La Structure de Données Lofig

La structure de données `Lofig` est conçue pour représenter les *netlists* en mémoire. Les attributs et les méthodes des différents objets constituants cette structure ont été présentés en cours.

Compte tenu du volume de travail demandé, ce TME sera réalisé en deux séances (4 & 5).

Programmation Modulaire

La structure de données est trop grosse pour pouvoir être implantée dans un unique fichier. Nous allons donc la découper en plusieurs fichiers, en suivant la règle **«une paire de fichiers (`.h`, `.cpp`) par classe de la base.**

Ce qui donne:

- `Locon` : `Locon.h` et `Locon.cpp`.
- `Losig` : `Losig.h` et `Losig.cpp`.
- `Loins` : `Loins.h` et `Loins.cpp`.
- `Lofig` : `Lofig.h` et `Lofig.cpp`.

Structure Complète du TME

En plus des fichiers directement liés à l'implantation de la base de donnée, le projet comprend une petite bibliothèque de modèles prédifinis (`Library` et une classe `Indentation` pour gérer l'indentation de l'affichage dans un flux.

`Library` et `Indentation` vous sont founis, ainsi qu'un `main` de démonstration implantant l'`halfadder`.

Afin de pouvoir vérifier la validité des *netlists* que vous devrez décrire, une méthode `xmlDrive(std::cout&)` (non présentée en cours) est fournie pour chaque classe de la base, dans des squelettes de fichiers.

Fichiers Fournis:

- Locon.cpp

- Losig.cpp
- Loins.cpp
- Lofig.cpp
- Library.h
- Library.cpp
- Indentation.h
- Indentation.cpp
- Main.cpp

Compilation du Programme

Lorsque l'on fait de la programmation modulaire, il est hors de question de compiler «à la main» tous les fichiers. On passe obligatoirement par un `Makefile`. Ce fichier devra obligatoirement accompagner votre compte- rendu.

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 );
};
```

Objets de la Base de Données

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& );
};
```

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 ();`

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 `Losig` (de la `Lofig` `owner`).

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 :

- `void 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. Dans la première variante, on dispose déjà du modèle (`Lofig`). Dans la seconde variante, le modèle est donné par son nom, il faut donc faire une recherche de la `Lofig` associée puis appeler la première variante. On effectuera une vérification de cohérence pour empécher la création de deux instances de même nom.
  - `void addInstance ( Lofig*, const std::string& insName );`
  - `void addInstance ( const std::string& modelName, 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& );
};
```

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.

Dans un premier temps, on ignorera les problèmes liés à la destruction des objets.

Travail à réaliser

Plutôt que d'implanter la totalité de la base en une seule fois, puis tout compiler et enfin tout déboguer, il est préférable d'adopter une approche par étape.

Question 1

Création de d'une base de donnée vide: écrire les déclarations des différentes classes, mais ne créer que des fonctions membre aux corps vide.

Créer le `Makefile` de compilation du projet.

Compiler et vérifier qu'il n'y a pas d'erreur.

Question 2

En plus de l'affichage `XML`, il vous est possible d'utiliser un visualisateur (primitif) de *netlist*. Pour l'utiliser, récupérez les fichiers suivants:

- viewer.mk portion de Makefile
- MbkBridge.h *header* pour le convertisseur vers `Hurricane`.
- MbkBridge.cpp corps du convertisseur.
- RawViewer.h *header* pour le visualisateur.

- RawViewer.cpp corps du visualisateur.

Dans votre `Makefile`, ajoutez près du début du fichier:

```
# "tme45" est le nom de votre binaire.
all: tme45

include viewer.mk
```

Principe de la première règle: si rien n'est précisé sur la ligne de commande, `make` exécute la première règle rencontrée. Pour que, par défaut, il construise votre binaire, ajoutez *avant* le `include` une régle **all**.

Si vous avez répartis vos `.h` et `.cpp` dans différents répertoires, positionnez dans le `Makefile` les variables suivantes (en adaptant):

```
INC_DIR = ./includes
OBJ_DIR = ./obj
SRC_DIR = ./src
```

Positionnement de l'environnement: avant de lancer `make`, positionnez l'environnement avec la commande suivante:

```
source /soc/coriolis2/etc/coriolis2/coriolis2.sh
```

Enfin, dans le `Main.cpp`, ajouter:

```
#include "MbkBridge.h"
#include "RawViewer.h"

// ...

int main ( int argv, char* argv )
{
  // ...
  RawViewer::run ( argc, argv, placeAsNetlist(toHurricane(halfadder)));
}
```

Question 3

Implanter, classe par classe les métodes de `Locon`, `Losig`, `Loins` et `Lofig`. Compiler systématiquement entre chaque étape en vérifiant, avec l'affichage `xml` que les composants ajoutés à chaque étape apparaissent bien. On utilisera le `Main` fourni (de l'`halfadder`).

Ordre d'implantation des classes:

1. La classe `Loins`, en ommetant la gestion de la duplication des connecteurs du modèle. Dans la même étape, ajouter dans la `Lofig` l'addition des `Loins`.
2. La class `Locon`, avec la gestion de l'addition des `Locon` dans au niveau de la `Lofig`.
3. Ajouter le support de la duplication des `Locon` du modèle dans la `Loins` lors de l'instanciation.
4. La class `Losig` avec les différentes fonctions de connexion dans les `Loins` et les `Lofig`.

Question 4

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.

Question 5

En analysant le chaînage des objets entre eux, proposer les séquences de destruction appropriées pour chaque objet, puis les implanter dans les destructeurs.

Note: Normalement, la destruction d'un `Losig` devrait entraîner son retrait dans toutes les `Locon` ou il apparaît, dans un soucis de simplicité, nous ne le ferons pas.