PROJET C

Rapport d’analyse

ROMAIN FABRE – ANTHONY JANIN

2A CSI  2017-2018

Table des matières

[Introduction 3](#_Toc474519016)

[TP n°1 4](#_Toc474519017)

[Unité Arithmétique Logique 4](#_Toc474519018)

[Explication du .vhd : 4](#_Toc474519019)

[Explication du testbench : 6](#_Toc474519020)

[Création d’un compteur UP/DOWN 9](#_Toc474519021)

[Explication du .vhd 9](#_Toc474519022)

[Explication du testbench 10](#_Toc474519023)

[TP n°2 12](#_Toc474519024)

[Registre à décalage 12](#_Toc474519025)

[Explication du .vhd 13](#_Toc474519026)

[Explication du testbench 14](#_Toc474519027)

[TP3 16](#_Toc474519028)

[Explication du .vhd 17](#_Toc474519029)

[Explication du testbench 18](#_Toc474519030)

[Conclusion 20](#_Toc474519031)

# Introduction

Un langage de description matériel est un langage informatique permettant la description d’un circuit électronique. Celui-ci peut décrire les fonctions réalisées par le circuit (description comportementale) ou les portes logiques utilisées par le circuit (description structurelle). Il est possible d'observer le fonctionnement d'un circuit électronique modélisé dans un langage de description grâce à la simulation. À la différence d'un langage de programmation logiciel, la syntaxe et la sémantique d'un HDL incluent des notations explicites pour exprimer le temps et le parallélisme qui sont les attributs principaux du matériel. Un synthétiseur logique permet de transformer un circuit décrit dans un langage de description de matériel en une netlist.

Ce compte rendu traitera des TP n°1, 2 et 3 de la première série de TP de VHDL. Ces trois TP avaient plusieurs objectifs, cités ci-dessous :

TP n°1 :

* Comprendre et compiler une description VHDL et son testbench
* Apprendre à modifier un composant VHDL purement combinatoire et son

testbench

* Apprendre à écrire un composant séquentiel en VHDL (compteur)
* Simuler et observer des traces de simulation, prise en main des outils

TP n°2 :

* Ecrire la description VHDL d’un registre à décalage et la tester
* Commencer la synthèse d’un circuit combinatoire, prise en main des outils

TP n°3 :

* Ecrire et tester un automate (FSM)
* Faire une synthèse de FSM afin d’aboutir à une implémentation physique

# TP n°1

## Unité Arithmétique Logique

### Explication du .vhd :

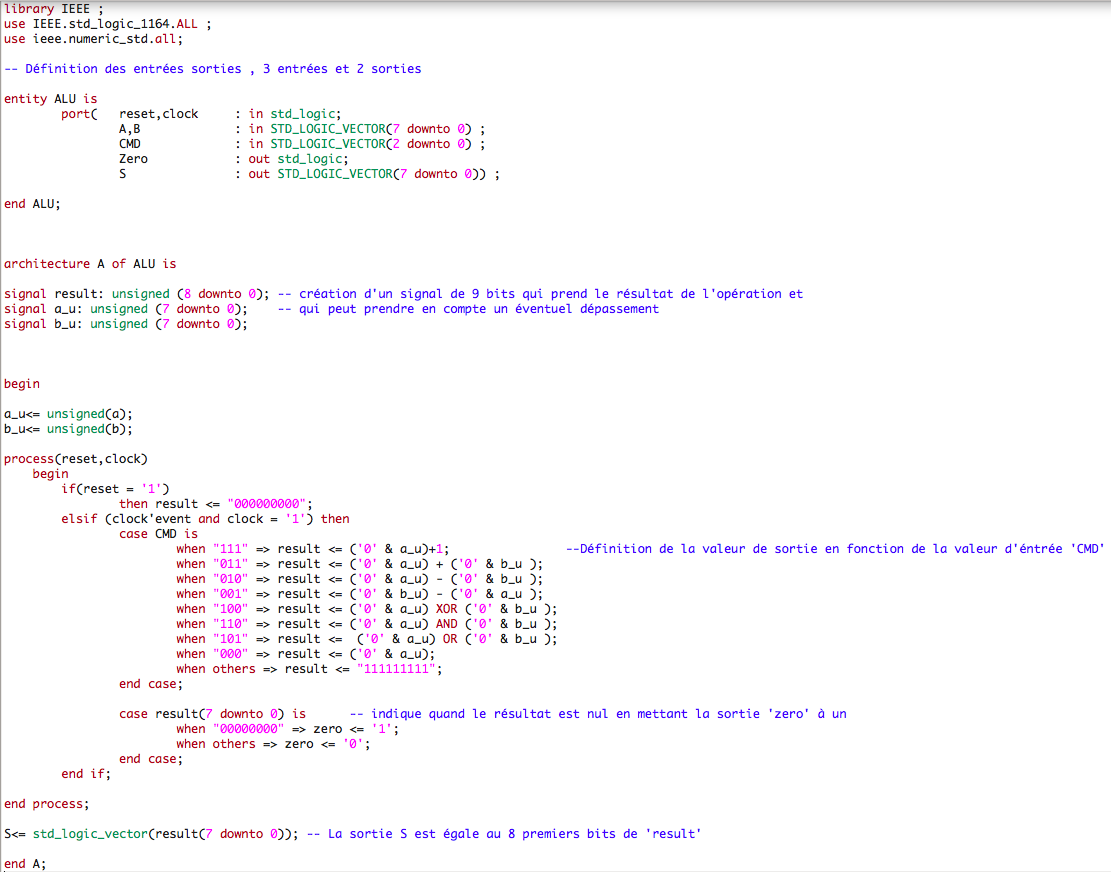


Figure 1: Code commenté de ALU.vhd

Le code ci dessus décrit une unité arithmétique logique.

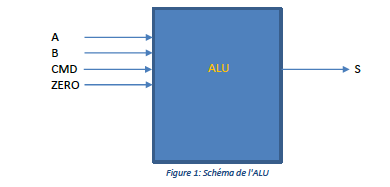


Figure : Schéma d'entrées/sorties de l'ALU

On commence par décrire les entrées sorties du composant, ici : une clock, un reset, deux entrées A et B, un signal de commande ‘CMD’, deux signaux de sorties ‘S’et ‘Zero’.

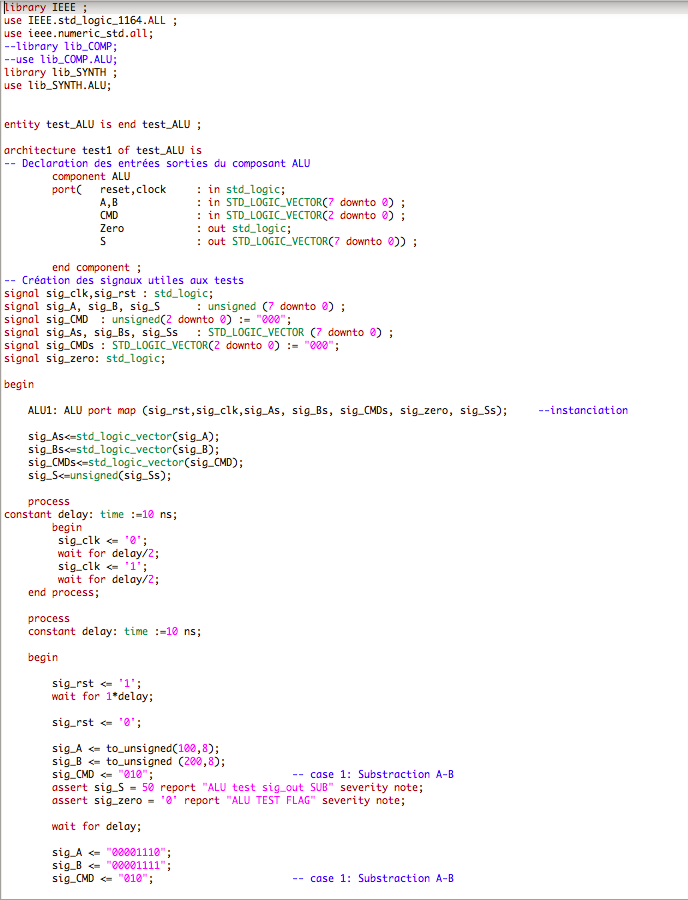
On déclare ensuite l’architecture A du composant ALU : déclaration d’un signal ‘result’ où sera stocké la valeur de sortie avant d’être assignée à S.

On vient ensuite créer un process ayant une liste de sensibilité composée du reset et de la clock. Le premier if() permet de rendre le ‘reset’ prioritaire sur toute les autres actions : si le reset est à ‘1’, aucune action n’est réalisée par le composant. Ensuite, on trouve un elseif qui rend le process sensible aux fronts montants d’horloge.

On trouve ensuite un case qui va décrire et définir la valeur de sortie en fonction de la valeur du signal d’entrée ‘CMD’. De ce fait, la description de l’ALU prend tout son sens, car l’utilisateur va pouvoir choisir l’opération à effectuer grâce au signal de commande. De plus, lorsque que la sortie est égale à ‘0’, le signal ‘Zero’ est mis à la valeur ‘0’.

Après le process, on attribue la valeur de ‘result’ à la sortie ‘S’.

### Explication du testbench :



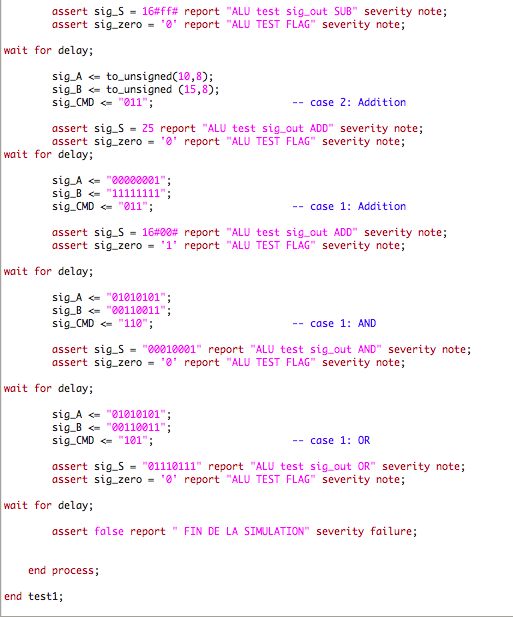
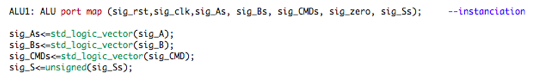


Figure 3 : Code commenté du test bench de l'ALU

On cherche à tester le composant grâce à un programme émulant un testbench.

Dans ce fichier, on commence par créer des signaux utiles aux tests. On map ensuite les signaux sur les bons ports du composant :



C’est ensuite dans le process que l’on va créer la clock et les différentes phases du test du composant.

Le premier process de ce programme permet de créer une clock de 10ns.

Le deuxième process traite des différentes phases de tests du composant. On commence par rendre le reset actif afin de vérifier que tout les signaux sont remis à leurs valeurs initiales. Toutes les autres lignes sont dédiées aux différentes phases de tests, addition, soustraction, ET logique, OU logique, XOR logique. Nous avons rajouté au code des assert qui permettent d’avoir des indications sur l’avancement du code lors de la simulation. La sévérité des assert est « note » pour que l’assert nous affiche seulement un message et ne stoppe pas le programme.



Figure : Chronogramme de la simulation de l'ALU

Pour vérifier que notre composant (c.-à-d. notre description VHDL) fonctionne correctement, il suffit de vérifier que le résultat obtenu en sortie corresponde à l’opération décidée par la commande. On remarque que le résultat est cohérent avec la commande employée.

Par exemple : l’addition (CMD = 011) réalise l’opération suivante : 1010 + 1111 = 11001 - OK

La simulation s’arrête après cinq délais de 10ns (chaque délai correspondant à un test), soit au bout de 50ns, et renvoi un message « Fin de la simulation » généré par une *assert* (indiquant dans ce cas le bon déroulement de la simulation).

Enfin, dans le but de tester les limites de cette description, le TP nous propose de réaliser l’opération suivante : 100-200. Il faut donc modifier les valeurs d’A et de B lorsque la commande correspondante est activé. En faisant 100-200, les variables étant du type *unsigned* la valeur affichée par le chronogramme n’est pas cohérente, cependant en observant la simulation, en fonction de l’interpration du MSB que l’on adopte (c-à-d si l’on considère le mot *signed* ou *unsigned*), on retrouve un résultat correct : -100.

À noter que le résultat de l’opération 100-200 n’est pas égale au résultat attendu par l’*assert*, ce qui a pour effet de générer un message d’erreur. On peut donc conclure sur l’efficacité du banc de test proposé.

Le programme proposé a donc été réalisé intelligemment et facile à interpréter étant donné que l’on connait la valeur attendue en sortie selon la commande, tachons de nous en inspirer pour la suite.

## Création d’un compteur UP/DOWN

Dans la suite de ce TP, il s’agit de réaliser un simple compteur, en appliquant ce qui a été compris précédemment concernant l’ALU. Ce compteur UP/DOWN sur 8 bits peut être décrit schématiquement ci-dessous :

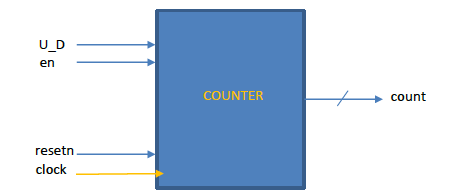


Figure 5 : Schéma du compteur UP/DOWN

* Resetn : permet de réinitialiser le compteur
* En : permet d’activer ou non le compteur
* Clock : entrée du signal d’horloge
* U\_D : entrée permettant d’activer le comptage ou le décomptage
* Count : sortie sur 9 bit où se trouve la valeur du compteur

### Explication du .vhd

Vous trouverez ci dessous le code commenté du .vhd pour que vous puissiez comprendre comment est organisé le programme.

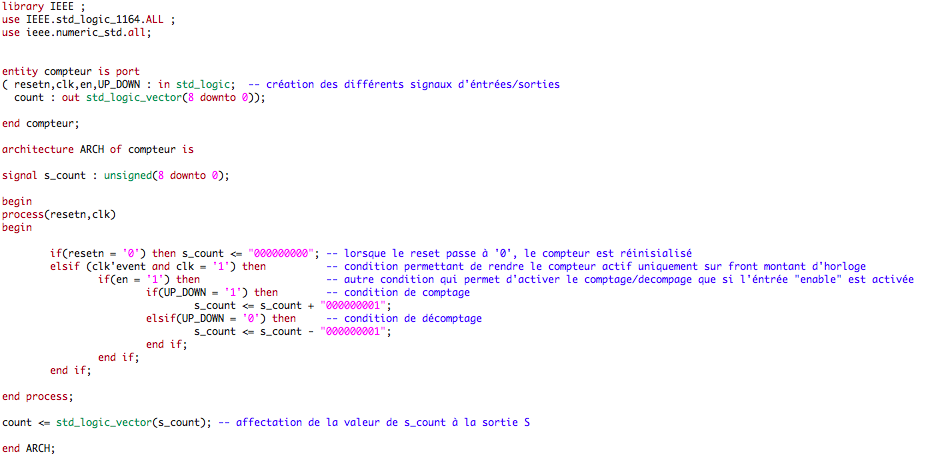


Figure 6 : Code commenté de UP/DOWN.vhd

### Explication du testbench

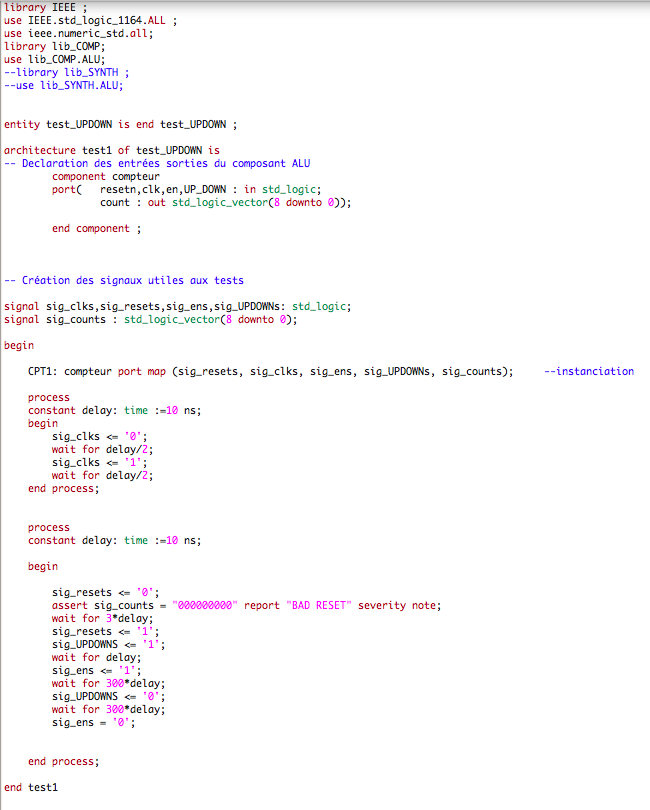


Figure : Code du testbench du compteur UP/DOWN

Tout comme pour l’ALU, le premier process consiste à créer une horloge de période 10ns.

Le second sert à tester le compteur : on commence par activer le reset en le mettant à ‘0’, on vérifie si le reset est fonctionnel grâce à un assert qui nous affiche « BAD RESET » si la valeur du compteur n’est pas ‘00000000’. On attend ensuite 3 périodes pour lâcher le reset, pour mettre le composant en mode comptage grâce à l’entrée « UPDOWNS » et pour activer le comptage grâce à l’entrée « en ». ON laisse le compteur travailler pendant 300 période avant le le passer un décomptage, puis après 300 période, on le désactive.

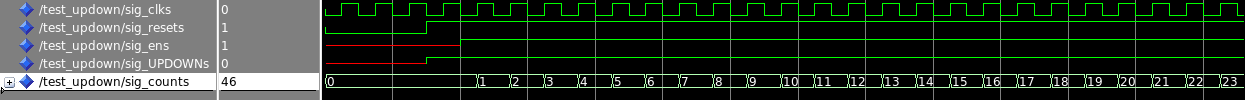


Figure : Chronogramme du composant en comptage

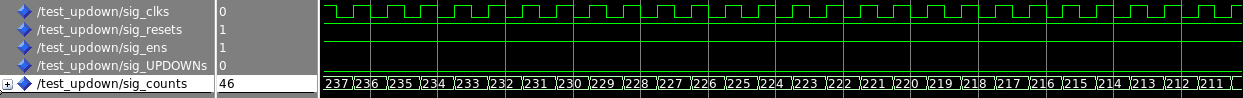


Figure : Chronogramme du composant en décomptage

ON voit ci dessus les deux chronogrammes obtenus suite à la simulation du testbench. On voit que le compteur fonctionne sur la figure 8, et que le décompteur fonctionne sur la figure 9. Le compteur étant un compteur 8bits, il nous permet d’aller jusqu’à la valeur 255. Le composant fonctionne donc correctement !

# TP n°2

## Registre à décalage

À présent, nous allons nous intéresser à la réalisation d’un registre à décalage sur 11 bits, dont les attentes du cahier des charges peuvent être représenter schématiquement ci-dessous :

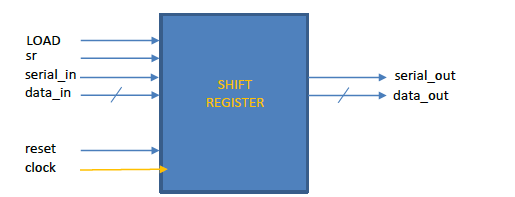
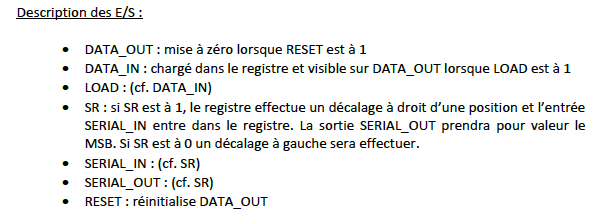
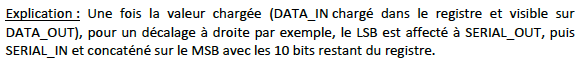


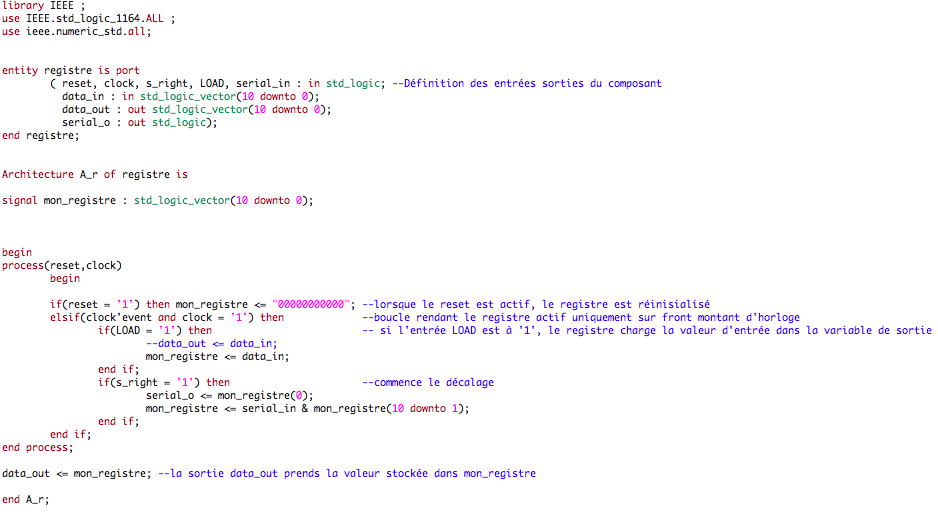
Figure : Schéma du registre à décalage





### Explication du .vhd

Figure : Code commenté du registre à décalage

Vous trouverez ci dessous le code commenté du .vhd pour que vous puissiez comprendre comment est organisé le programme 

### Explication du testbench



Figure : Code du banc de test du registre à décalage

Le premier process sert (encore et toujours) à créer un signal d’horloge.

Le deuxième commence par charger une valeur dans la data\_in, puis active le reset pendant 20ns, et le relâche. L’entrée « load » est ensuite mise à 1 pour charger la valeur présente sur data\_indans le registre de data\_out. Un 0 logique est appliqué sur l’entrée serial\_in, puis le « sr » est passé à ’1’ pour débuter le décalage à droit. ON laisse le registre agir pendant 3 périodes, puis on relâche le « sr ».

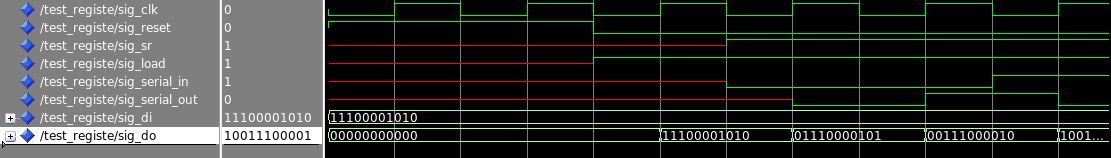


Figure 12 : Chronogramme du testbench du registre à décalage

Ci-dessus, le chronogramme obtenu après simulation du code du registre à décalage. On remarque que le registre est entièrement fonctionnel et qu’il répond parfaitement aux exigences du cahier des charges.

# TP3

Le but de ce TP est de réaliser un circuit FSM de type arbitre, fonctionnant en style tourniquet. Cet arbitre contrôle l’accès de 3 composants de type processeurs (P1, P2,P3) à une ressource partagée de type mémoire (M), voici la machine à états à réaliser :

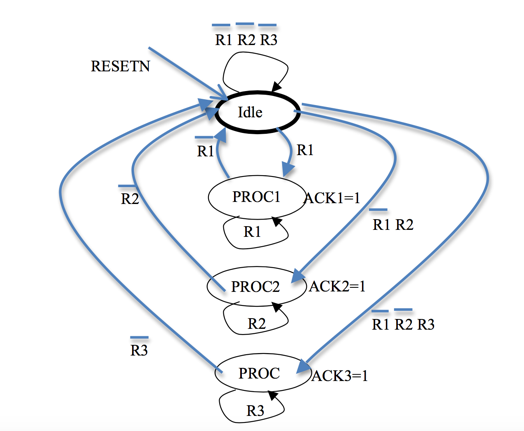


Figure 13 : Synthèse de la machine à états

Tout d’abord nous commençons par écrire la description VHDL de cet arbitre :

### Explication du .vhd

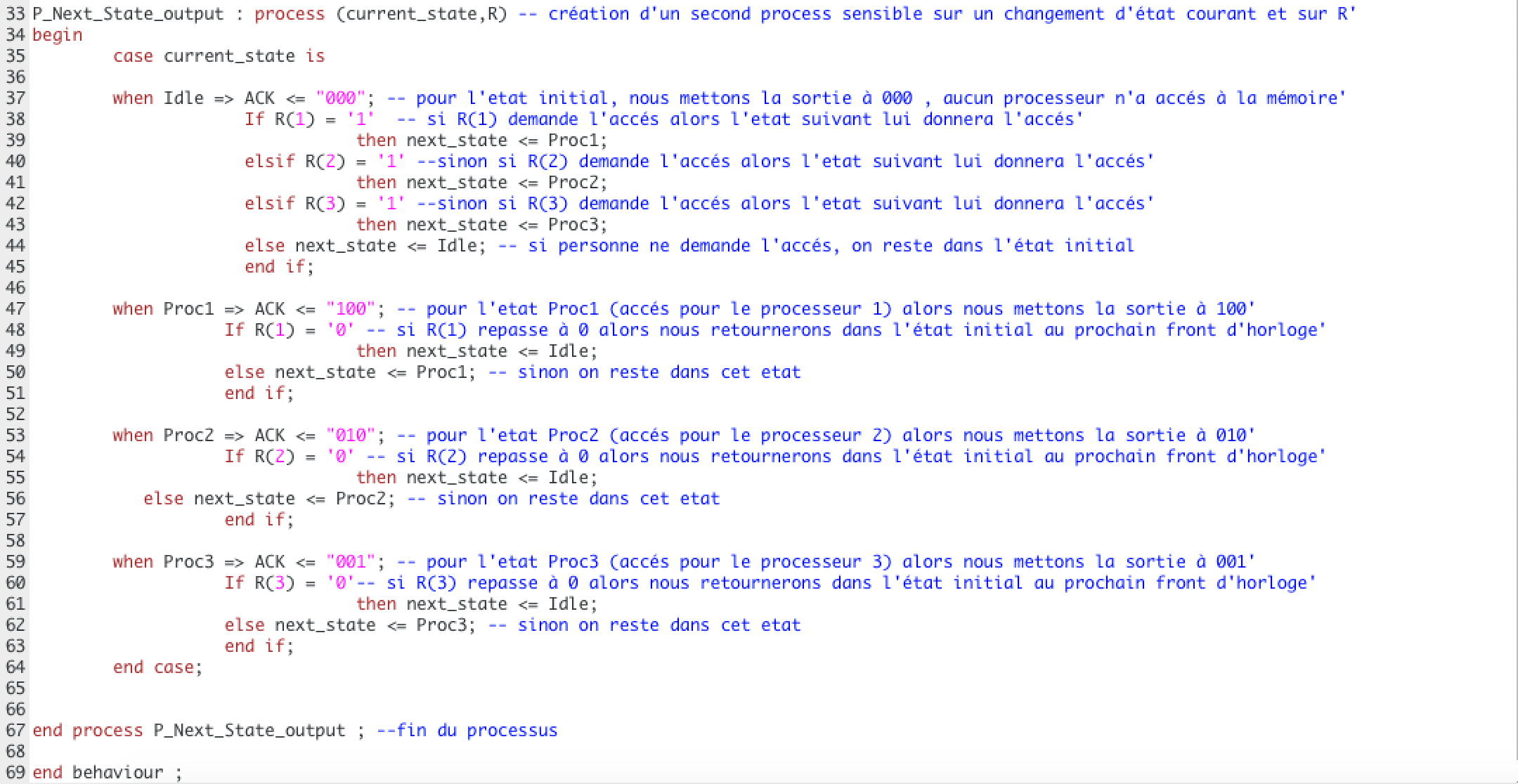
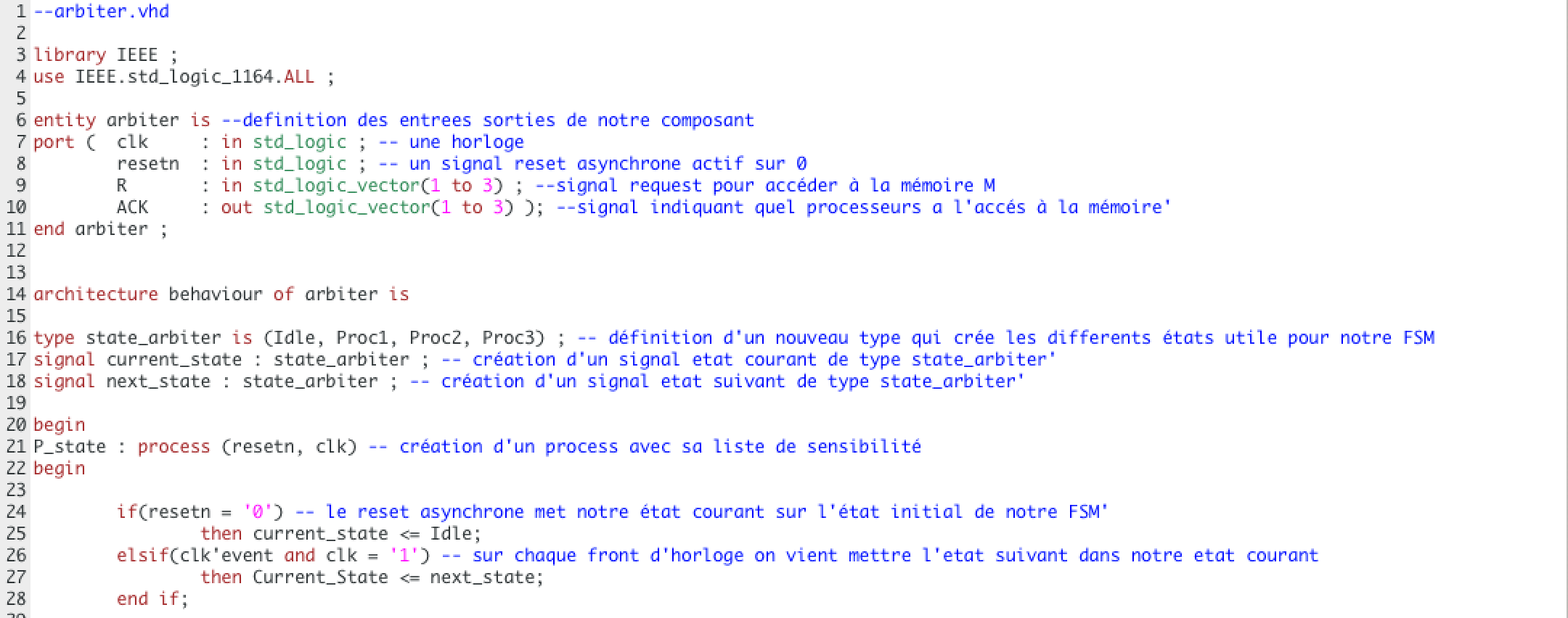


Figure 14 : Code commenté de la FSM

### Explication du testbench

Maintenant que le code de la FSM a été réalisé, il faut développer un testbench permettant de parcourir tous les cas possibles pour voir s’il y a ou non des problèmes dans notre programme.

Pour réaliser ce testbench nous créons un nouveau fichier .vhd :

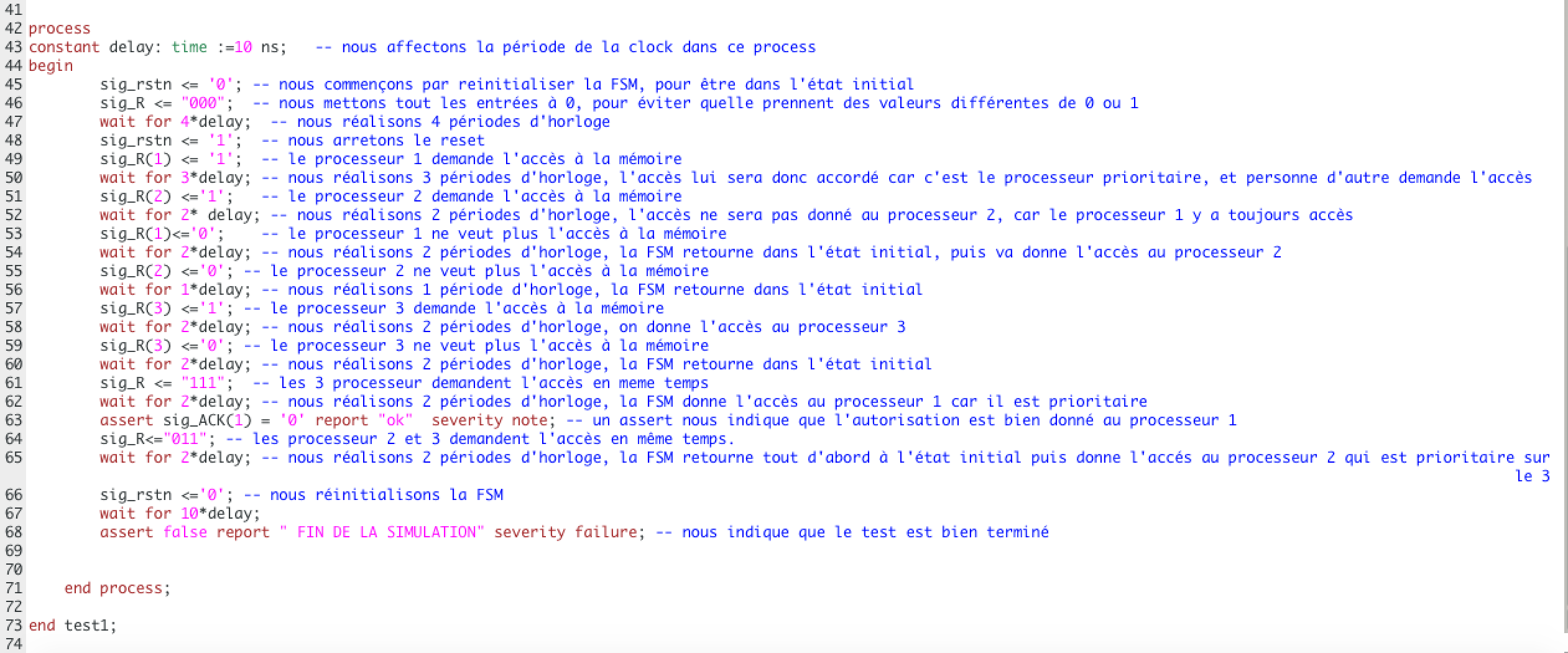
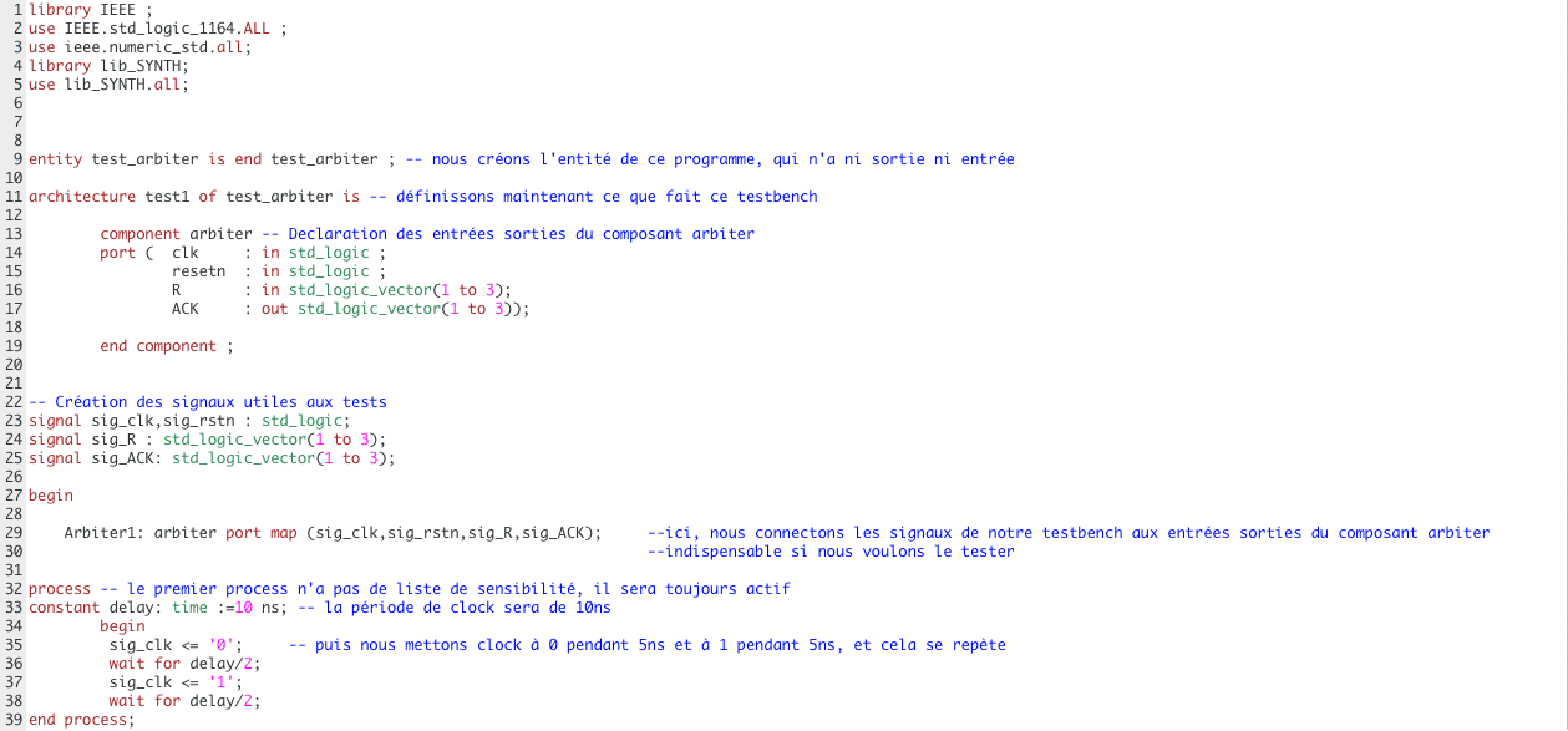


Figure 15 : Code commenté du testbench de la FSM

Maintenant que l’arbitre et le tesbench sont réalisé, nous pouvons tester la FSM grâce au chronogramme (sur Vsim).

Figure 16 : Chronogramme de la FSM

010

Le chronogramme traduit exactement ce que nous avons voulu faire grâce au testbench, l’arbiter est donc fonctionnel et répond à ce qui était demandé.

Nous effectuons maintenant la synthèse de notre arbitre avec des codages binaires et one hot voici ce que nous obtenons (pour un FPGA 3S200ft256) :

Pour le one hot : Pour le binaire :

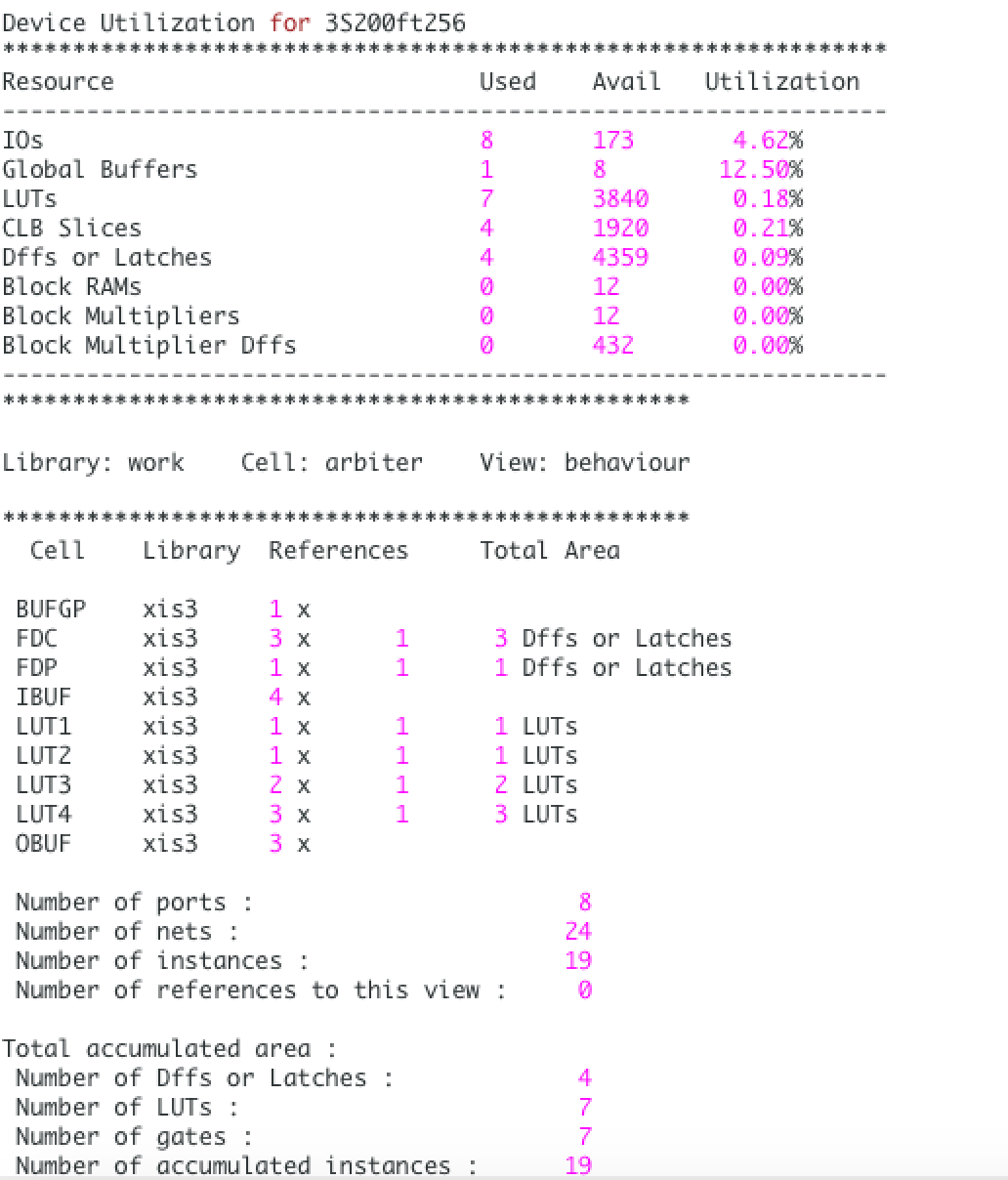
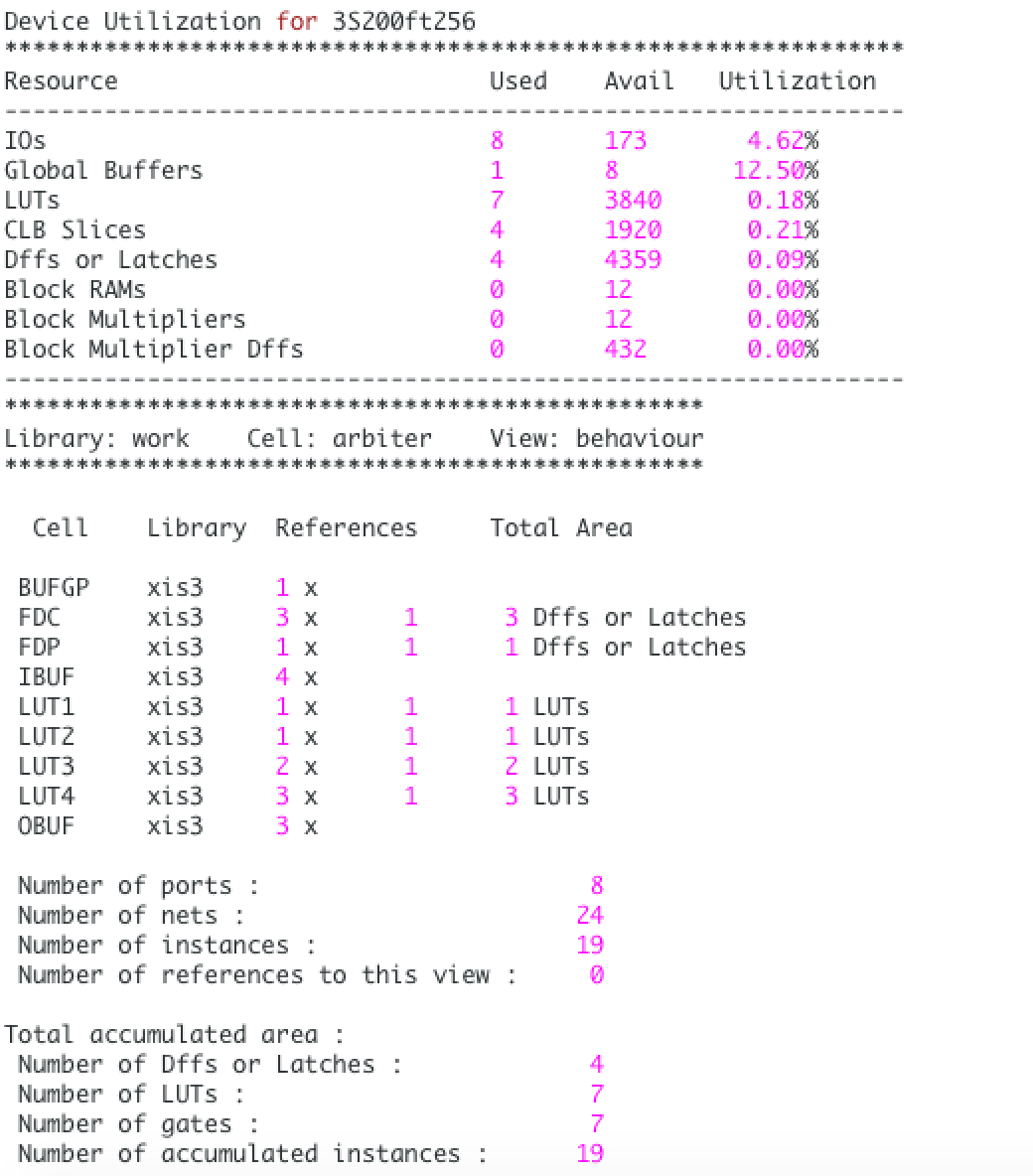


Figure 17 : Synthèse avec binaire

Figure 18 : Synthèse avec code one-hot

Nous obtenons la même chose au niveau de l’utilisation des ressources.

# Conclusion

En abordant les étapes de conception d’un composant numérique (description VHD, élaboration d’un testbench, simulation, synthèse…), ces TP nous en permis de mettre en pratique les notions étudiées en cours.

En plus de nous familiariser à ce langage de description matériel, nous avons pu nous entrainer à réaliser et interpréter des chronogrammes, observer concrètement le comportement des composants, mais aussi, grâce à la synthèse, de comprendre l’intégralité de l’architecture d’un composant numérique.