );`
    - `void setSignal ( const std::string& );`
- Positionnement de la direction :
    - `void setDirection ( unsigned int );`

De plus, le code de trois fonctions membres vous sont fournies pour pouvoir afficher l'objet dans un flux. Dans `Locon.h` :

```
class Locon {
  public:
    static std::string   typeToString ( unsigned int );
    static std::string   dirToString  ( unsigned int );
  public:
            void         xmlDrive     ( std::ostream& );
};
```

Code de Locon.cpp.

Fonctions membres de la classe Losig

Le constructeur :

- `Losig ( Lofig*, const std::string&, unsigned int type );`

Les arguments correspondent respectivement à la `Lofig` auxquel le signal appartient, son nom et son type. L'identificateur sera demandé directement à la `Lofig` dans le corps du constructeur.

Les accesseurs :

- `Lofig* getOwner ();`
- `std::string& getName ();`
- `unsigned int getId ();`
- `unsigned int getType ();`

Le code de la fonction utilitaire d'affichage dans un flux : Losig-Skeleton.cpp

Fonctions membres de la classe Loins

Le constructeur :

- `Loins ( Lofig* owner, Lofig* model, const std::string& );`

Le destructeur :

- `~Loins ()`

Il entrainera la destruction de ses connecteurs `Locon`.

Les arguments sont l'`owner`, la `Lofig` *dans laquelle* l'instance est crée, le `model`, la `Logig` *dont* on créé une instance et le nom de cette instance. Le constructeur se charge de la duplication des connecteurs du modèle `model` dans l'instance.

Les accesseurs :

- `std::string& getName ();`
- `Lofig* getModel ();`
- `Lofig* getOwner ();`
- `std::list<Locon*>& getConnectors ();`
- `Locon* getConnector ( const std::string& );`

Les modifieurs :

- `bool connect ( const std::string& name, Losig* );`

Réalise l'association entre un connecteur (`Locon`) *de l'instance* et un signal `name` (de la `Lofig` `owner`).

Le code de la fonction utilitaire d'affichage dans un flux : Loins-Skeleton.cpp

Fonctions membres de la classe Lofig

Le constructeur :

- `Lofig ( const std::string& );`

Le constructeur est extrèmement simple, il se contente de positionner le nom de la `Lofig`, tous les autres attributs sont *vides*.

Le destructeur :

- `~Lofig ()`

Il entrainera la destruction~:

- De ses instances.
- De ses signaux.
- De ses connecteurs.

Mais pas de ses modèles, qui sont autant de `Lofig` pouvant apparaître comme modèles dans d'autres `Lofig`

Les accesseurs :

- `unsigned int getMode ();`
- `const std::string& getName ();`
- `std::list<Lofig*>& getModels ();`
- `std::list<Locon*>& getConnectors ();`
- `std::list<Loins*>& getInstances ();`
- `std::list<Losig*>& getSignals ();`

Les modificateurs :

- `setMode ( unsigned int );`

Les modificateurs relatifs aux modèles :

- Ajoute un nouveau modèle, soit en donnant directement sa `Lofig`, soit par nom. On n'ajoutera pas un modèle s'il est déjà présent.
  - `void addModel ( const std::string& );`
  - `void addModel ( Lofig* );`
- Retrait d'un modèle de la liste. Mêmes variantes que pour l'ajout.
  - `void removeModel ( const std::string& );`
  - `void removeModel ( Lofig* );`
- Recherche d'un modèle par son nom. S'il n'est pas trouvé, la fonction renverra `NULL`.
  - `Lofig* findModel ( const std::string& );`

Les modificateurs relatifs aux connecteurs :

- Ajoute un nouveau connecteur à la `Losig`. On doit fournir son nom, sa direction ainsi que le signal auquel il est relié. Cela implique que l'on créé les signaux *avant* les connecteurs.
  - `void addConnector ( const std::string&, unsigned int dir, Losig* );`
- Suppression d'un connecteur, soit directement par un pointeur sur le `Locon`, soit par nom.

    - ♦ `void removeConnector ( const std::string& );`
    - ♦ `void removeConnector ( Locon* );`
- Recherche d'un connecteur par son nom.
    - ♦ `Locon* findConnector ( const std::string& );`
- Réalisation d'une connexion. La première version correspond à la (re)connexion d'un signal sur un `Locon` du modèle. La seconde version à la connexion d'un `Locon` d'une instance (référencé par son nom `conName`) à un signal du modèle référencé par son nom `sigName`.
    - ♦ `bool connect ( const std::string& conName, const std::string& sigName );`
    - ♦ `bool connect ( const std::string& insName, const std::string& conName, const std::string& sigName );`

Les modificateurs relatifs aux instances :

- Ajout d'une nouvelle instance. La première variante non seulement ajoute l'instance, mais aussi la créé (l'alloue par appel au constructeur de `Loins`). Dans le second cas on dispose déjà de l'instance. Dans les deux cas on effectuera une vérification de cohérence pour empécher la création de deux instances de même nom.
    - ♦ `void addInstance ( const std::string& modelName, const std::string& insName );`
    - ♦ `void addInstance ( Lofig*, const std::string& insName );`
- Supprime une instance du modèle.
    - ♦ `void removeInstance ( const std::string& );`
    - ♦ `void removeInstance ( Loins* );`
- Cherche une instance par son nom.
    - ♦ `Loins* findInstance ( const std::string& );`
- Retourne `true` si l'instance pointée appartient bien à ce modèle.
    - ♦ `bool hasInstance ( Loins* );`

Les modificateurs relatifs aux signaux :

- Ajout d'un signal. Faire une vérification de cohérence, on ne doit pas avoir deux signaux de même nom.
    - ♦ `void addSignal ( const std::string&, unsigned int type );`
- Destruction d'un signal, par nom ou diectement par pointeur.
    - ♦ `void removeSignal ( const std::string& );`
    - ♦ `void removeSignal ( Losig* );`
- Recherche d'un signal, par nom ou par identificateur. Si le signal n'existe pas, renvoyer `NULL`.
    - ♦ `Losig* findSignal ( const std::string& );`
    - ♦ `Losig* findSignal ( unsigned int id );`
- Renvoie un nouvel identificateur de signal. Il doit avoir la garentie d'être unique.
    - ♦ `unsigned int _newSignalId ();`

Fonctions statiques :

- Retourne la liste de toutes les `Lofig` présentent en mémoire.
    - ♦ `static std::list<Lofig*>& getAll ();`
- Recherche une `Lofig` dans la liste générale.
    - ♦ `static Lofig* findFromAll ( const std::string& );`
- Désalloue la totalité des `Lofig`.
    - ♦ `static void destroyAll ();`
- Ajoute une `Lofig` à la liste générale (vérifier l'unicité).
    - ♦ `static void _addToAll ( Lofig* );`
- Retire une `Lofig` de la liste générale (par nom ou directement par pointeur).
    - ♦ `static void _removeFromAll ( Lofig* );`

- `static void _removeFromAll ( const std::string& );`

De plus, le code de deux fonctions membres vous sont fournies pour pouvoir afficher l'objet dans un flux. Dans `Lofig.h` :

```
class Lofig {
  public:
    static std::string   modeToString ( unsigned int );
  public:
            void         xmlDrive     ( std::ostream& );
};
```

Le code de la fonction utilitaire d'affichage dans un flux : Lofig-Skeleton.cpp

Destruction de la Base de Données

Cette base de donnée est composées d'objets contenant des pointeurs sur différents autres objets. La destruction de l'un de ces objets doit donc se faire de façon à préserver la cohérence de la base de données.

Par exemple, la destruction d'un signal *devrait* entrainer son retrait des différents `Locon` qui pointent sur lui. De même, la destruction d'une `Lofig` devrait entraîner sa suppression de toutes les listes de modèles.

Cette gestion est beaucoup trop compliquée à implanter dans le cadre d'un TME, nous ne le ferons donc pas.

En revanche, la fonction `Lofig:destroyAll` détruisant la totalité de la base de données sera implantée.

Travail à réaliser

Implanter la structure de donnée `Lofig` à partir la spécification ci-dessus.

Pour valider votre travail, il vous est demandé de décrire la *netlist* ci dessous, représentant un *full adder* (brique de base des additionneurs).

Un *full adder* se décompose en deux *half adder*. A titre d'exemple, la construction du *half adder* vous est fournie Main-Skeleton.cpp. La représentation compléte du *full adder* sous forme de structure `Lofig` est présentée figure 2.

La Mini Bibliothèque de Modèles

La bibliothèque vous fourni trois modèles (`Lofig`), `And2`, `Xor2` et `Or2`.

Interface simplifiée de la bibliothèque :

```
class Library {
  public:
    enum ModelType { And2=0, Or2, Xor2, ModelTypeSize };
  public:
    static void     destroy     ();
    static Lofig*   getModel    ( unsigned int modelType );
};
```

Code de la bibliothèque : Library.h Library.cpp