Projet Informatique en langage C/C++

Simulateur de circuits numérique de type Netlist

Katell Morin-Allory, Xavier Lesage, Djamel Aouane

1 Introduction

Dans ce projet, on vous demande de développer simulateur d'un circuit numérique, représenté en tant que Netlist ou « réseau de portes ». Ce document est volontairement très générique : en tant que Projet, on vous demande de réaliser toutes les vraies étapes du Génie Logiciel, dont le codage n'est qu'une partie

2 Sujet détaillé

2.1 Organisation

Un Simulateur est un programme capable de prendre en entrée un Fichier de Description Matérielle (Hardware Description Language) représentant le circuit cible, de lui appliquer une série de stimuli (le testbench) et de sauvegarder les résultats, i.e. les sorties du circuit. Le cas échéant, il peut aussi afficher à l'écran les signaux pour aider le debug et la validation. Un exemple est le logiciel Modelsim qui simule plusieurs HDL, dont notamment le VHDL et Verilog.
Dans un souci de simplicité, pour ce projet on va considérer un circuit cible de type Netlist, i.e. un graph de porte booléenne. Ils existent plusieurs formats dédiés, comme par exemple le EDIF, mais ils présentent une série de spécificités et difficulté liés à leur utilisation dans le flot de synthèse, et qui ne sont pas intéressantes pour ce qui nous concerne. On va donc vous proposer d'utiliser en tant qu'entrée le format DOT. C'est un format Linux dédiée aux graphes, et a le double avantage d'être très simple et de pouvoir être facilement visualisé, par exemple grâce à l'utilitaire graphwiz.

Vos premières tâches vont donc être :

- **Comprendre** la syntaxe et son utilisation dans ce contexte, détaillé en Annexe ;
- **Spécifier** une structure de données internes capable de représenter les circuits
- **Implémenter** un parseur capable de transformer un fichier DOT dans votre structure de données.

Ces trois phases sont à la base de tout effort de Génie Logiciel.

Une fois le circuit chargé en mémoire, il vous faudra appliquer l'algorithme de simulation expliqué dans le tutorat. Pour cette phase, vous allez utiliser le format Wavedrom (www.wavedrom.com et en Annexe) avec lequel vous êtes déjà familiers.
NB : vous n'êtes pas forcé d'utiliser tout de suite un fichier d'entrée, vous pouvez commencer par remplir à la main les structures de données internes, et passer à Wavedrom pour les cas plus complexe. Faites bien attention à suivre les trois phases Compréhension/Spécification/Implémentation.

En dernière étape, il vous faudra sauvegarder le résultat des sorties de simulation. Pour cette étape aussi vous allez utiliser Wavedrom

2.2 Fonctionnalités demandées

Dans ce projet, on vous demande de suivre une approche incrémentale de fonctionnalités, en partant des plus simples aux plus compliquées. On vous demande donc le support dans l'ordre de ces ensembles :

- Portes logiques de base à 1 ou 2 entrées (NOT, AND, OR, XOR, etc..)
- Multiplexeurs 2→1;
- Portes logiques de base et Multiplexeurs > 2 entrées
- Logique Séquentielle : bascule (Flip/Flop) et registres

2.3 Exemples de fichiers d'entrée

Les spécifications détaillées du format DOT sont données en annexe et sur internet. Dans cette section, on vous donne quelques exemples de son utilisation dans ce projet.
Porte AND2

```
digraph test {

I1 [label = "INPUT"];
I2 [label = "INPUT"];

GATE [label = "AND2" ];

O [label = "OUTPUT"];

I1 -> GATE -> O;
I2 -> GATE;

}
```

Multiplexeur 2→1

```
digraph test {

I1 [label = "INPUT"];
I2 [label = "INPUT"];
I3 [label = "INPUT"];

M [label = "MUX2" sel = "I3"];

O [label = "OUTPUT"];

I1 -> M -> O;
I2 -> M;

}
```

Flip-Flop (Bascule)

```
digraph test {

I [label = "INPUT"];

Bascule [label = "FF"];

O [label = "OUTPUT"];

I -> Bascule -> O;

}
```

3 Bonne pratiques du génie logiciel

Ils existent plusieurs stratégies de Génie Logiciel, chacune avec ses points forts et point faibles. Pour ce projet, on vous conseil certaines pratiques, mais rien n'est imposé : ce sera à vous de choisir si et lesquelles appliquer.

3.1 Gestion des sources

On vous conseil d'utiliser un gestionnaire de sources tel que GIT pour pouvoir travailler facilement en binôme et garder une trace de votre développement.

3.2 Développement guidé par le test

Dans un projet informatique, la qualité du code a un impact fondamental sur la qualité finale du programme. On vous conseille donc d'appliquer la technique dite de Tests Unitaires : à chaque fois qu'on ajoute un élément (classe, methode, fonction, etc…), une batterie de test lui est associée pour vérifier son fonctionnement. Ces tests peuvent être appelés à tout moment pour vérifier la « non régression » et donc assurer la qualité finale.
On vous propose l'utilisation de la suite « ctest « .

4 Aspects Pratiques

4.1 Binômes

Le projet sera mené en binôme. La répartition du travail interne sera décidée par vous-même, et clairement indiquée et justifiée dans le rapport. Les binômes devront être formés dès que possible au plus tard le **27 Novembre**. Merci de nous informer au plus tôt de la constitution des binômes par email à l'adresse suivante :

katell.morin-allory@univ-grenoble-alpes.fr

4.2 Encadrants

Les étudiants bénéficient de l'aide des encadrants notamment pour:

l'organisation du projet

- l'analyse du problème
- la conception du programme
- la programmation en langage C/C++
- l'environnement de développement (make, autres outils gnu, etc)
- toute autre question liée au projet...

Les enseignants ne sont pas là pour concevoir le programme, programmer ou corriger les bugs à la place des étudiants. Si les enseignants l'estiment nécessaire, ils peuvent néanmoins débloquer les groupes en difficulté. Les questions posées par les étudiants doivent être précises et réfléchies.

4.3 Aspects matérielles

Pour ce projet, le travail s'effectuera sur des PC avec un système d'exploitation Linux. Tous les outils nécessaire (compilateur C++, débogueur, etc.) seront installés sur les stations de travail de PHELMA.

Vous pouvez naturellement travailler sur votre ordinateur personnel si vous en possédez un, mais attention le projet final doit fonctionner correctement à PHELMA, sous Linux, le jour de l'examen! Soyez prudent si vous développez sous Windows, le portage n'est pas toujours automatique...

4.4 Site web du projet

Il est disponible sur chamilo. Vous y trouverez le sujet et toutes informations que nous jugerons nécessaires. Vous pourrez aussi utiliser le forum pour poser des questions.

4.5 Déroulement du projet dans le temps et évaluation

La phase d'analyse est prévue pour une/deux semaines (**semaine 48-49**). Des créneaux réguliers seront réservés avec les encadrants. La phase d'analyse se terminera par la remise d'un rapport d'analyse au plus tard le **lundi 4 Décembre**. Ce rapport donnera lieu à une note qui interviendra pour 10% dans la note finale du projet.

La phase développement est prévue pour durer le reste du projet. A l'issue de cette phase, vous aller rendre une version de votre projet, qui sera utilisée lors de l'examen, ainsi qu'un rapport finale. Ce rapport donnera lieu à une note qui interviendra pour 30% dans la note finale du projet

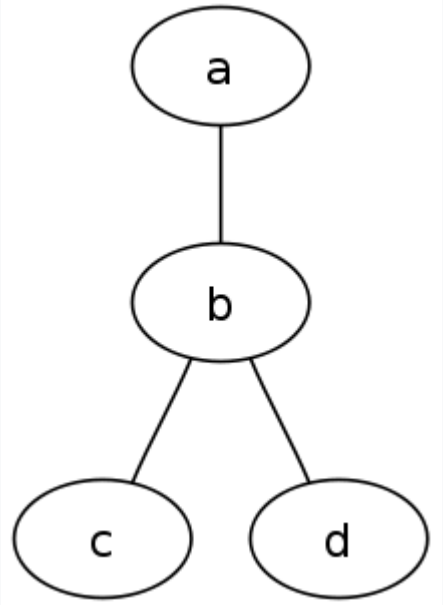
L'examen final sera organisé semaine 51. Et en janvier, Il comptera pour 60% de la note. Il sera organisé en deux phases : lors de la première phase vendredi semaine 51, les étudiants dérouleront une série de test, et corrigerons au mieux leur programme. Lors de la deuxième phase en janvier, chaque trinôme sera seul avec un des enseignants du projet. Une phase de démonstration et de test aura lieu. Ces deux parties compteront pour respectivement 60% dans la note finale attribuée au projet.

5 The DOT format (from Wikipedia)

DOT is a graph description language. DOT graphs are typically files with the filename extension *gv* or *dot*. The extension *gv* is preferred, to avoid confusion with the extension *dot* used by versions of Microsoft Word before 2007.[1]

Various programs can process DOT files. Some, such as *dot*, *neato*, *twopi*, *circo*, *fdp*, and *sfdp*, can read a DOT file and render it in graphical form. Others, such as *gvpr*, *gc*, *acyclic*, *ccomps*, *sccmap*, and *tred*, read DOT files and perform calculations on the represented graph. Finally, others, such as *lefty*, *dotty*, and *grappa*, provide an interactive interface. The *GVedit* tool combines a text editor with noninteractive image viewer. Most programs are part of the Graphviz package or use it internally.

Undirected graphs

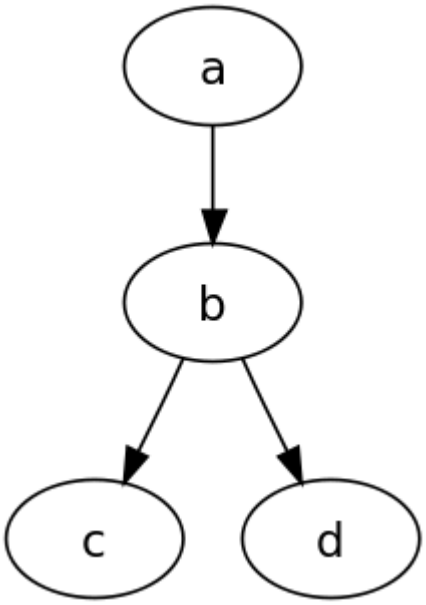


An undirected graph

At its simplest, DOT can be used to describe an undirected graph. An undirected graph shows simple relations between objects, such as friendship between people. The *graph* keyword is used to begin a new graph, and nodes are described within curly braces. A double-hyphen (--) is used to show relations between the nodes.

```
// The graph name and the semicolons are optional
graph graphname {
    a -- b -- c;
    b -- d;
}
```

Directed graphs

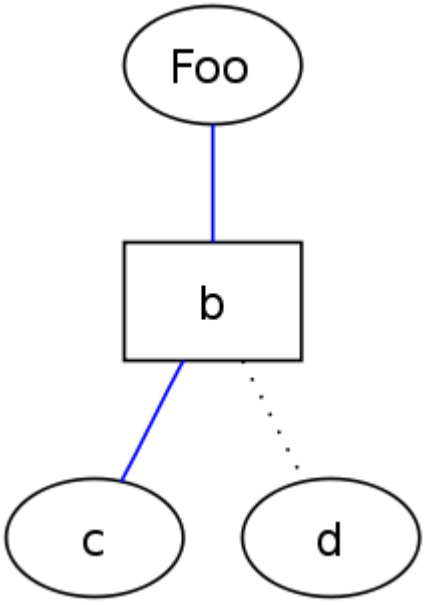


A directed graph

Similar to undirected graphs, DOT can describe directed graphs, such as flowcharts and dependency trees. The syntax is the same as for undirected graphs, except the *digraph* keyword is used to begin the graph, and an arrow (->) is used to show relationships between nodes.

```
digraph graphname {
    a -> b -> c;
    b -> d;
}
```

Attribute



A graph with attributes

Various attributes can be applied to graphs, nodes and edges in DOT files. [2] These attributes can control aspects such as color, shape, and line styles. For nodes and edges, one or more attribute–value pairs are placed in square brackets ([]) after a statement and before the semicolon (which is optional).

Graph attributes are specified as direct attribute–value pairs under the graph element, where multiple attributes are separated by a comma or using multiple sets of square brackets, while node attributes are placed after a statement containing only the name of the node, but not the relations between the dots.

```
graph graphname {
    // This attribute applies to the graph itself
    size="1,1";
    // The label attribute can be used to change the label of a node
    a [label="Foo"];
    // Here, the node shape is changed.
    b [shape=box];
    // These edges both have different line properties
    a -- b -- c [color=blue];
    b -- d [style=dotted];
    // [style=invis] hides a node.
  }
```

Annex B: Wavedrom

Hitchhiker's Guide to the WaveDrom

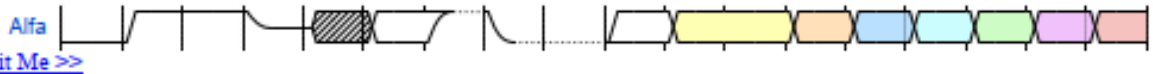
WaveDrom is a JavaScript application. WaveJSON is a format that describes Digital Timing Diagrams. WaveDrom ... the diagrams directly inside the browser. Element "signal" is an array of WaveLanes. Each WaveLane has two mandato... fields: "name" and "wave".

Step 1. The Signal

Lets start with a quick example. Following code will create 1-bit signal named "Alfa" that changes its state over time.

```
{ signal: [{ name: "Alfa", wave: "01.zx=ud.23.456789" }] }
```

Every character in the "wave" string represents a single time period. Symbol "." extends previous state for one more period. Here is how it looks:



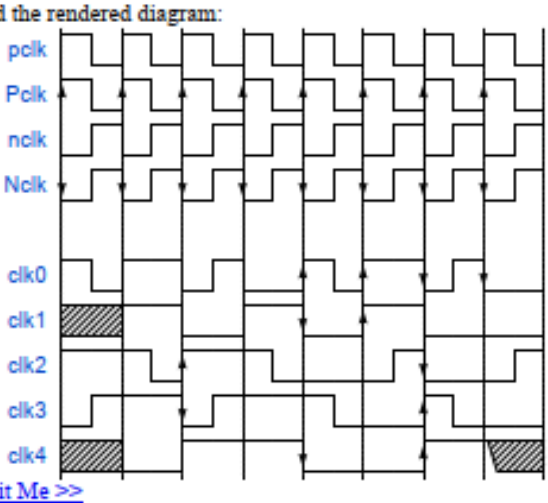
Edit Me >>

Step 2. Adding Clock

Digital clock is a special type of signal. It changes twice per time period and can have positive or negative polarity. It also can have an optional marker on the working edge. The clock's blocks can be mixed with other signal states to create the clock gating effects. Here is the code:

```
{ signal: [
  { name: "pclk", wave: 'p.......' },
  { name: "Pclk", wave: 'P.......' },
  { name: "nclk", wave: 'n.......' },
  { name: "Nclk", wave: 'N.......' },
  {},
  { name: 'clk0', wave: 'phnlPHNL' },
  { name: 'clk1', wave: 'xhlhLHl.' },
  { name: 'clk2', wave: 'hpHplnLn' },
  { name: 'clk3', wave: 'nhNhplPl' },
  { name: 'clk4', wave: 'xlh.L.Hx' },
]}
```

and the rendered diagram:



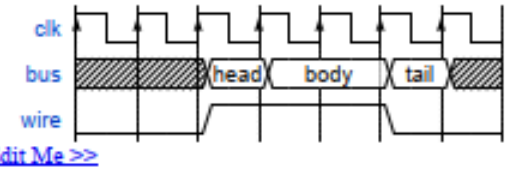
Edit Me >>

Step 3. Putting all together

Typical timing diagram would have the clock and signals (wires). Multi-bit signals will try to grab the labels from "data" array.

```
{ signal: [
```

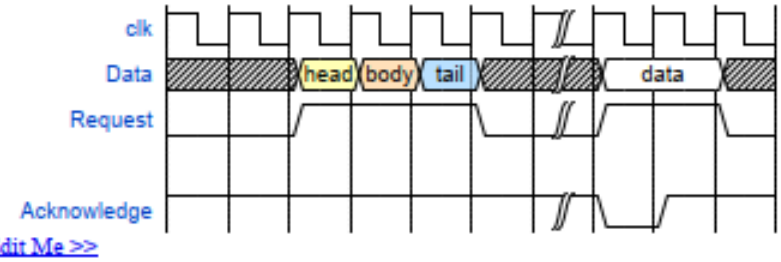
```
    { name: "clk",  wave: "P......" },
    { name: "bus",  wave: "x.==.=x", data: ["head", "body", "tail", "data"] },
    { name: "wire", wave: "0.1..0." }
  ]}
```



Edit Me >>

Step 4. Spacers and Gaps

```
{ signal: [
    { name: "clk",          wave: "p.....|..." },
    { name: "Data",         wave: "x.345x|=.x", data: ["head", "body", "tail", "data"
    { name: "Request",      wave: "0.1..0|1.0" },
    {},
    { name: "Acknowledge", wave: "1.....|01." }
]}
```



Edit Me >>

From https://wavedrom.com/tutorial.html