Modélisation Objet -- MOBJ

Contents

TME 4 & 5 --Structure de Données Netlist 1
Programmation Modulaire 1
Configuration de `cmake` 1
Objets de la Structure de Données Netlist 2
Structure Générale des Fichiers 2
La Classe `Indentation` 3
Spécification de la Classe Term 3
Spécification de la Classe Net 6
Spécification de la Classe Instance 7
Méthodologie de Destruction 7
La Classe Cell 8
Travail à Réaliser 8
Question 1 8
Étape 1 8
Étape 2 9
Question 2 9
Question 3 11

TME 4 & 5 --Structure de Données Netlist

La structure de données NETLIST est conçue pour représenter le schéma de connexion d'un circuit 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).

Note

Abus de langage: Dans la suite on utilisera de façon interchangeable les termes suivants: `Cell` ou modèle, `Net` ou signal, `Term` ou connecteur.

Programmation Modulaire

Nous adoptons la règle de découpage classique d'un programme C++ complexe, c'est à dire «une paire de fichiers (`.h`, `.cpp`) par classe» de la structure.

Ce qui donne (fichiers fournis):

- `Indentation` : Indentation.h et Indentation.cpp.
- `Point` : Point.h et Point.cpp.
- `Node` : Node.h et Node.cpp.

- `Cell` : Cell.h et Cell.cpp.

Et à réaliser:

- `Term` : `Term.h` et `Term.cpp`.
- `Net` : `Net.h` et `Net.cpp`.
- `Instance` : `Instance.h` et `Instance.cpp`.

Ainsi qu'un programme de test: Main.cpp

Configuration de `cmake`

Fichier de configuration de `cmake` du TME45: CMakeLists.txt .

Note

Rappel la méthode pour organiser votre code et compiler est décrite dans procédure de compilation.

Objets de la Structure de Données Netlist

Structure Générale des Fichiers

Tous les objets de la structure de données devront être inclus dans un **`namespace`** appelé NETLIST. Concrètement cela signifie que tous vos fichiers d'en tête (`.h`), devront être organisés de la façon suivante:

```
#ifndef  NETLIST_CELL_H   // Le define *doit* être différent dans
#define  NETLIST_CELL_H   // chaque fichier (i.e. nom de classe).

// Les includes sont mis *à l'extérieur* du namespace.
#include <string>
#include "Indentation.h"

namespace Netlist {

  // Les *forward declarations* des autres objets de la
  // structure de données sont mis *à l'intérieur* du namespace.
  class Term;
  class Net;
  class Instance;

  class Cell {
    public:
       void  toXml ( std::ostream& );
    // Declaration of the class...
  };
}

#endif  // NETLIST_CELL_H
```

Et le module C++ associé (`.cpp`):

```
#include "Term.h"
#include "Net.h"
#include "Instance.h"
```

```
#include "Cell.h"

namespace Netlist {

  void  Cell::toXml ( ostream& stream ) {
     // Code of the toXml() method...
  }

}
```

Note

Un **namespace** peut être réouvert autant de fois que l'on veut. Ici il est ouvert à la fois dans le `.h` et dans le `.cpp`.

Note

Ordonnancement des includes: dans le `.cpp` les fichiers d'en tête doivent être inclus en partant des classes les plus simples vers les plus complexes. Ceci afin de ne pas rencontrer de problèmes liés à l'ordre des déclarations.

La Classe `Indentation`

L'implantation de la classe `Indentation` vous est fournie. Son API est la suivante:

- Une variable globale `indent`, qui, envoyée dans un flot de sortie affiche un nombre variable de tabulations.
- La taille de la tabulation d'`indent` peut être modifiée avec les opérateurs de pré-incrémentation (`++indent`), de post-incrémentation (`indent++`), de pré-décrémentation (`--indent`) et de post-décrémentation (`indent--`). Les *pré* sont effectives immédiatement (dans l'instruction courante) alors que les *post* ne prennent effet qu'à l'instruction *suivante*.

Exemple:

```
void  Cell::toXml ( ostream& stream )
{
  stream << indent++ << "<cell name=\"" << name_ << "\">\n";
  stream << indent   << "<Contents_of_the_Cell/>\n";
  stream << --indent << "</cell>\n";
}
```

Affichera:

```
<cell name="halfadder">
    <contents_of_the_Cell>
</cell>
```

Spécification de la Classe Term

Organisation choisie pour relier entre eux, `Node`, `Term` et `Net`.

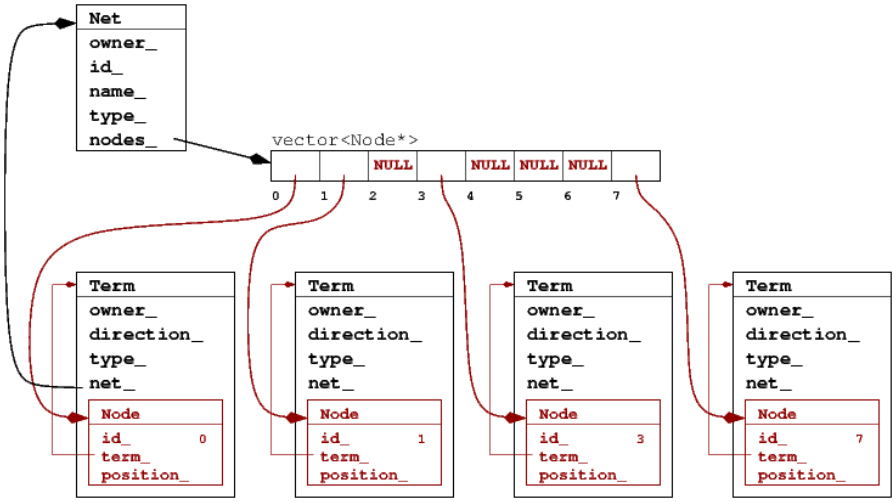

Figure 1: Figure 1 -- Relation entre `Net`, `Term` & `Node`.

Points importants du chaînage

- Un `Node` est associé de façon unique avec un `Term`.
- L'`id_` du `Node` est son index dans le tableau de `Node` (`nodes_`) du `Net`. Si le `Node` (et donc son `Term`) n'est associé à aucun `Net` son index vaut `Node::noid`.
- Dans le `Net`, le tableau `nodes_` peut contenir des cases *vides*, c'est à dire ne pointant vers aucun `Node`. Dans ce cas la valeur du pointeur stocké devra être mise à `NULL`.

Création du chaînage

- À la création d'un `Net` le tableau de `Node*`, `nodes_` est vide, c'est à dire qu'aucun `Term` n'y est relié.
- À la création d'un `Term`, l'`id_` de l'objet niché `node_` est initialisé à `Net::noid` signifiant qu'il n'est relié à aucun `Net`.
- La connexion d'un `Term` à un `Net` déterminé, c'est à dire la création des arcs du graphe, est répartie entre les méthodes `Term::setNet(Net*)` et `Net::add(Node*)`. Comme on peut le voir sur le schéma, la relation est cyclique: le `Term` a un pointeur vers le `Net` et ce dernier, *via* le tableau `nodes_` peut retrouver tous les `Term` qui lui sont connectés. On choisit de mettre le principal travail de mise à jour des pointeurs et des index dans `Net::add(Node*)`. C'est cette méthode qui gérera le tableau `nodes_` ainsi que les index des `Node`. Avant son rangement dans le tableau, un `Node` peut soit avoir un index déjà défini, ce qui indique la case *qu'il souhaite occuper*, soit valoir `Net::noid`, qui veut dire qu'on le mettra dans la première case libre du tableau (on l'ajoute en fin de tableau s'il est complet).

Elle définit deux types énumérés:

```
enum Type      { Internal=1, External=2 };
enum Direction { In=1, Out=2, Inout=3, Tristate=4, Transcv=5, Unknown=6 };
```

En relation avec ces `enum`, trois méthodes *statiques* seront ajoutées à la classe `Term` pour les convertir depuis/vers une chaîne de caractères (`string`).

```
static std::string  toString    ( Type );
static std::string  toString    ( Direction );
static Direction    toDirection ( std::string );
```

Elle a pour attributs:

```
void*          owner_;
std::string    name_;
Direction      direction_;
Type           type_;
Net*           net_;
Node           node_;
```

Note

Attribut node_: Le fait que l'attribut `node_` soit d'un type classe est parfaitement légal. C'est d'ailleurs une construction très courante en C++.

Un terminal (`Term`) peut appartenir à une `Cell` ou à une `Instance`, il aura donc deux constructeurs. La valeur de l'attribut `type_` sera déduite du constructeur appelé.

```
 Term ( Cell*     , const std::string& name, Direction );
 Term ( Instance*, const Term* modelTerm );
~Term ();
```

Note

Attribut type_ : Si le connecteur appartient à une `Cell`, son type sera `External`, inversement s'il appartient à une `Instance` il sera `Internal`.

Le constructeur est chargé de maintenir la cohérence de la structure de données. Concrètement, le nouveau connecteur (`Term`) devra être ajouté à la liste, soit de la `Cell`, soit de l'`Instance` pour laquelle il vient d'être créé. Inversement, le destructeur devra le retirer de son propriétaire.

Dans le cas d'un terminal d'`Instance`, il s'agit de dupliquer intégralement le `Term` du modèle dans l'instance. Il est proche (mais pas identique) à un constructeur par copie.

Prédicats et accesseurs:

```
inline bool                isInternal   () const;
inline bool                isExternal   () const;
inline const std::string&  getName      () const;
inline Node*               getNode      ();
inline Net*                getNet       () const;
inline Cell*               getCell      () const;
       Cell*               getOwnerCell () const;
inline Instance*           getInstance  () const;
inline Direction           getDirection () const;
inline Point               getPosition  () const;
inline Type                getType      () const;
```

Méthode `getNode()` : elle renvoie un *pointeur* sur l'attribut `node_`. Cela permettra d'accéder au modificateur `Node::setId()` (au TME6).

Fonctionnement des méthodes `getInstance()`, `getCell()` et `getOwnerCell()`:

Méthode	**`::getInstance()`**	**`::getCell()`**	**`::getOwnerCell()`**
External `Term`	`NULL`	`Cell` propriétaire	`Cell` propriétaire
Internal `Term`	`Instance` propriétaire	`NULL`	`Cell` propriétaire de *l'instance* propriétaire

Dit encore autrement, les méthodes `::getCell()` et `::getInstance()` renvoient l'objet auquel le `Term` appartient. Donc, dans le cas d'un `Term` appartenant à une instance `::getCell()` renverra `NULL`. En revanche, `::getOwnerCell()` renvoie la `Cell` *dans laquelle* l'objet se trouve, ce qui, dans le cas d'un `Term` d'instance est la `Cell` possédant celle-ci.

Dans le schéma suivant, **a** est un `Term` appartenant à la `Cell` *halfadder* (il est `External`) et **b** appartient à l'`Instance` *xor2_1* (il est `Internal`). Mais dans les deux cas, l'*ownerCell* est le *halfadder*.

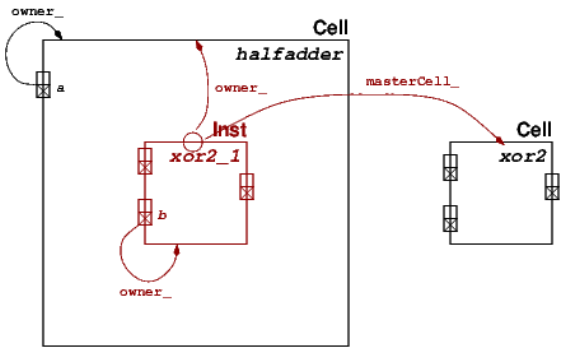

Figure 2: Figure 2 -- `Cell` et `Instance` `Term`.

Modificateurs:

```
        void  setNet       ( Net* );
        void  setNet       ( const std::string& );
inline void  setDirection ( Direction );
        void  setPosition  ( const Point& );
        void  setPosition  ( int x, int y );
```

Note

Méthode setNet(): C'est dans cette méthode que le chaînage entre `Net`, `Term` et `Node` sera réalisé (en appelant les méthodes appropriées du `Net`).
Si un pointeur `NULL` est passé en argument, cela signifie que le `Term` doit être déconnecté (s'il était relié à un `Net`).
Le `Net` peut être spécifié directement par un pointeur ou bien par son nom, c'est la deuxième surcharge de `setNet()`.

Spécification de la Classe Net

Tout comme le `Term`, le `Net` aura un type externe ou interne. Pour éviter de multiplier les constantes, on réutilisera l'`enum Type` `Term`.

Attributs. Un `Net` est unique au niveau d'une `Cell`, c'est une équipotentielle. Pour l'identifier, on ne se fie pas au nom, au lieu de cela on utilise un identificateur (numéro) unique, stocké dans l'attribut `id_`. Ces numéros sont gérés au niveau de la `Cell`.

```
Cell*                owner_;
std::string          name_;
unsigned int         id_;
Term::Type           type_;
std::vector<Node*>   nodes_;
```

Constructeurs & Destructeur. Ils devront gérer l'ajout et le retrait du `Net` au niveau de la `Cell`:

```
 Net     ( Cell*, const std::string&, Term::Type );
~Net     ();
```

Accesseurs:

```
Cell*                      getCell       () const;
const std::string&         getName       () const;
unsigned int               getId         () const;
Type                       getType       () const;
const std::vector<Node*>&  getNodes      () const;
size_t                     getFreeNodeId () const;
```

La méthode `getFreeNodeId()` renverra l'index de la première case libre dans le tableau `nodes_`. Si aucune case n'est libre, elle renverra la taille du tableau, c'est à dire l'index situé immédiatement après le dernier élément. Ce choix facilite l'écriture de la méthode `Net::add(Node*)`.

Modificateurs:

```
void  add    ( Node* );
bool  remove ( Node* );
```

Spécification de la Classe Instance

Attributs:

```
Cell*                owner_;
Cell*                masterCell_;
std::string          name_;
std::vector<Term*>   terms_;
Point                position_;
```

Constructeurs & Destructeur. Ils devront gérer l'ajout et le retrait de l'`Instance` au niveau de la `Cell`. Le constructeur devra *dupliquer* la liste des terminaux de la `Cell model` qu'il instancie. A l'inverse, le destructeur détruit ses terminaux.

```
Instance      ( Cell* owner, Cell* model, const std::string& );
~Instance      ();
```

Accesseurs:

```
const std::string&          getName       () const;
Cell*                       getMasterCell () const;
Cell*                       getCell       () const;
const std::vector<Term*>&   getTerms      () const;
Term*                       getTerm       ( const std::string& ) const;
Point                       getPosition   () const;
```

Modificateurs. `connect()` va associer le `Net` au terminal de nom `name` (s'il existe).

```
bool  connect       ( const std::string& name, Net* );
void  add           ( Term* );
void  remove        ( Term* );
void  setPosition   ( const Point& );
void  setPosition   ( int x, int y );
```

Méthodologie de Destruction

Si les constructeurs des objets mettent à jour le chaînage entre les différents composants d'une NETLIST, les destructeurs doivent aussi le faire. D'autre part, tous les objets ayant étés alloués dynamiquement en mémoire (i.e. par des appels à `new`) doivent être libérés par des `delete`. Mais où doit-on effectuer ces appels?

Nous allons raisonner en terme *d'appartenance*:

- Une `Cell` *possède* ses `Term`, `Net` et `Instance`, elle sera donc chargée de leur destrution. On va détruire ces composants dans l'ordre suivant:

  1. Les `Net`.
  2. Les `Instance`.
  3. Les `Term`.

  C'est grosso-modo, l'ordre inverse de création (ce n'est pas une règle absolue). Ce destructeur vous est fourni.

- Une `Instance` possède ses terminaux, elle doit donc les détruire dans son destructeur.
- Un `Net` connecte des `Term` ensemble, mais il ne les possède pas. Dans le destructeur, il va donc les détacher (déconnecter) mais pas les détruire.
- Lorsqu'un `Term` est détruit, si il est accroché à un `Net`, il doit s'en déconnecter dans son destructeur.

La Classe Cell

Methode	Description
`Cell(const std::string&)`	Constructeur (ajoute à la liste globale)
`~Cell()`	Destructeur. Retire de la liste globale, détruit tous ses sous-objets (`Instance`, `Term` & `Net`).
Accesseurs	
`std::vector<Instance*>& getInstances() const`	Retourne la table des instances
`std::vector<Term*>& getTerms() const`	Retourne la table des connecteurs (`Term`)
`std::vector<Net*>& getNets() const`	Retourne la table des signaux (`Net`)
`std::string& getName() const`	Le nom du modèle
`Instance* getInstance(const std::string&) const`	Recherche une instance, `NULL` si non trouvée
`Term* getTerm(const std::string&) const`	Recherche un connecteur (ou renvoie `NULL`
`Net* getNet(const std::string&) const`	Recherche un signal (ou renvoie `NULL`)
Modificateurs	
`void add(Instance*)`	Ajoute une nouvelle `Instance`
`void add(Term*)`	Ajoute un nouveau connecteur (`Term`)
`void add(Net*)`	Ajoute un nouveau signal (`Net`)
`void remove(Instance*)`	Retire une `Instance` (ne le désalloue pas)
`void remove(Term*)`	Retire un connecteur (ne le désalloue pas)
`void remove(Net*)`	Retire un signal (ne le désalloue pas)
`bool connect(const std::string& name, Net* net)`	Associe le connecteur `name` avec le signal `net`. Renvoie `true` en cas de succès.
`unsigned int newNetId()`	Renvoie un nouvel identificateur de signal. Il est garanti unique.
Méthodes statiques	
`Cell* find(const std::string&)`	Recherche un modèle (`Cell`) dans le tableau global des modèles.

Travail à Réaliser

Question 1

Implantation des classes `Term`, `Net` & `Instance`.

Plutôt que d'écrire la totalité des classes (déclarations & définitions) *puis* de compiler et supprimer les très nombreuses erreurs, il est préférable d'adopter une approche par étapes. L'idéal serait une approche *classe par classe* mais à cause de l'interdépendance de la structure de données, ce n'est pas possible.

Étape 1

Validation des fichiers d'en tête (`.h`). Écrire les déclarations des classes, avec leur attributs et leurs méthodes. Créer les (`.cpp`) avec les définitions des méthodes, mais dont on laissera le corps vide.

Compiler, corriger les erreurs dans les en-têtes.

A ce stade, on n'essaira pas d'exécuter le programme. Les méthodes ayant toutes un corps vide.

Étape 2

Implanter, classe par classe l'ensemble des définitions des fonctions membres.

Compiler et corriger après l'implantation de chaque classe.

Question 2

Pour visualiser la structure de données réalisée et la sauvegarder sur disque nous allons réaliser un petit driver XML.

Pour cela, implanter pour chaque classe, une fonction membre `toXml(ostream&)` qui affichera le contenu d'un objet sous forme XML. Le résultat de l'exécution du programme qui vous est fourni devra donner **exactement**:

```
Construction du modele <and2>.
<?xml version="1.0"?>
<cell name="and2">
  <terms>
    <term name="i0" direction="In"/>
    <term name="i1" direction="In"/>
    <term name="q" direction="Out"/>
  </terms>
  <instances>
  </instances>
  <nets>
  </nets>
</cell>

Construction du modele <or2>.
<?xml version="1.0"?>
<cell name="or2">
  <terms>
    <term name="i0" direction="In"/>
    <term name="i1" direction="In"/>
    <term name="q" direction="Out"/>
  </terms>
  <instances>
  </instances>
  <nets>
  </nets>
</cell>

Construction du modele <xor2>.
<?xml version="1.0"?>
<cell name="xor2">
  <terms>
    <term name="i0" direction="In"/>
    <term name="i1" direction="In"/>
    <term name="q" direction="Out"/>
  </terms>
  <instances>
  </instances>
  <nets>
```

```
  </nets>
</cell>

Construction du modele <halfadder>.
<?xml version="1.0"?>
<cell name="halfadder">
  <terms>
    <term name="a" direction="In"/>
    <term name="b" direction="In"/>
    <term name="sout" direction="Out"/>
    <term name="cout" direction="Out"/>
  </terms>
  <instances>
    <instance name="xor2_1" mastercell="xor2" x="0" y="0"/>
    <instance name="and2_1" mastercell="and2" x="0" y="0"/>
  </instances>
  <nets>
    <net name="a" type="External"/>
      <node term="a" id="0" x="0" y="0"/>
      <node term="i0" instance="xor2_1" id="1" x="0" y="0"/>
      <node term="i0" instance="and2_1" id="2" x="0" y="0"/>
    </net>
    <net name="b" type="External"/>
      <node term="b" id="0" x="0" y="0"/>
      <node term="i1" instance="xor2_1" id="1" x="0" y="0"/>
      <node term="i1" instance="and2_1" id="2" x="0" y="0"/>
    </net>
    <net name="sout" type="External"/>
      <node term="sout" id="0" x="0" y="0"/>
      <node term="q" instance="xor2_1" id="1" x="0" y="0"/>
    </net>
    <net name="cout" type="External"/>
      <node term="cout" id="0" x="0" y="0"/>
      <node term="q" instance="and2_1" id="1" x="0" y="0"/>
    </net>
  </nets>
</cell>
```

Question 3

Décrire le circuit `fulladder` présenté en cours:

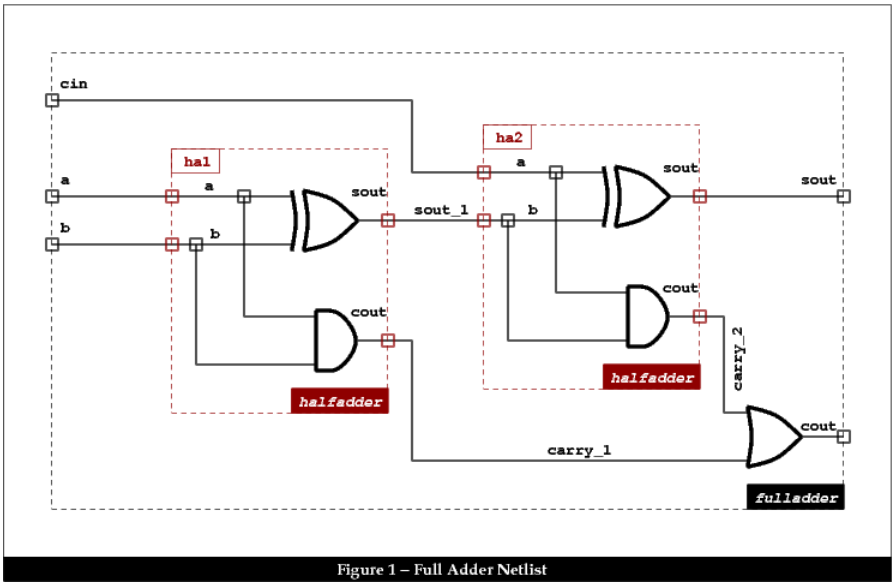

Figure 3: Figure 3 -- NETLIST du Fulladder.