TP CRFC

TP1 Test structurel de circuits combinatoires et séquentiels

Introduction générale des TP CRFC

L'objectif cette série de TPs est de vous faire comprendre les méthodes et les outils avancés pour concevoir des circuits intégrés dits « testable et robustes». L'ensemble des étapes essentielles à la réalisation « backend » d'un circuit a été vu en 2ème année lors du TP VLSI (spécification, codage VHDL-RTL, simulation simple et back annoté, synthèse suivi de placement routage sur cible ASIC et FPGA Xilinx).

Dans ces TP de 3eme année, nous étudierons dans un premier temps les techniques nous permettant d'améliorer la testabilité des circuits combinatoires et séquentiels (insertion de points de test, scan-path)
Nous étudierons ensuite les techniques d'autotest matériel (BIST) pour les circuits logiques et les mémoires.
Nous verrons enfin comment implémenter une solution de tolérance aux fautes à un circuit de type mémoire (ajout de redondance d'information).
Vous disposerez d'énoncés de TPs et d'annexes décrivant brièvement le fonctionnement des outils que vous utiliserez.

1. Configuration de l'environnement de travail

Tous les fichiers nécessaires au bon déroulement des TPs se trouvent dans le répertoire et les sous-répertoires: ~xph3seilan/TP_TEST/
Vous recopierez chez vous ce répertoire **en n'oubliant pas les fichiers .*, fichiers qui pointent vers les licences des outils.**

> Faire un source .bashrc*** à chaque fois que vous allez utiliser les outils CAO de cette série de TPs. Dans ce fichier on précise la configuration de tous les outils CAO utilisés dans ce TP. Ce fichier est spécifique à l'environnement du CIME. Il est impossible de lancer les outils utilisés dans ces TP sans avoir exécuté cette commande !!!

2. Préliminaires

Ce TP de 4 heures est le premier d'une série de 5. Le travail effectué sert à définir la note de contrôle continu. Vous êtes instamment prié(e) de vous conformer aux instructions suivantes, destinées à faciliter votre travail et notre correction.

1) Créer un répertoire TP_**TEST** dans votre répertoire racine d'utilisateur. Cette opération doit être faite une fois au début du premier TP.
2) Créer un répertoire **TpX (1-5)** dans ce répertoire test pour chaque nouveau TP. Pour le TP numéro 1, vous devez donc créer un répertoire **TP1** dans le répertoire **Test**. Copier dans votre TP1 la totalité de fichiers se trouvant dans le répertoire source ~xphe3seilan/TP_TEST/TP1

3. Plan du TP1

- Introduction au logiciel **TetraMAX** (**`tmax`**, mode graphique)
- Introduction au mode commande de **TetraMAX** (**`tmax -shell`** fichier_script.cmd)
- **TetraMAX**: simulation de fautes et ATPG
- **DFT**: insertion de points de test pour circuit combinatoires
    1. problème d'observabilité
    2. problème de contrôlabilité
- **DFT :** les circuits séquentiels

I) TetraMAX: simulation de fautes et ATPG

- Copier du répertoire TP_TEST le fichier .bashrc_tetramax2013 et faire source afin de pouvoir utiliser les licences Tetramax.
- S'assurer que vous avez bien copié du répertoire source le fichier **`sxlib.v`** (librairie de modèles de simulation pour Tetramax) dans votre répertoire TP1.
- S'assurer que vous avez bien copié le fichier contenant une description structurelle VHDL **`mux.vhd`**.
- Déterminer « à la main » les vecteurs nécessaires au test structurel du multiplexeur. Commenter votre méthode de génération.
- Ecrire un fichier nom_de_fichier_vecteur.**stil** contenant les vecteurs de test calculés par vos soins. Pour cela, vous prendrez le fichier /exemples/ll
- mux.stil que vous compléterez avec vos vecteurs.
- Utiliser **TetraMAX** en mode ***ATPG*** afin qu'il crée automatiquement les vecteurs de test et les enregistrer dans un fichier au format **STIL**. (utiliser les scripts *.cmd se trouvant le sous répertoire exemples)
- Utiliser **TetraMAX** en mode ***simulation de faute*** sur le circuit pour calculer le taux de couverture. (utiliser les scripts *.cmd se trouvant le sous répertoire exemples)
- Comparer avec les vecteurs que vous avez produit manuellement, et commenter le taux de couverture.

II) DFT : insertion de point de test

L'idée clé d'un TPI (Test Point Insertion) est l'insertion de logique supplémentaire dans le circuit afin de rendre les signaux internes du circuit plus facilement contrôlables et/ou observables. Cette logique supplémentaire augmentera la testabilité du circuit, cependant elle ne doit pas influencer la fonctionnalité du circuit en mode opératoire normal. Traditionnellement, les points de test (TP) peuvent être catalogués en points d'observation et points de contrôle.

1) Problèmes d'observabilité

Un point d'observation (OP) est un ajout d'une sortie primaire afin d'obtenir une meilleure observabilité pour des signaux du circuit.

> *Définition : l'**observabilité** représente la « facilité » de vérifier sur les sorties primaires du circuit la présence d'une valeur donné sur un noeud.*

L'impact d'un point d'observation inséré sur une ligne interne l d'un circuit donné est montré sur la figure 1; il augmente l'observabilité de la ligne l elle-même ainsi que l'observabilité des lignes qui se trouve dans le cône d'entrance de la ligne l. « Augmente » signifie qu'il est plus aisé, ou moins difficile, de propager un changement de valeur de la ligne l vers les sorties primaires.

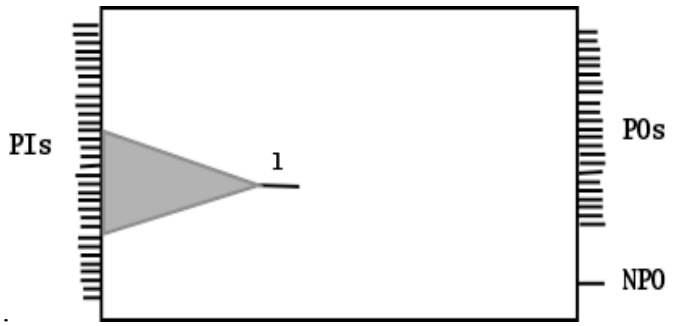


Figure 1:Région influencée par un point d'observation sur la ligne l

Exercice :

Nous allons travailler sur le circuit représenté en figure 2. Les chiffres entre parenthèses sur les lignes du circuit représentent la valeur de l'observabilité selon la technique d'analyse de testabilité **SCOAP**. Plus cette valeur est élevée plus l'observabilité de la ligne est difficile.

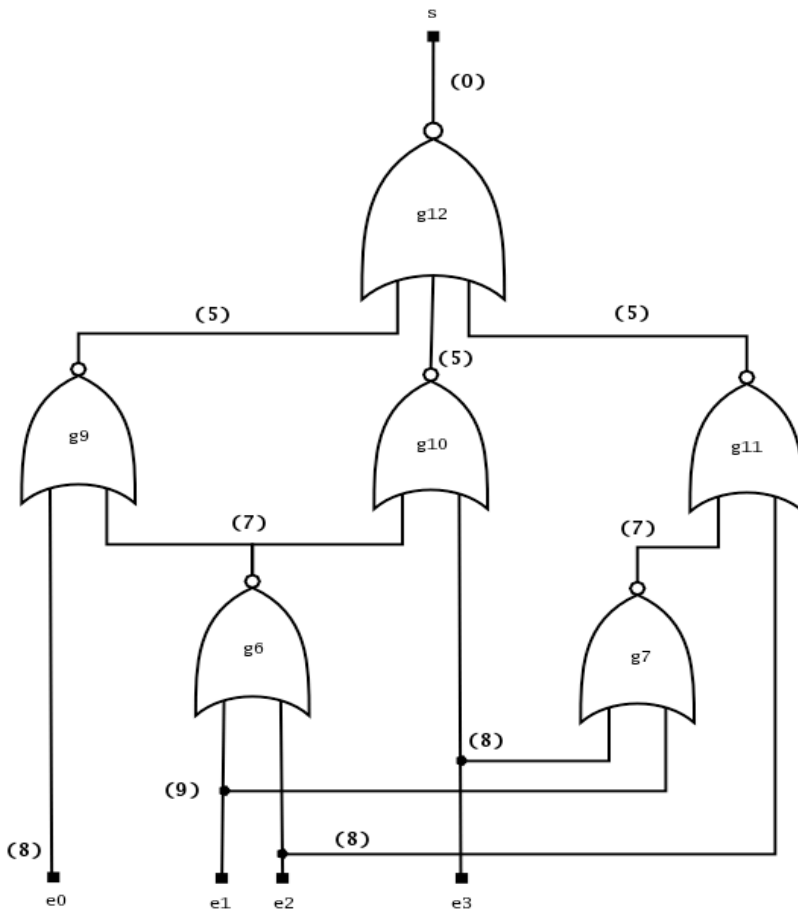


Figure 2: circuit posant un problème d'observabilité

Vous pouvez récupérer le circuit dans :
~xph3seilan/TP_TEST/TP1/comb/circuit_op.vhd

- Déterminer le taux de couverture du circuit avec TetraMAX, qu'observez vous ?
  Remarque sur l'utilisation de TetraMAX: en mode ATPG, par défaut, le logiciel nous donne le taux de couverture du test (test coverage) or ce qui nous intéresse dans cet exercice c'est le taux de couverture de faute (fault coverage). Il faut donc préciser à TetraMAX que l'on souhaite voir le « fault coverage » avec la commande :
  > set_faults -fault_coverage
- Quels sont les collages non détectés ?
  Remarque sur l'utilisation de TetraMAX: pour voir les fautes non détectées utilisez la commande « report faults » de cette manière :
  - ***report_faults -class UD***
- En utilisant les résultats de TetraMAX et le schéma de la figure 2 modifier votre circuit pour améliorer sa testabilité (plusieurs solutions possibles). Quelle solution adoptez vous et pourquoi ?

2) Problèmes de contrôlabilité

Un point de contrôle (CP) est un ajout d'une entrée primaire et de logique supplémentaire dans le circuit sous test afin d'obtenir une meilleure contrôlabilité de certaines parties de circuit.

> *Définition: la **contrôlabilité** caractérise la facilité de positionner un noeud à une valeur logique prédéfinie à partir des entrées primaires du circuit. La contrôlabilité peut être divisée en contrôlabilité à 1, et contrôlabilité à 0.*

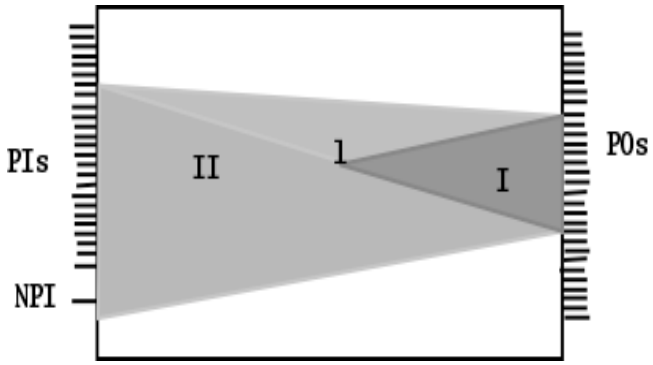


Figure 3: Régions sous influence d'un point de contrôle sur la ligne l

A la différence d'un point d'observation qui n'affecte que l'observabilité du cône d'entrance, un point de contrôle influence la contrôlabilité **et** l'observabilité de régions du circuit. En effet l'observabilité d'une ligne dépend de la contrôlabilité des autres lignes. Ainsi, un changement de contrôlabilité implique un changement d'observabilité. En fait un changement de contrôlabilité implique une **diminution** de l'observabilité du cône d'entrance où le point de contrôle a été ajouté. La zone (I) de la figure 3 indique la région du circuit où la contrôlabilité des lignes a changé à cause de l'augmentation de contrôlabilité de la ligne l. La zone (II) montre la région où l'observabilité des lignes a changé.

Exercice:

Nous allons travailler sur le circuit représenté en figure 4. Les chiffres entre parenthèses sur les lignes de la figure 4 représentent les valeurs de contrôlabilité à 0 et à 1, toujours selon la technique d'analyse de testabilité **SCOAP**. Plus cette valeur est élevée plus la contrôlabilité de la ligne est difficile.

Vous pouvez récupérer le circuit dans :
~xph3seilan/TP_TEST/TP1/`comb/circuit_cp.vhd`

- Déterminer le taux de couverture du circuit avec TetraMAX, qu'observez vous ?
- Quels sont les collages non détectés ?
- En utilisant les résultats de TetraMAX et le schéma de la figure 4 modifier votre circuit pour améliorer sa testabilité (plusieurs solutions possibles). Quelle solution adoptez vous et pourquoi ?

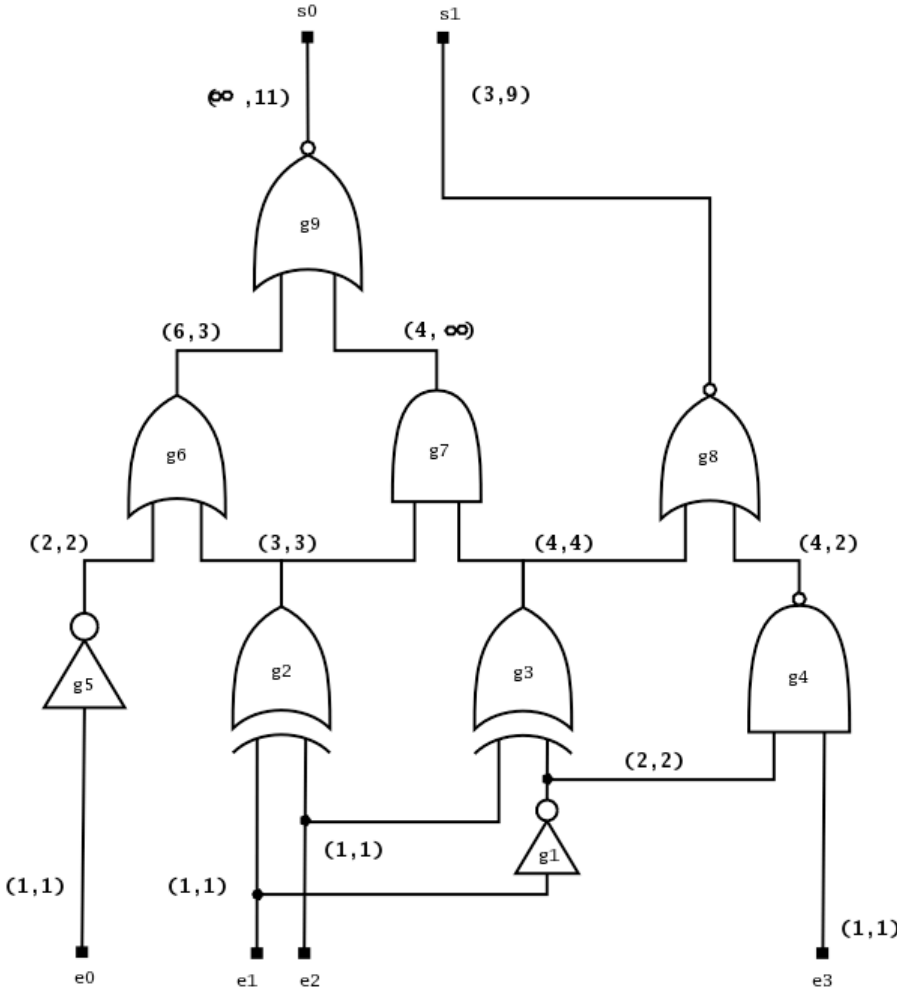


Figure 4: circuit posant un problème de contrôlabilité

II) DFT : les circuits séquentiels

Le test structurel d'un circuit combinatoire est un problème assez simple, facilement résoluble par un outil de génération automatique de vecteurs de test . En effet, les problèmes d'observabilité et de contrôlabilité restent assez simples.

Par contre, dès que le circuit comporte des éléments séquentiels, cela devient difficile pour un générateur automatique de vecteurs de test de trouver la séquence qui lui permettra d'observer ou de contrôler tel ou tel point du circuit.

Nous nous proposons dans cette partie d'analyser la problématique du test de circuits séquentiels en utilisant Tetramax
Les fichiers nécessaires à la bonne marche de ce TP se trouvent dans :

~xph3seilan/TP_TEST/ et dans ~xph3seilan/TP_TEST/TP2/

1) Testabilité d'un circuit séquentiel très simple

a. Protocole de test d'un circuit

Outre les fichiers correspondant à la netlist du circuit (circuit.vhd ou .v) et les informations sur la bibliothèque (bibliothèque.v), Tetramax a besoin en entrée d'un fichier spécifiant comment le circuit doit être testé. Il s'agit du fichier circuit.spf (stil protocol file). Après chargement de ce fichier Tetramax vérifie que ce protocole est compatible avec le circuit à tester (DRC).
Ce fichier circuit.spf contient les informations relatives au circuit à tester (entrées/sorties, timing, contraintes sur les entrées, structure des chaînes de scan, procédures d'application des vecteurs de test…).

Question 1 :

Comparez les fichiers spf correspondant à :

- un circuit combinatoire (example_comb.spf)
- un circuit séquentiel sans chaîne de scan (example_seq_noscan.spf)
- un circuit séquentiel avec chaîne de scan (example_seq_scan.spf)

Quelles sont les différences ? Expliquez.

b. Génération de vecteurs de test

Lorsqu'on utilise Tetramax pour un circuit séquentiel, on a le choix entre plusieurs types d'ATPG. L'outil peut générer des vecteurs de test en mode :

- Basic scan : L'initialisation des bascules se fait à l'aide des/de la chaîne(s) de scan.
Le circuit est rendu combinatoire grâce à cette initialisation et Tetramax peut donc générer les vecteurs de test comme pour un circuit combinatoire pur. La commande Tetramax est la suivante : ***run_atpg basic_scan_only***

- Fast sequential : Dans ce mode il faut indiquer comme paramètre à Tetramax, le nombre de coups d'horloge maximum qu'il peut appliquer au circuit pour initialiser les différentes bascules du circuit. Les commandes Tetramax sont les suivantes :
set_atpg –capture_cycles 6
run_atpg fast_sequential_only

- Full sequential : Dans ce mode on laisse Tetramax toute liberté pour générer les vecteurs de test. La commande est : ***run_atpg full_sequential***

Question 2 : Pour un circuit séquentiel sans chaîne de scan

- Essayez de générer à la main les vecteurs de test du circuit circuit_seq.vhd. Qu'en concluez vous ?
- En utilisant les fichiers circuit_seq.vhd et circuit_seq.spf utilisez Tetramax en mode Fast sequential pour générer des vecteurs de test avec un nombre de cycles de capture égal à 2. Quel est le taux de couverture ?
  Renouvelez l'opération pour un nombre de cycles de captures égal à 6. Qu'observez vous au niveau du taux de couverture ? Pourquoi ?
- Modifiez le fichier .stil généré par Tetramax pour remplacer les vecteurs générés par les vecteurs que vous aurez déterminés à la main.

Question 3 : Pour un circuit séquentiel avec chaîne de scan

- Dessinez le schéma du circuit fourni circuit_seq_scan.vhd.
- Décrivez les étapes permettant d'appliquer un vecteur de test au circuit en spécifiant pour chaque étape la valeur des broches de test
- Générez les vecteurs de test avec Tetramax en mode basic scan. Quel est le taux de couverture obtenu ? Quelle est la différence avec le taux obtenu pour un circuit non scan ?
  Commentez.

TP TEST 2

Synthèse en vue du test d'un circuit séquentiel
Utilisation de Design Compiler et Tetramax

Introduction

Les fichiers nécessaires à la bonne marche de ce TP se trouvent dans :

~xph3seilan/TP_TEST/ et dans ~xph3seilan/TP_TEST/TP2/

Synthèse et Design for Test du circuit « Réveil Numérique »

Dans cet exercice, vous allez réaliser la synthèse logique d'un circuit séquentiel relativement simple, et la synthèse testable de ce circuit. Nous allons commencer par une synthèse simple de ce circuit modélisé en VHDL. Puis, nous allons ajouter de façon automatique la logique de scan sur les bascules, et les points de test pour améliorer la testabilité. Nous allons regarder le cout en matériel et l'impact sur les performances, ainsi que la couverture de fautes. Puis on vérifiera la couverture de fautes avec Tetramax.

Préparation : copier dans votre répertoire TP2 les sources du répertoire TP_TEST/TP2/TP_SCAN_INSERTION
Copier également le fichier de configuration .bashrc_synth_DV2013 situé dans TP_TEST

Vérifier que dans ce répertoire vous trouvez les sources du design (./vhdl), les scripts (./Script), et les fichiers de configuration de l'outil Synopsys (.synopsys_dc.setup – pour design Vision et tetramax_setup.dc – pour Tetramax). Vérifier également l'existence d'un fichier corelib_pour_ATPG.v.

Cette partie de TP sera faites dans le répertoire TP_SCAN_INSERTION.

1. Synthèse en utilisant Design Compiler à travers l'interface Design Vision

Suivre les étapes de la synthèse présentées en annexe Design Vision (voir plus loin).

Objectifs :

- Faire une première synthèse du circuit COMPUTE_BLOCK (voir annexe synthèse simple) avec une horloge générée à clk=10ns.
- Noter la surface du circuit COMPUTE_BLOCK, le temps critique et la puissance consommée.
    - Observation de la surface initiale du bloc. Noter !
    - Observation de la répartition des slacks. Noter !
    - Observation de la consommation : quel bloc consomme le plus ?

- Quels types de bascules ont été utilisés ? Enumérer 2-3 types. _____
- Observer les éventuels warnings et les comprendre.
- **Sauvegarder le résultat en verilog et VHDL et ddc.**

2. Synthèse avec l'insertion de scan (suivre les étapes présentées dans l'annexe)

Objectifs :

- Réaliser une synthèse avec insertion des éléments de testabilité et de scan dans le module COMPUTE_BLOCK. Il s'agit d'utiliser une méthodologie de type full scan avec des bascules de type Mux Scan. Suivre la méthodologie d'insertion du scan et les conseils donnés dans l'annexe 5. Répondre aux questions posées dans l'annexe 5.
- Suite à l'insertion du scan (suivre correctement les étapes de l'annexe) sauvegarder une netlist en format vhdl et verilog.
- Générer un ficher de protocole de test (alarm_clock.spf). Le comprendre !

3. Analyse de couverture de fautes et génération de vecteurs de test (ATPG) – suivre les étapes de l'annexe Tetramax

Objectifs:

- Générer les vecteurs de test pour le circuit avec scan.
- Analyser et commenter le fichier de vecteurs de test (patterns_alarm_clock.v) et les résultats obtenus dont la couverture de fautes
- Réaliser une simulation avec injection de fautes de collages sur certains nœuds internes (2 points pour la partie combinatoire, 1 faute sur les bascule scan, position 5) pour montrer la détection par les vecteurs de test générés.

TP Test 3

Création et utilisation d'un BIST logique

Préambule au TP:

Vous trouverez dans le répertoire ~xph3seilan/TP_TEST les fichiers du TP3 qu'il vous faudra copier sur votre compte.

Objectifs du TP :

Ce TP a pour but de vous faire concevoir et simuler une architecture testable d'un circuit logique arithmétique basée sur l'utilisation de LFSR et MISR.

Partie 1 : Conception de composants autour d'un BIST logique

On cherche à construire un BIST logique pour une unité arithmétique-logique (ALU) à 8 opérations. Elle a 2 opérandes sur 4 bits et un signal de commande sur 3 bits. Cet ALU se trouve dans le répertoire ~xph3seilan /TP_TEST/TP3/vhd sous la forme d'une description VHDL RTL (fichier ALU.vhd) et d'une netlist synthétisée en technologie CORELIB 0.35 um – fichier ALU_synth.vhdl

Comme vous l'avez vu en cours, un BIST logique est composé d'un générateur de vecteurs de test aléatoires de type LFSR, d'un analyseur de signature de type MISR, et d'un contrôleur de test.
Avant l'assemblage de l'architecture finale, il faudra construire d'abord ces éléments du BIST.

- **A) Construction du LFSR**
    1. Construire en VHDL un registre à décalage de type LFSR capable de fournir des valeurs logiques sur l'ensemble des entrées de l'ALU(c'est à dire sur 11 bits). Ce circuit sera connecté à l'ensemble des entrées du circuit ALU (entrées opérandes A et B + entrées de type sélection CMD). Pour réaliser votre LFSR , utiliser le polynôme générateur de LFSR qui vous a été donné en cours.
    2. Ce LFSR possède aussi une fonction de type « Shift Register ». Ce signal permet de l'initialiser à des valeurs quelconques par décalage, envoyées de l'extérieur.
    3. Pour l'architecture de ce registre LFSR se référer à l'architecture de type BILBO vue en cours.
    4. Prévoir les signaux supplémentaires sont :
        - SIN – data in 1 bit- pour le décalage,
        - SOUT- data out sur 1 bit – pour le décalage
        - SHIFT/GENVECT – signal qui permet la sélection des modes de SHIFT (le LFSR ne fait que des décalages), ou GENVECT (le LFSR agit en tant que générateur de vecteurs de test sur 11bits qui sont ensuite appliqués au circuit)

- **B) Construction du MISR**
    1. Construire en VHDL l'analyseur de signature parallèle MISR basé sur un LFSR de polynôme générateur de degré 4 (***! attention prévoir pour le MISR un registre parallèle à 4 entrées et 4 sorties !***). Se référer au cours pour le polynôme générateur et pour l'architecture parallèle.
    2. Ce registres MISR a plusieurs fonctions supplémentaires tel que :
        - le « SHIFT » - utile pour être initialisé par décalage mais aussi pour sortir la signature par décalage à la fin des cycles de simulation (on sera donc en mode TEST)
        - « MISR » - le système est en mode TEST et le fonctionnement est de compactage de sorties du circuit en une signature unique
        - « REGISTRE » - le MISR doit être chargé en parallèle (mode de fonctionnement NORMAL).

Les signaux supplémentaires à prévoir sur 1 bit

- sont SHIFT/MISR – ces fonctions sont exclusives, le système est en mode test.
- TEST/NORMAL – le système est en mode TEST (alors il sera configuré soit pour faire du SHIFT soir pour faire MISR) ou il est en mode NORMAL alors il agit en simple registre à chargement parallelle
- SIN_M, $SOUT_M$ – entrée et sortie de chargement/déchargement sériel sur 1 bit
- 4 sorties parallèles Qout en provenance de bascules du MISR – pour utiliser la signature en lecture parallèle

Compiler les deux descriptions VHDL (utiliser pour cela les commandes de l'outil Modelsim)

Partie 2 : Mise en place d'une architecture testable de l'ALU

Créer une architecture TOP (description en VHDL RTL) incluant le circuit arithmétique et son BIST logique selon le schéma donné en figure 1.

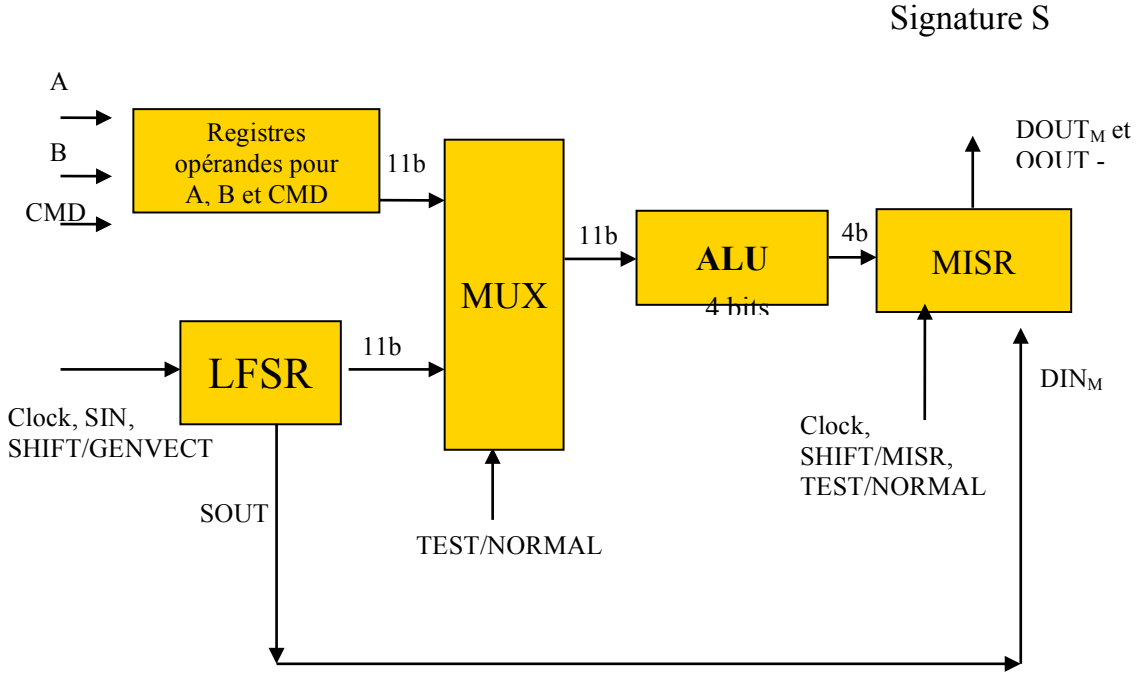


Figure 1. Schéma de principe d'un BIST logique autour d'une ALU

Partie 3. Simulation du BIST

1) Concevoir un testbench (qui peut être vu comme le contrôleur de BIST) et réaliser une simulation complète du BIST à l'aide de l'outil Modelsim. Pour ce faire vous devez valider dans un premier temps le mode de fonctionnement NORMAL, et prévoir une validation fonctionnelle complète de votre design. Dans ce mode, le circuit réalise toutes les opérations prévues dans les spécifications. (Prendre pour la simulation une clk de 10ns). Les opérandes sont envoyées de l'extérieur, et le registre MISR est en mode REGISTRE (pas de fonctionnement LFSR)

2) Dans la suite du testbench, l'ALU sera configurée en mode **TEST** et elle doit effectuer les opérations suivantes :

   2.1. Par décalage (mode **SHIFT sur les deux composants**) initialiser le **MISR** à « 0000 » et le **LFSR** à « 01010101010 »

   2.2. Toujours en mode **TEST**, faire une simulation complète du circuit en absence de toute faute (le **LFSR** est en mode **GENVECT** et le **MISR** est en mode **MISR**).

   Durant les simulations, observer comment le module LFSR génère les vecteurs de test pseudo-aléatoires à chaque cycle d'horloge. Observer également le fonctionnement du circuit MISR. Combien de cycles de simulation sont nécessaire durant la phase de test ?

3) Injecter dans le CUT quatre fautes de collage à 1 ou à 0 (au choix) sur des nœuds de l'ALU et observer le fonctionnement du BIST logique .
   Est-il possible de diagnostiquer/localiser une faute ? Pourquoi ?
   Est-ce qu'on observe un alias ?

Comment faire pour réduire la probabilité d'aliasing ?

Justifier vos réponses…

Annexe 3. Description de l'UAL

Le composant ALU représente une unité arithmétique logique à 8 opérations et 2 opérandes sur 8 bits. Le deux opérandes sont notées A et B. Les 8 opérations sont codées sur 3 bits (CMD[2 :0]). Le fonctionnement d'une tranche UAL est donné dans le tableau plus bas.

CMD2	**CMD1**	**CMD0**	**ALU_OUT**
0	0	0	A-B
0	0	1	A+B
0	1	0	A+1
0	1	1	A
1	0	0	A or B
1	0	1	A and B
1	1	0	A xor B
1	1	1	A bar
others	others	others	----

Le circuit utilisé dans le TP BIST représente une UAL avec deux opérandes sur 4 bits, et 8 opérations codées sur 3 bits (CMD).

TP TEST 4

Conception d'un BIST de mémoire embarquée

Test des mémoires embarquées

Introduction

Qu'est ce qu'un BIST de mémoires embarquées?

Le **BIST de Mémoires** (Memory Built In Self Test) est une méthodologie de test de mémoires intégrées automatisée. Le test s'effectue à l'aide d'un contrôleur supplémentaire (par rapport au contrôleur classique de mémoires) qui s'interface avec le bloc de mémoire. Le contrôleur est capable de vérifier le bon fonctionnement de la mémoire, adresse par adresse, d'identifier une adresse erronée et de rapporter l'état de la mémoire (correcte ou erronée). Dans une version simpliste, un contrôleur de BIST indique seulement si la mémoire a passé le test ou pas. Les modèles plus évolués de contrôleurs de BIST permettent d'avoir des informations de diagnostique supplémentaires, comme par exemple l'adresse et les bits fautifs ou en quelles conditions la mémoire se comporte d'une manière incorrecte.

Pourquoi a-t-on besoin de ce test?

Tout fabricant de circuits intégrés doit s'assurer que ses circuits fonctionnent correctement en fin de fabrication. Pour atteindre un certain niveau de qualité requis par les clients, le circuit est testé constamment pendant toutes les phases de production. Comme vous le savez, le test est très coûteux: il exige du temps, des équipements spécialisés et très chers, le savoir-faire. Aujourd'hui, la complexité des circuits augmente sans cesse et le test est confronté à beaucoup de défis.

Dans les circuits actuels, les mémoires embarquées représentent la plus grande jusqu'à 80% de la surface totale. Souvent, le rendement de production (manufacturing yield) est lié à la qualité de fabrication de ces mémoires. Par conséquent, il est nécessaire de tester les mémoires. De plus, les informations de diagnostique obtenues après le test, permettent d'utiliser des techniques de réparation afin d'éliminer les défauts.

Les équipements automatisés de test (ATE) ne permettent pas de tester la mémoire directement, car le bloc de mémoire est souvent enfoui dans la structure du circuit. Des techniques de BIST de mémoires ont vu le jour afin de rendre le test de mémoires économiquement et techniquement viable.

Les avantages de test de mémoires à l'aide du BIST de mémoires

Le BIST offre certains avantages :

1. Premièrement, les dépenses liées aux testeurs externes sont minimisées.
2. Le nombre de signaux de test et contrôle est réduit.
3. Le contrôleur de BIST est implémenté sur mesure, totalement adapté à la mémoire qui devra être testée.

4. Le temps de test est optimisé et la vérification peut se faire à des fréquences très élevées, éventuellement à la vitesse nominale de fonctionnement.
5. Souvent, la mise en pratique du test est très simple et ceci permet d'éliminer le besoin de forte spécialisation des ingénieurs de test.

Présentation du BIST pour lés mémoires

Le principe de connexion d'un bloc BIST à la mémoire est illustré dans la figure 1.

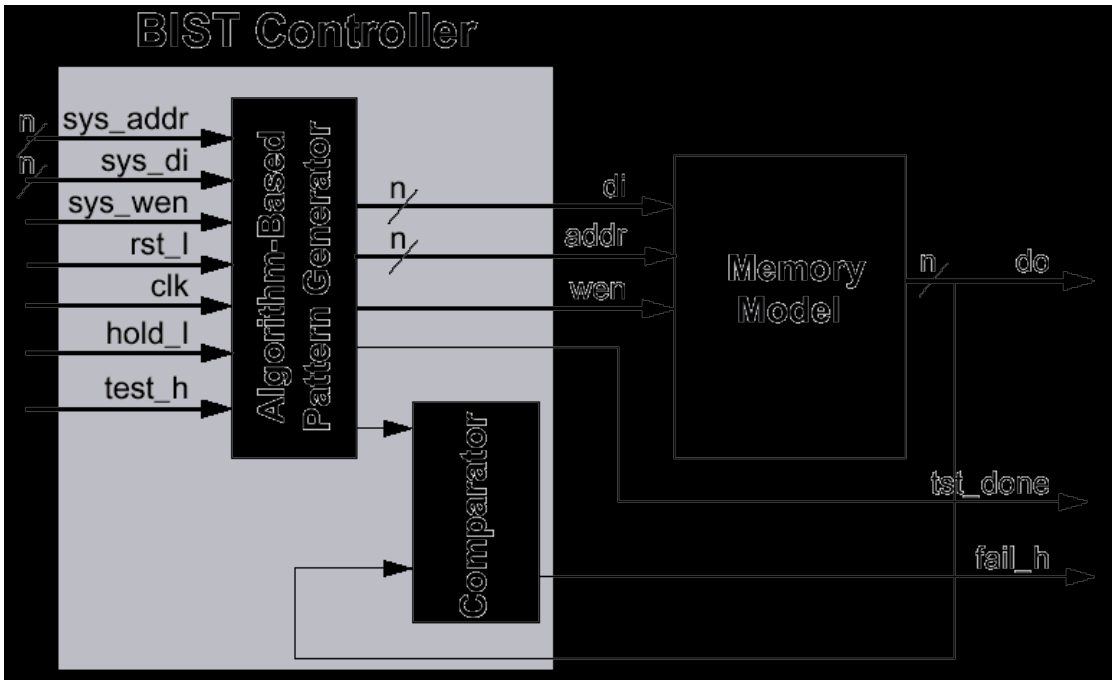


Figure 1 : Contrôleur du BIST de Mémoires

Le contrôleur du BIST présetné dans la figure 1 est un schéma de principe. Le contrôleur de BIST s'interpose entre le reste de l'application et la mémoire et génère/récupère les signaux de contrôle et de données (di, do).

- **Pendant le fonctionnement normal du système**, le contrôleur BIST est transparent, permettant l'accès direct à la mémoire de l'extérieur. Les entrées Sys_addr, sys_di, Sys_wen sont des signaux qui viennent de l'extérieur, et s'appliquent directement à la mémoire en fonctionnement normal.
- **Pendant les tests**, le BIST contrôle la mémoire, il pilote les signaux d'entrée de la mémoire (addr, di, Wen) et il observe les sorties en les comparant avec les données de référence.
- **Dans le schéma plus haut, le module top contiendra un système de multiplexage de donnée en entrée de la mémoire, piloté par le signal test_h.**

Le test consiste à écrire et lire dans la mémoire des données typiques. Les données en sortie sont comparées à la valeur de référence et les différences permettent de mettre en évidence les

défauts éventuels se trouvant dans la mémoire.

Les défaillances qui peuvent apparaître dans une mémoire sont les suivantes:

des fautes de type collage (stuck-at-0 et stuck-at-1): certaines cellules de mémoire sont collées à 0 ou 1 (fautes permanentes)

- des fautes de transition: certaines cellules ne peuvent pas effectuer certaines transitions
- des fautes de couplage: le changement d'une cellule agresseur provoque la transition de la cellule victime.
- des fautes de voisinage: une catégorie plus générale que les fautes de couplage. Le contenu de la cellule victime dépend du contenu de ses voisines
- des fautes dans le décodeur d'adresses. Ces fautes peuvent avoir des manifestations variées: certaines cellules ne peuvent pas être accédé ou sont accédées par des d'autres adresses, etc.
- des fautes dans le circuit de lecture/écriture: certains bits d'un mot de données sont toujours erronés.
- des fautes « at-speed » ou de timing: ce sont des fautes qui se manifestent seulement aux fréquences de fonctionnement élevées, la mémoire fonctionne parfaitement aux basses fréquences.

La manière dont la mémoire est écrite et lue ainsi que le sens de parcours de la mémoire a beaucoup d'influence sur le type des fautes qu'un BIST peut détecter. En effet, pendant le test on peut faire varier deux paramètres: la valeur des données à écrire et la manière dont l'espace d'adresse est parcouru. (Rappel sur la possibilité de tester tout cas de faute de couplage possible)

Pendant les phases de lecture, un comparateur est utilisé pour comparer les données lues de la mémoire avec les données de référence.

Afin de parcourir la mémoire, on doit pouvoir accéder toutes les adresses et on doit également changer le sens de parcours des adresses pour la détection de fautes de couplage et de voisinage. Un générateur d'adresse à base d'un compteur UP/DOWN peut générer linéairement tout l'espace adresses entre l'adresse 0 et n-1 dans les deux sens. (n= la taille de la mémoire).

L'algorithme de test est constitué d'une série de séquences spécifiques. Une séquence consiste en certaines opérations de lecture et écriture qui seront exécutées pour chaque adresse. Les capacités de détection de fautes sont données par la qualité de l'algorithme de test.

Pendant le TP nous allons utiliser une version réduite de l'algorithme de test reconnu comme un standard industriel: le MARCH-C (connu aussi sous le nom de Marinescu) .

Cet algorithme consiste dans les opérations suivantes:

↑ (W0); ↑ (R0W1); ↑ (R1W0); ↓ (R0W1); ↓ (R1W0); ↓ (R0)

Commentaire:

↑ (W0) – parcours de la mémoire de l'adresse 0 à n-1 en écrivant des données à 0.

↓ (R0W1) – parcours de la mémoire de l'adresse n-1 à 0, lire 0 et écrire 1

Description des circuits utilisés pendant le TP

Les fichiers fournis pour le déroulement de ce TP se trouvent dans le répertoire /xph3seilan /TP_TEST/TP4 et sont des descriptions VHDL de la mémoire et du contrôleur BIST et modelsim.ini. Copier également le fichier de configuration .bashrc_modelsim10_0b se trouvant en répertoire /xph3seilan/TP_TEST/.

A. *Le bloc mémoire*

La mémoire utilisée lors du TP est décrite dans le fichier ***memory.vhd***. Il s'git d'une mémoire synchrone avec une capacité de 1024 adresses et un mot mémoire sur 16 bits. Les entrées/sorties sont:

ADD – bus d'adresses (10 bits)

DIN – bus de données en entrée sur 16 bits

DOUT – bus de données en sortie sur 16 bits

CSn – chip select. Si le signal est 0, la mémoire est active, donc on peut faire des accès.

WEn – write enable: 0 pour Write, 1 pour Read

CLK – entrée du signal clock

B. *Le BIST*

Le bloc BIST implémente l'algorithme MARCH_C et utilise un compteur pour la génération d'adresses et un comparateur pour la vérification de résultats (voir fichier ***mbist.vhd***). L'interface contient les signaux suivants :

MEM_ADD – bus d'adresses (10 bits)

MEM_DIN – sortie de données sur 16 bits

MEM_DOUT – entrée de données sur 16 bits

MEM_CSn – chip select

MEM_WEn – write enable

CLK – signal clock

TEST_RUN – commande du début du test

TEST_FAIL – sortie utilisée pour indiquer une erreur détectée par le test

TEST_SO – scan-out, utilisé pour extraire les informations de diagnostique

Fonctionnement

Le fonctionnement du BIST est déclenché par le signal TEST_RUN. Le test commence lorsque ce signal passe à 1.

Les sorties MEM_xxx seront connectées à la mémoire fournissant ainsi les séquences de test. Si une défaillance a été détectée à une certaine adresse, le signal TEST_FAIL s'active temporairement pendant la période d'horloge d'accès à l'adresse où se trouvent les cellules erronées. A partir de ce moment, par le signal TEST_SO on récupère en série les bits de l'adresse erronée (le LSB sort en premier, suivi par les autres bits vers les MSB).

A la fin du test, la mémoire est de-sélectionnée et le signal TEST_FAIL rapporte le bon déroulement du test. Si ce signal est à 1, la mémoire est considérée défaillante, sinon elle est considérée correcte.

Travail demandé pour ce TP

Exécuter un source du .bashrc_modelsim10_2c se trouvant en répertoire /xph3seilan/TP_TEST/ afin d'obtenir les licences des outils utilisés dans ce TP.

1. *Testbench*

Après avoir analysé les fichiers .vhd donnés, extraire la FSM implémentée dans le fichier mbist.vhd. Comprendre le fonctionnement du BIST de mémoire donné.

Implémentez en VHDL un testbench pour faire fonctionner l'ensemble : BIST-mémoire.

Commentaires à prendre en compte pour l'écriture du testbench :

- Vous devez connecter la mémoire au BIST de mémoire
- Vous devez générer un signal d'horloge avec lequel il faut piloter les entrées d'horloge des deux blocs
- Vous devez générer le signal TEST_RUN avec une valeur initiale de 0 et activation à 1 après un délai raisonnable (exemple : 100ns)

2. *Simulation sans fautes*

Simuler le circuit en utilisant l'outil ModelSim (licence 10_0b).

- Créer une bibliothèque de travail avec les commandes connues sur Modelsim:

  vlib lib_work et vmap lib_work $HOME/chemin complet de votre bibliothèque/lib_work

- Compiler la mémoire, le bloc BIST et en dernier le testbench

  vcom +acc -work lib_work <nom_du_fichier_vhdl>

- Simuler le circuit:

  vsim <nom_de_l'éntite_test_bench>

- Représenter dans la fenêtre des formes d'onde les signaux sur les interfaces de la mémoire et du BIST
- Observer sur les formes d'onde les séquences de test. Identifiez les opérations élémentaires (read 0/1 et write 0/1). Identifiez les séquences de test (w0, etc)
- Identifier la fin du test et l'état du signal TEST_FAIL

3. Simulation avec injection de fautes

1. Modifier le testbench pour injecter des fautes dans la mémoire. Indication: pour une certaine adresse, écrire des mauvaises données dans la mémoire. Ceci sera fait dans le fichier testbench crée, et non pas dans le BIST de mémoire.
2. Simuler le circuit et suivre l'évolution du signal TEST_FAIL. (A l'adresse d'injection de fautes, vous devez remarquer l'activation de ce signal et des informations qui apparaissent sur TEST_SO).
3. Identifier la valeur sur TEST_SO. Elle doit correspondre à l'adresse que vous avez choisie plus haut.

Questions :

- Pour l'algorithme de MARCH proposé de combien de cycles d'horloge a-t-on besoin pour un test complet de fautes de collage SA et de transitions ?
- Il existe une limitation fondamentale dans la capacité de rapport d'erreurs de ce BIST. Identifier cette limitation et proposer des solutions?
- Quels sont, selon vous, les problèmes de performance qui apparaissent dans le cas des le test des mémoires de grande capacité? Quelles sont les solutions envisageables ?
- Implémenter et tester l'algorithme MARCH C donné dans le cours. Dans le compte rendu, fournir le code rajouté commenté.

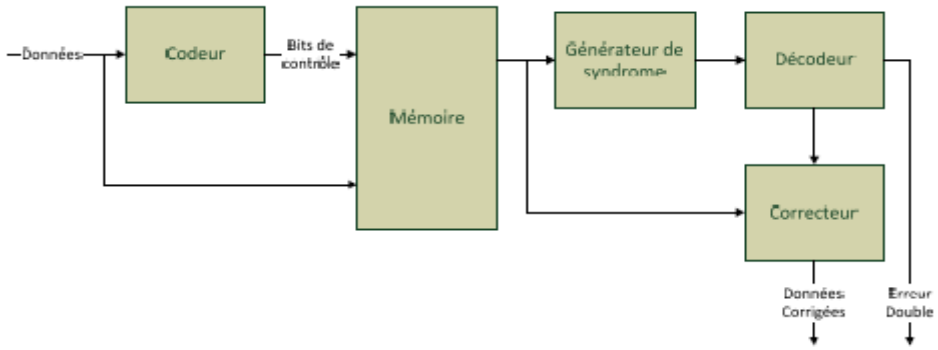
TP TEST 5

Tolérance aux fautes : détection/correction d'erreurs

1. Objectifs

Dans ce TP vous aurez pour objectif d'appliquer les techniques de détection/correction d'erreurs à une mémoire.

2. Rappels



Le système tolérant aux fautes sera architecturé autour de la mémoire fournie lors du TP précédent.
Fonction de chaque bloc :
- Codeur : il génère les bits de contrôle, le nombre de bits de contrôle nécessaires est directement lié au nombre de bits de données : 2^r>= k+r+1 (avec : *k* nombre de bits de données et *r* nombre de bits de contrôle)
- **Générateur de syndrome** : il s'occupe de générer, à partir des valeurs stockées en mémoire (data et bits de contrôle), le vecteur nécessaire pour détecter et corriger l'erreur/les erreurs.
- **Décodeur** : à partir des bits du syndrome, positionne le bit à corriger en cas d'erreur, signale éventuellement une éventuelle erreur double.
- **Correcteur** : il corrige, si nécessaire, la donnée sortant de la mémoire à partir des informations reçues du décodeur.

Detection simple/correction simple

Codage de Hamming :
Soit la matrice H ci-dessous :

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Soit *m* le mot que l'on cherche à transmettre. Dans notre cas, le mot a pour valeur :
m = [p1 p2 d3 p4 d5 d6 d7 p8 d9 d10 d11 d12 d13 d14 d15]

Les bits notés pi sont les bits de contrôle, et les bits di sont les données que l'on cherche à transmettre. Les bits de contrôle prennent les positions 2^i. Les données sont insérées dans les autres positions.
En notant que + = XOR on a
p1 = d3 + d5 + d7 + d9 + d11 + d13 + d15
p2 = d3 + d6 + d7 + d10 + d11 + d14 +d15
p4 = d5 + d6 + d7 + d12 + d13 + d14 + d15
p8 = d9 + d10 + d11 + d12 + d13 + d14 + d15

Une fois le message transmis, il faut, à la réception, calculer le syndrome d'erreur S, où S = [s4 s3 s2 s1], matrice une ligne à 4 colonnes. Si aucune erreur n'est présente, on doit avoir S = [0 0 0 0]. Si cela n'est pas le cas, alors la valeur décimale du mot binaire (s4 s3 s2 s1) nous donne la position de l'erreur dans le mot.
Afin de calculer S, nous utilisons la formule suivante :
S = v. T(H) où v est le mot codé et T(H) la transposée de la matrice de Hamming.
On obtient alors:
s1 = v1 + v3 + v5 + v7 + v9 + v11 + v13 + v15
s2 = v2 + v3 + v6 + v7 + v10 + v11 + v14 + v15
s3 = v4 + v5 + v6 + v7 + v12 + v13 + v14 + v15
s4 = v8 + v9 + v10 + v11 + v12 + v13 +v14 + v15

Détection double/correction simple

Pour pouvoir détecter une erreur sur deux bits, tout en ne pouvant en corriger qu'une, il faut utiliser une matrice génératrice étendue décrite ci-dessous.

$$H = \{R\ lignes | N\ colonnes\} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$
$$Mot = [P_{ov}\ P_1\ P_2\ D_3\ P_4\ D_5\ D_6\ D_7\ P_8\ D_9\ D_{10}\ D_{11}\ D_{12}\ D_{13}\ D_{14}\ D_{15}]$$

Coté codage cela introduit un bit de contrôle supplémentaire pov= p1+p2+d3+P4+d5+…+d15
Coté décodage/détection/correction le nouveau syndrome est S = [s4 s3 s2 s1 sov] avec S = v. T(H) et sov= v0+v1+v2+v3+…+v15
Si sov=0 → (s4 s3 s2 s1)= 0 alors pas d'erreurs
→ (s4 s3 s2 s1) différent de 0 alors erreur double (pas de correction)
Si sov=1 → erreur simple correction avec la valeur de (s4 s3 s2 s1)

3. Travail attendu

3.1. Vous allez devoir construire autour de la mémoire qui vous est fournie dans le répertoire TP4 les différents blocs permettant d'avoir au final une mémoire robuste permettant de détecter des erreurs double et corriger des erreurs simples (fichiers VHDL).
3.2. Vous écrirez le testbench permettant de valider votre architecture. Pour compléter la validation vous injecterez des erreurs simples et des erreurs doubles dans votre mémoire et vérifierez le bon fonctionnement du système.
Dans votre rapport, discutez les avantages et inconvénients de cette technique.

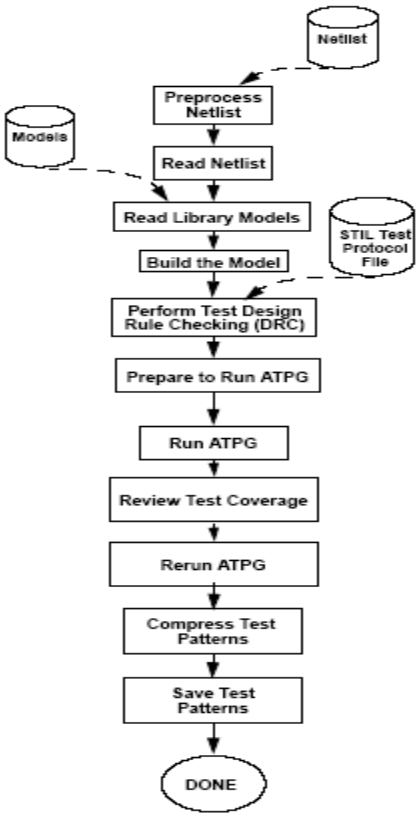
Annexe 1: Utilisation de l'outil TetraMAX (Synopsys) - Types de fautes - Couverture de fautes

I Description sommaire de Tetramax

Configuration de l'outil TetraMAX
La configuration de l'outil TetraMAX est réalisée par l'intermédiaire d'un fichier qui contient un ensemble de commandes. Ce fichier est dénommé "tetramax_setup.dc" et il est utilisé pendant l'exécution des commandes et surtout pour la génération du protocole de test.

Lancer l'outil "TetraMAX"
Pour lancer l'outil d'analyse de couverture de fautes et génération de vecteurs de test TetraMAX, il suffit d'entrer la commande UNIX "tmax &"dans une fenêtre cmdtool ou xterm. **Il faudra lancer l'outil TetraMAX en s'assurant que le fichier "tetramax_setup.dc" se trouve dans ce répertoire.**

Flot d'analyse de couverture de fautes à travers les menus
La figure suivante illustre le concept et le flot ATPG.



Tetramax Design Flow

Le flot d'analyse de couverture de fautes et de génération de vecteurs de test est le suivant (et peut s'exécuter par une succession de séquences « presse bouton » en cliquant de gauche à droite sur les onglets se trouvant en dessous des menus principaux).

1. **Lecture de la netlist synthétisée** : Netlist ->Sélectionner le fichier verilog généré lors de la synthèse et faire RUN
2. **Lecture des modèles ATPG** des cellules de bibliothèque (pour la bibliothèque 0.35 on utilise le fichier AMS 0.35 CORELIB, pour le premier TP, utiliser le fichier sxlib.v).

   Netlist -> Sélectionner le fichier corelib_pour_ATPG.v (sxlib.v) -> RUN.

   Ces fichiers contiennent une partie des descriptions ATPG des cellules de bibliothèque.
3. **Build** : construction du modèle ATPG de tout le circuit. Le circuit est mappé sur les modèles ATPG des cellules. Attention à la sélection du module hiérarchique top (qui est CORE dans le cas du reveil dans le TP2) ! Se renseigner sur les options de cette fenêtre en se reportant éventuellement au manuel d'utilisation, qui se trouve dans le Help!
4. **Etudier les warnings qui en résultent !** Menu Rules-> Report Violations pour plus d'informations sur un message de warning.
5. **Lire le protocole de test généré par l'outil de synthèse.** Lors de cette étape l'outil d'analyse vérifie l'existence de la chaîne de scan, les connections logiques, si toutes les bascules sont pilotées par une horloge contrôlée de l'extérieur pendant le test, s'il existe deux modes de fonctionnement : normal et test (vérification du signal qui est utilisé pour la commutation). Si le fichier .spf n'existe pas, il peut être créé via TetraMAX, mais une connaissance très détaillée de la netlist est nécessaire, autrement gare aux erreurs systématiques !

   Onglet DRC, onglet Run, sélectionner le fichier .spf et Run.
6. **Correction des éventuelles erreurs DRC**.
7. **Création de la liste de fautes et génération de vecteurs** : onglet ATPG, onglet "general settings"
   a. **full-sequential** – si le circuit est séquentiel, il s'agit de ATPG optimisé ou **basic-scan** dans le cas d'un circuit combinatoire ou avec scan
   b. Pattern Source: **Internal**
   c. Fault Sources : **Add all faults**, modèle Stuck-at ou autre.
   d. Pour la simplicité on peut aussi exécuter la commande : **run_atpg -auto**
   e. Les options des modèles de fautes peuvent être imposées également à travers le menu Faults -> Set Faults et sélection de modèles
   f. Les vecteurs de test (patterns) peuvent être imposés dans le menu Patterns ->Set Patterns
8. **Vérification de la couverture de fautes**
   a. Faults -> Report Faults cocher All et observer
   b. Analyze -> Faults, sélectionner une classe de fautes et analyser les portes.
9. **Compression des vecteurs de test Onglet Compress et OK**
10. **Ecrire le fichier de vecteurs de test :**

    Write Patterns, donner le nom du fichier, Pattern Source Internal, et le format doit être Verilog Single File, Scan Format doit être Parallel, sans compression de fichier.

II) Types de fautes et couverture de fautes

There are five higher-level fault categories containing a total of 11 lower-level fault classes, organized as follows:

DT: detected
- DR: detected robustly
- DS: detected by simulation
- DI: detected by implication

PT: possibly detected
- AP: ATPG untestable-possibly detected
- NP: not analyzed-possibly detected

UD: undetectable
- UU: undetectable unused
- UT: undetectable tied
- UB: undetectable blocked
- UR: undetectable redundant

AU: ATPG untestable
- AN: ATPG untestable-not detected
- ND: not detected
- NC: not controlled
- NO: not observed

"Undetectable" UD means that the fault cannot be detected by any means. A straightforward example of this is an unused Q-bar output of a flip-flop, which is classified as UU (undetectable unused). It is impossible to detect faults at these types of locations because they cannot be observed, either directly or indirectly.

"ATPG Untestable" AU means that TetraMAX could not find a successful detection pattern while maintaining all design constraints (for example, constant ports, contention avoidance). This type of fault might become testable if restrictions are relaxed or if functional test is used instead.

For a detailed breakdown of fault classes, use the -summary verbose option of the set faults command:
TEST_T> **set_faults -summary verbose**

The three possible quality measures are defined as follows:
• Test coverage = detected faults / detectable faults

$$\text{Test Coverage} = \frac{DT + (PT \times PT\_credit)}{All\_Faults - UD - (AN \times AU\_credit)} \times 100$$

where PT_credit =50% and AU_credit =0%

• Fault coverage = detected faults / all faults

$$\text{Fault Coverage} = \frac{DT + (PT \times PT\_credit)}{All\_Faults} \times 100$$

TEST_T> **set_faults -fault_coverage**

TEST_T> **report_faults -summary**

• ATPG effectiveness = ATPG-resolvable faults / all faults

$$ATPG\_eff = \frac{DT + UD + AN + (NP \times PT\_credit)}{All\_Faults} \times 100$$

Annexe 2: Rappels du concept de test avec «scan path »

1) Le Scan path (chemin de scan)

Le ***scan path*** est introduit dans un circuit complexe afin d'améliorer la testabilité de ce circuit. Il consiste en la transformation du circuit séquentiel en un circuit combinatoire pendant le test en augmentant ainsi la contrôlabilité et l'observabilité du circuit et en diminuant le temps effectif de test. Afin d'accomplir cette tâche, on remplace les bascules normales du circuit par des bascules séquentielles scannables qui sont liées entre eles pour former une chaîne (registre de décalage), appelée chaîne de scan. Ceci est fait automatiquement par l'outil de synthèse Design Vision, les étapes seront détaillées dans la section « pratique »

Le principe du remplacement d'une bascule normale par une bascule scannable est illustré dans la figure 1. On remarque le rajout de deux nouvelles entrées sc_en (*scan enable*) et sc_in (*scan in*). De plus on dispose d'une nouvelle sortie sc_out (*scan out*). L'entrée *scan enable* est utilisée pour permettre au circuit de fonctionner en mode normal ou en mode test.

Le mode test consiste à charger des vecteurs de test dans la chaîne de scan à travers l'entrée *scan in*. Les valeurs des noeuds internes, difficilement accessibles de l'extérieur en utilisant la chaîne de scan peuvent ainsi être observées. Les valeurs seront sorties par décalage à la sortie *scan out*.

Rappel sur la méthodologie de test à l'aide du scan path :

1 Activer le mode test (sc_enable) et envoyer vers la chaîne de scan le vecteur de test (initialiser les bascules)
2 Passer en mode normal de fonctionnement et appliquer un vecteur de test aux entrées réelles du circuit
3 Observer les sorties primaries
4 Activer le clock pour une période pour charger l'état interne dans les bascules
5 Activer le mode test (sc_enable) pour sortir les résultats (en même temps faire 1)

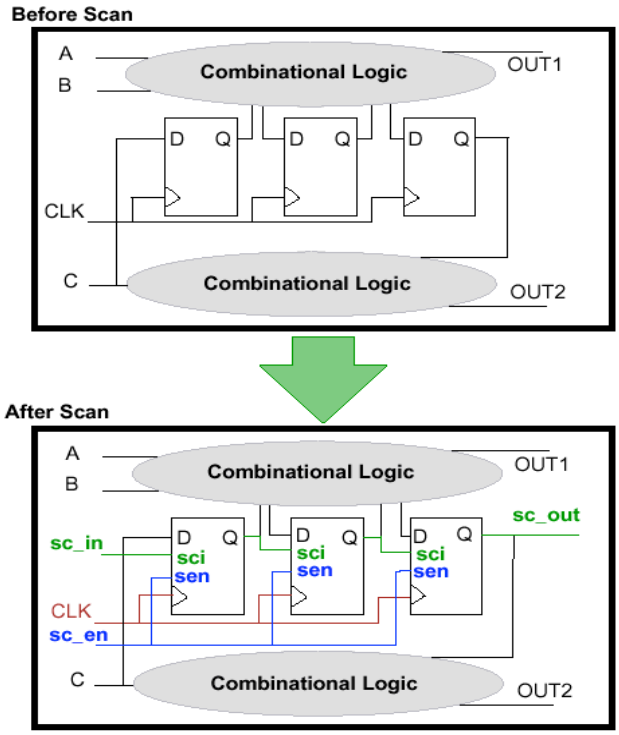


Figure 1. Insertion de DFT dans un circuit séquentiel .

Les bascules scan seront seulement de type mux-scan (voir figure 2). Ainsi, les bascules D sont remplacés par des bascules ayant dans leur structure interne un MUX supplémentaire, permettant le choix entre l'entrée originale et l'entrée de scan.

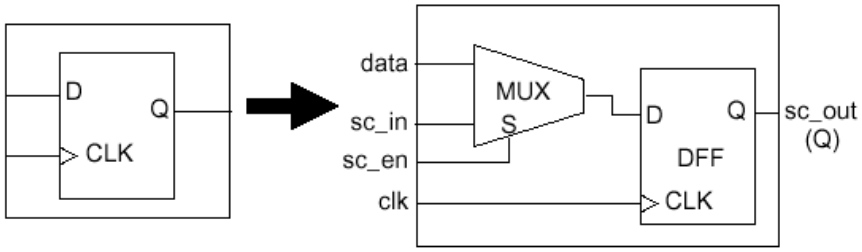


Figure 2. Transformation d'une bascule simple dans une bascule scannable

La méthodologie de travail avec l'outil de synthèse testable Design Vision (Synopsys) vous est présenté en annexe 3. Le principe d'utilisation de cet outil pour le scan y est présenté.

2) ATPG :

Il a pour but de déterminer un jeu de vecteurs de test qui assurent un taux de couverture de fautes acceptable.
Les méthodes les plus souvent utilisées pour la génération des vecteurs de test sont :

- **La génération aléatoire** : des séquences de vecteurs de test générées aléatoirement. La sélection parmi ces vecteurs est ensuite faite afin de ne garder que les vecteurs qui détectent des fautes différentes (on élimine les vecteurs qui détectent les mêmes fautes).
- **La génération déterministe** : pour chaque faute qu'il reste dans la liste de fautes, on génère un vecteur de test qui la propage à la sortie du circuit.

A la fin de la phase d'ATPG on réduit la longueur de la séquence de test générée par une opération de compactage logique.

Les séquences de vecteurs de test ainsi créées seront utilisées pendant le test de production. Parmi les tests possibles nous remarquons :
Le test logique/fonctionnel : on envoie des vecteurs de test aux entrées du circuit en observant les sorties afin de détecter les défauts liés au processus technologique: stuck-at, open, short. Ces défauts produisent une erreur logique à la sortie du circuit sous test.

- **Stuck-at :** un modèle de fautes qui assimile le défaut comme une connexion à GND or VDD : le nœud affecté ne peut plus changer de valeur logique.

L'outil de génération automatique de vecteurs de test **Tetramax (Synopsys)** vous est présenté en annexe.

Annexe 3: Description du circuit « Réveil Numérique »

Ce circuit hiérarchique est composé de neuf modules répartis sur quatre niveaux (*c.f.* figure) : TOP, CORE, COMPUTE_BLOCK, ALARM_BLOCK, ALARM_SM_2, CLOCK_GEN, COMPARATOR, TIME_BLOCK, CONVERTOR_CKT.

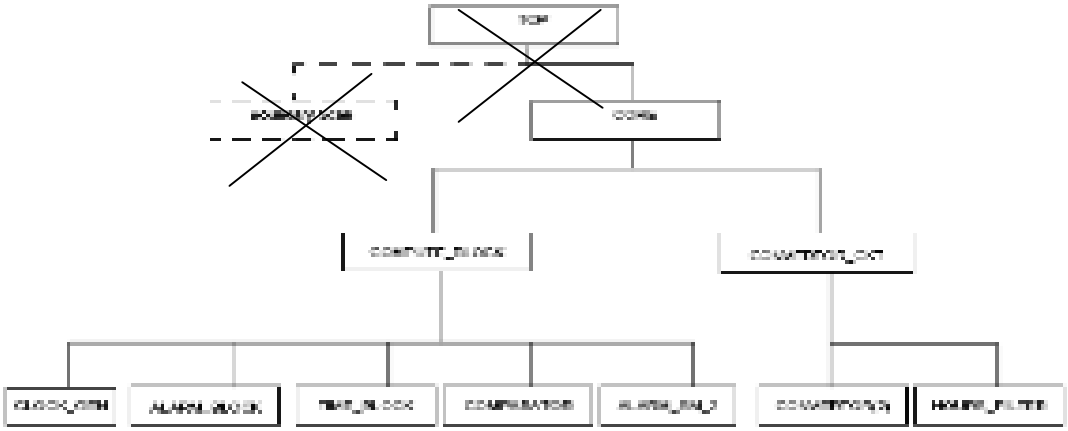


Figure: Architecture et hiérarchie du circuit ALARM_CLOCK proposé

Pour des raisons de simplicité, le module CORE est en réalité le niveau de hiérarchie le plus élevé.
Par la suite les différents modules du système sont présentés brièvement.

COMPUTE_BLOCK
Ce module permet de contrôler les réglages de l'heure courante et de l'alarme. Les informations produites par ce module sont transmises au module CONVERTOR_CKT qui les convertit en affichage sur 7 segments. Par défaut le réveil affiche l'heure courante.

ALARM_BLOCK
Ce circuit permet de contrôler l'heure de déclenchement de l'alarme.

- Les ports d'entrées sont CLK, RESETN, ALARM (autorise le réglage de l'heure de l'alarme), ENABLE (autorise la gestion sur le bus), HRS et MINS : (règle les heures/minutes de l'alarme si ALARM est vrai).
- Les ports de sorties sont ALARM_PM_OUT, HRS_OUT, MINS_OUT (réglage de am/pm, heures et minutes de l'alarme, entrée du COMPARATOR) et DISPLAY_BUS (gestion du bus pour les réglages).

Ce module est constitué de deux sous modules : une machine à état (ALARM_STATE_MACHINE) et un compteur (ALARM_COUNTER) décrites par la suite.

- **ALARM_STATE_MACHINE** : Ce module règle l'heure de l'alarme. C'est une machine à 3 états : IDLE, SET_MINUTES, SET_HOURS.
  - L'état IDLE est l'état par défaut. En fonction des entrées, l'état peut évoluer à SET_MINUTES ou SET_HOURS.
  - L'état SET_MINUTES permet d'incrémenter les minutes de l'alarme en envoyant une impulsion sur le signal de sortie MINS_OUT connecté au module ALARM_COUNTER. Tant que l'on reste dans cet état, les minutes sont incrémentées. *
  - Le fonctionnement de SET_HOURS est similaire.

- **ALARM_COUNTER**: Ce module incrémente les heures et les minutes de l'alarme, et prend en compte les réglages de type am/pm. Par défaut, la sortie HRS_OUT vaut 12, MINS_OUT vaut 0 et AM_PM_OUT vaut AM.

ALARM_SM_2 design : Ce module implémente une machine à deux états qui permet d'activer l'alarme.

- Les ports d'entrées sont CLK, RESETN, COMPARE_IN (venant du COMPARATOR, quand l'heure de l'alarme est égale à l'heure courante, le signal vaut 1), TOGGLE_ON (contrôle l'alarme).
- Les états sont IDLE ou ACTIVATE. L'état par défaut est IDLE : dès que COMPARE_IN et TOGGLE_ON valent 1, l'état courant est ACTIVATE. Tant que TOGGLE_ON est à 1, l'alarme sonne.

CLOCK_GEN Design : Ce module ne sera pas utilisé dans la cadre du projet.

COMPARATOR Design : Ce module compare l'heure courante et l'heure de l'alarme.

- Ces ports d'entrées sont ALARM_AM_PM, ALAM_HRS et ALARM_MINS (sorties de ALARM_BLOCK), TIME_AM_PM, TIME_HRS, TIME_MINS (sorties de TIME_BLOCK). Quand les entrées sont deux à deux égales, le signal de sortie RINGER (entrée de ALARM_SM_2) vaut 1.

TIME_BLOCK design : La conception de ce module est similaire à celui de ALARM_BLOCK. Il permet de contrôler l'heure courante.

- Ces ports d'entrées sont CLK, RESETN, ENABLE (contrôle le bus), HRS et MINS (règle les heures/minutes courantes si SET_TIME est vraie), SET_TIME (contrôle le réglage de l'heure courante).
- Les sorties sont AM_PM_OUT, HRS_OUT, MIN_OUT (réglage des heures, minutes et am/pm de l'heure courante, entrées du COMPARATOR), DISPLAY_BUS (gestion du bus du réveil).

Il est constitué de deux sous modules : TIME_STATE_MACHINE et TIME_COUNTER.

- **TIME_STATE_MACHINE** : Ce module règle et fait évoluer l'heure courante. C'est une machine à trois états : COUNT_TIME (état par défaut), SET_MINUTES, SET_HOURS.
    - Dans l'état COUNT_TIME, toutes les secondes une impulsion est envoyée sur le signal SECS_OUT : elle fait évoluer l'heure courante et est connectée à TIME_COUNTER.
    - Quand SET_TIME vaut 1 et MINS vaut 1, l'état courant devient SET_MINUTES. Cet état permet d'incrémenter les minutes de l'heure courante en envoyant des impulsions sur le signal MIN_OUT connecté au TIME_COUNTER. Tant que l'on reste dans cet état les minutes sont incrémentées.
    - Le fonctionnement de SET_HOURS est similaire.

- **TIME_COUNTER** : Ce module permet d'incrémenter les heures et les minutes de l'heure courante et reflète les réglages am/pm. Par défaut HRS_OUT=12, MINS_OUT=0 et AM_PM_OUT=AM.

CONVERTOR_CKT : Ce module implémente une fonction permettant de décoder une valeur binaire vers un affichage 7 segments. Il utilise deux sous modules : CONVERTOR et

HOURS_FILTER. Il utilise deux instances de CONVERTOR (une pour les heures une pour les minutes) qui effectuent la conversion, et une instance de HOURS_FILTER qui supprime le zéro des heures lorsque nécessaire (on aura « 9:00 » au lieu de « 09:00 »).

Annexe: La synthèse simple avec Design Vision (Synopsys)

Préparation : Configuration de l'outil de synthèse

Dans ces exercices nous allons utiliser l'outil de synthèse de la suite Synopsys™. Cet outil s'appelle Design Compiler™ . Là dessus, plusieurs outils sont greffés, dont l'outil de synthèse simple, l'outil de testabilité (DFT Compiler™), l'outil d'évaluation du timing (PrimeTime™), l'outil d'évaluation de la puissance consommée (PrimePower™) et plein d'autres.
L'interface graphique de cet outil s'appelle Design Vision et nous allons l'utiliser autant que possible. Cependant toutes les commandes ne sont pas accessibles via l'interface graphique. Cela nous oblige à travailler de façon mixte : commandes de script – press boutton !

La configuration de l'outil Design Vision en vue de la synthèse est réalisée par l'intermédiaire d'un fichier qui contient un ensemble de commandes prédéfinies. Ce fichier est par défaut dénommé « .synopsys_dc.setup » et il est utilisé lors de l'exécution de chaque commande liée à la lecture d'une description, d'une compilation ou d'une optimisation. Les liens entre bibliothèques logiques et physiques sont spécifiés dans ce fichier.

Pour ce TP nous utilisons la bibliothèque AMS 0.35u.

Lancer l'outil "Design Vision"

Pour lancer l'outil de synthèse logique Design Vision il suffit d'entrer la commande UNIX dans une fenêtre cmdtool ou xterm: "design_vision &". Il faudra lancer l'outil de synthèse dans le répertoire de synthèse, et s'assurer que le fichier « .synopsys_dc.setup » se trouve dans ce répertoire.

Faire une synthèse à travers le menu de l'interface Design Vision

La suite du document présente **quelques** menus accessibles à travers cette interface graphique.

Etape 1. LIRE ET ELABORER UN DESIGN

Le menu "File" permet de réaliser au moins les actions suivantes :

Analyze -> lire la source HDL, vérifier la syntaxe HDL et créer les formats objet de bibliothèque HDL. Utiliser la bibliothèque WORK. La liste des fichiers doit être ordonnée en fonction de la hiérarchie du circuit (niveaux inférieurs ou "feuilles" avant les niveaux plus élevés).

Elaborate -> produire un design en cellules génériques indépendantes de la technologie à partir d'un format HDL intermédiaire (objets générés par la commande Analyze). A cette étape, le compilateur décompte les bascules, et crée la hiérarchie. Utiliser la bibliothèque WORK.

Save -> sauvegarde en format interne sous forme de base de données (extension par défaut .ddc).

Save As -> sauvegarde en format quelconque (choix multiple: VHDL, Verilog, Edif, …)

A noter : **Read** permet de lire le fichier source dans un format VHDL, Verilog, …

L'option **Read** est équivalente aux commandes **Analyze** + **Elaborate MAIS elle sera utilisée seulement pour les fichiers de bases de données .db.**

Etape 2. Se positionner sur le module hiérarchique TOP : COMPUTE BLOCK

Etape 3. S'assurer que le design est correct

link
check_design

Etape 4. Imposer des contraintes au design

Le menu « Attributes » : Ce menu est utilisé pour imposer des valeurs/variables aux objets sélectionnés. **Placez vous sur le circuit "top" (COMPUTE BLOCK) pour appliquer une contrainte à l'ensemble du circuit.**

Specify Clock -> préciser, pour le signal choisi comme signal d'horloge, la période et les instants des fronts (montant et descendant) à l'intérieur de la période. Avant d'utiliser ce menu, sélectionner le port sur l'icône ou le schéma du composant COMPUTE BLOCK : le nom du port doit apparaître en grisé dans la fenêtre "Specify clock" qui s'ouvre. Dans cet exemple le signal d'horloge interne à selectionner est « INT_CLK ».

Optimisation Constraints -> sert à préciser des contraintes de fonctionnement autres que les Operating Conditions, Wire load, Clock.

Par exemple le sous-menu :

Design Constraints -> pour la surface (mettre toujours 0), pour la consommation dynamique ou statique, ou le fanout. On se contentera de n'imposer que des contraintes de surface.

Optimisation directives -> possibilité éventuelle d'imposer des contraintes globales sur tout le circuit ou sur une cellule particulière sélectionnée, comme par exemple mettre à plat la description, rajouter des pads… .

Etape 5. Faire une synthèse

Le menu « Design » permet d'effectuer des optimisations, des analyses du design et de produire des rapports, en lançant d'autres outils Synopsys « built-in » (estimation de timing, de puissance, etc.) :

Compile Design -> vérifie la cohérence du design, réalise un mapping sur la bibliothèque technologique sélectionnée, avec éventuellement les options de compilations suivantes (les plus importantes)

Pour une première synthèse vérifier qu'aucune des options suivantes n'est cochée.

Top level - optimisation du plus haut niveau hiérarchique sans toucher aux blocs internes.

Ungroup all – mise à plat du design. Le design sera mis à plat pour pouvoir passer au Placement Routage.

Incremental mapping – permet de repartir de la version courante en cas d'optimisations successives.

Etape 6. Faire une vérification du design

Check Design-> vérifie les erreurs de structure et les violations de timing

Etape 7. Produire les résultats de la synthèse

Menu Design, Reports

Report Design-> générer des rapports sur toutes les contraintes/valeurs imposées au design

Report Constraints -> choix multiple, vérifier les contraintes

Report Ports, Cells, Nets, Clocks, Area – obtenir des informations sur le résultat de la synthèse.

Report Power – obtenir des résultats sur la puissance consommée (types et valeurs). Attention : considérer « NETS and CELLS ». Faire "show nets histogram". Cocher aussi « traverse hierarchy at all levels »

Le menu "Timing"

Check Timing-> vérifie seulement le timing du circuit, rapport en cas de violation.

Report Timing Paths -> rapport des chemins de propagation du plus long au plus court, choix multiples, laisser toutes les options par défaut

Path Slack -> créer des histogrammes des slacks (marge d'un chemin de propagation par rapport au chemin critique). Cliquer sur une histogramme, dans la fenêtre à droite on voit les chemins et les valeurs des slacks.

Net capacitance -> créer des histogrammes des capacités de nœuds. Cliquer sur un histogramme, dans la fenêtre à droite on voit les valeurs des capacités.

Path Profile View -> visualiser le chemin critique avec le temps de propagation des portes se trouvant sur le chemin. Cliquer sur Flat dans la fenêtre de visualisation.

Pour voir effectivement le chemin critique sur le design aller dans le menu View, Highlight, Critical Path. Le schematic doit alors être ouvert, au bon endroit de la hiérarchie. Vous trouverez ceci dans le menu Report Timing Path. Pour le désactiver il suffit de faire Clear Selected.

Etape 8. Sauvegarder le circuit au format verilog ou database (DDC)

Annexe Synthèse avec insertion du Scan Path avec l'outil Design Vision (Synopsys)

La méthodologie d'insertion de scan et des points de test dans un design séquentiel à l'aide de l'outil Design Vision™ est présentée par la suite.

Pour ce faire, on effectuera une synthèse avec option de scan, une évaluation de la testabilité durant la synthèse, suivie d'une correction du design en vue de l'amélioration de la testabilité, puis d'une nouvelle évaluation de la testabilité. Ceci sera fait sur des modules bas niveau, puis on fera une propagation des changements vers le niveau hiérarchique supérieur.

Design Vision est un outil capable d'insérer des modules DFT dans un design numérique (scan, BIST, Boundary Scan).
Toutefois, étant un outil relativement sophistiqué qui s'utilise durant les phases de conception numérique avancée, il n'est pas prévu de le manipuler à travers l'interface graphique, mais plutôt en mode lignes de commandes (script).

Pour effectuer une synthèse testable suivre les conseils suivants. On **vous demande** d'exécuter les étapes de façon cognitive, en essayant vraiment de comprendre ce qu'il se passe… Et de respecter l'ordre des commandes.

Tout d'abord travailler sur Compute_block qui a déjà été synthétisé, mappé sur une technologie et dont les caractéristiques de surface, consommation et timing sont déjà connus!

Ensuite vous allez travailler sur le block CONVERTOR, purement combinatoire.
A la fin, on propagera toutes les modification au module top niveau nommé CORE.

Etape 1

Recommencer avec un environnement propre, relancer l'outil Design Vision et netoyyer tous les fichiers précédents de travail dans le répertoire TP_SCAN_INSERTION (fichiers .syn, .pvl, .mr)

Lire les fichiers VHDL du design COMPUTE_BLOCK.
Utiliser Analyse et Elaborate.

Etape 2 Se positionner sur le niveau hiérarchique le plus élevé.

```
current_design COMPUTE_BLOCK
```

Etape 3 Imposer des contraintes à votre design (clock, surface)

```
create_clock -name "clk" -period 10 -waveform { 0 5 } { U7/INT_CLK}
set_max_area 0
```

Etape 4 Vérifier qu'il n'y a pas d'erreurs d'instanciation et de mapping de signaux.
link
Etape 5 Set-up du scan et de ses signaux ;

a) Mettre en place les modèles de test et s'assurer que l'outil va en produire un pour le circuit COMPUTE BLOCK ; il sera utile lorsqu'on va chercher à le propager vers le niveau supérieur hiérarchique.

```
set test_xg_use_models true
use_test_model -true COMPUTE_BLOCK
```

b) Création de l'horloge de test à partir d'un port existant et connecté. Ce signal doit être piloté par le testeur.

```
set_dft_signal -view existing_dft -port "CLK" -type ScanClock -timing [list 45 55]
```

c) Création d'un signal de type reset asynchrone à partir d'un port existant, et connecté. Ce signal doit être piloté par le testeur.

```
set_dft_signal -view existing_dft -type Reset -port RESETN -active_state 0
```

d) Définir le protocole de test de l'horloge de test, des entrées et des sorties. Le protocole de test sera très dépendant des paramètres de test suivants.

```
set test_default_period 100 – la période du cycle de test
set test_default_delay 0 –le délai de l'application des entrées par rapport à la clock
set test_default_bidir_delay 0 --- idem pour les pins bidir
set test_default_strobe 40 – le temps de strobe des sorties
set test_stil_netlist_format {verilog}
```

La figure 1 montre la relation entre les données de test, l'horloge, et le timing d'échantillonnage. La sortie doit être stable avant le front montant de l'horloge, après que tous les changements de valeurs sur les entrées se sont propagés à travers le circuit. Dans cette figure, l'horloge est active sur le front montant.

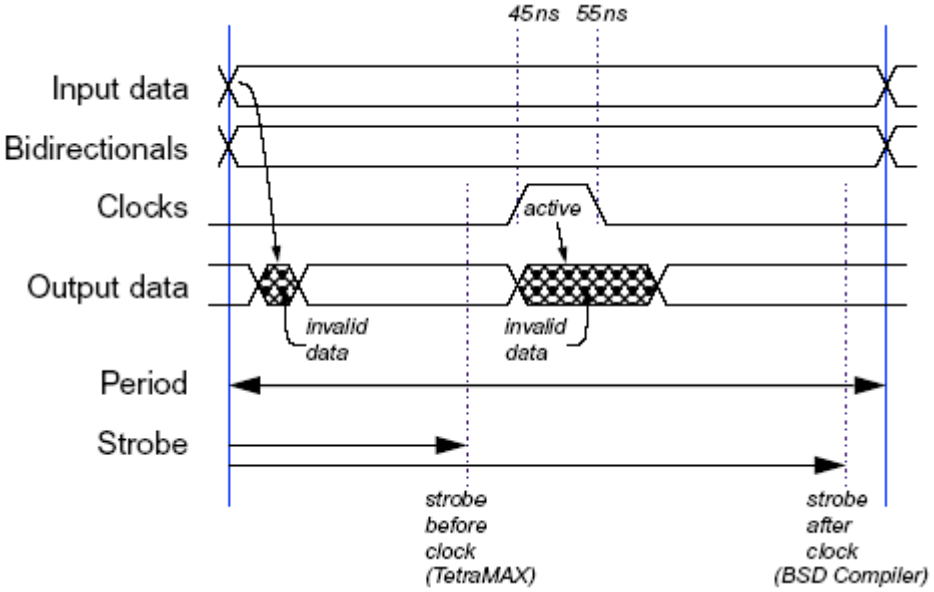


Figure 1 Chronogramme des signaux durant le test par un équipement de test externe.

e) Créer le protocole de test et préparation pour l'insertion du scan.

```
create_test_protocol
```

f) Vérifier la testabilité pre-scan.

```
dft_drc –verbose
```

Cette commande vérifie la testabilité d'un circuit. Elle permet de trouver les éventuelles erreurs de contrôlabilité et d'observabilité dans la netlist, pouvant conduire à des problèmes de testabilité. Elle permet d'identifier l'existence de signaux clock, reset générés par les blocks internes, qui ne pourront pas être pilotés de l'extérieur par le testeur.

Si cette commande rapporte des violations ; il faut chercher à les comprendre et à les analyser. Déterminer les modules qui ont des problèmes de testabilité.

Question A. Combien de violations ? _____
Question B. Pourquoi ? ____

d) Définir le type de scan et le nombre de chaines de scan qu'on souhaite avoir dans le circuit.

```
set_scan_configuration -style multiplexed_flip_flop
set_scan_configuration -chain_count 1
```

Etape 5 Synthèse avec option scan

```
current_design COMPUTE_BLOCK
compile –exact_map -scan
```

Question C. A l'aide de l'interface graphique, aller voir les cellules de scan. Quelles types de bascules ont été utilisées (enumérer 2-3)? _____

Etape 6 Création du protocole de test après insertion de scan et ré vérification de testabilité (normalement dans le casd e ce circuit on doit retrouver les mêmes erreurs pre-dft)

```
create_test_protocol
dft_drc -ver
```

Etape 7 Résolution de problèmes de contrôlabilité de la clock, qui est générée en interne. Pour ce faire, il faut rajouter de façon automatique, un point de test substituant l'horloge interne U7/INT_CLK par la clock externe. Ceci se fera en rajoutant un MUX, ayant pour signal de sélection le signal TM_FIX qui est une entrée non utilisée du block COMPUTE_BLOCK. Ce signal est un signal de type TestMode, piloté de l'extérieur par le testeur, alors que la clock, en plus d'être une clock de scan, est également une données de test !

```
set_dft_signal -view spec -type TestMode -port TM_FIX
set_dft_configuration -fix_clock enable
set_dft_signal -view spec -type TestData -port CLK
set_autofix_configuration -type clock -include_elements [get_object_name
[get_cells -hierarchical *]] -control_signal TM_FIX -test_data CLK
```

Etape 7 **Effectuer physiquement l'insertion des éléments DFT (points de test et connecter les bascules en ordre dans la chaîne de scan) en exécutant les commandes**

preview_dft –test_points all (afin de vérifier la faisabilité de la commande)
insert_dft
dft_drc - verbose

Question D. Combien de bascules apparaissent dans la chaine de scan ?_____
Question E. Combien de ports supplémentaires sont rajoutés ? _____
Question F. Quels sont les noms des ports ajoutés ?_____

Etape 8 Sauvegarde périodique

Sauvegarder, pas en mode graphique mais en mode script avec la commande suivante :

write -hierarchy -format ddc -output compute_block_scan_dft.ddc

L'outil peut crasher (raison inconnue), ce qui vous permettra de reprendre le travail à partir de ce point.

Si l'outil crashe, faire Read du fichier .ddc que vous avez sauvegarder auparavant, et passer directement à la suite. Si non, passer à la suite.

Etape 9 Préciser les entrées et sorties de scan, leur donner des attributs de type scan_in, scan_enable, etc… .

```
set_dft_signal -view spec -type ScanDataIn -port test_si
set_dft_signal -view spec -type ScanEnable -port test_se

set_dft_signal -view spec -type ScanDataOut -port SPEAKER_OUT ---
ici on décide de faire « sharing » du port de test avec un port existant
de données.

set_dft_signal -type TestMode -port TM_FIX
insert_dft ---nécessaire pour la connexion effective des entrées et
sorties
```

Etape 10 Vérification à nouveau du scan et de la testabilité et toute autre violation en relation avec le test

```
dft_drc -verbose -coverage_estimate
```

Question G. Quelle couverture de fautes est proposée? _____

Question H. Quelle est la première bascule dans la chaine de scan ? _____
Et la dernière_____ ?

Etape 11 Sauvegarde

```
change_names -rules verilog -hierarchy -verbose
write -hierarchy -format ddc -output compute_block_scan_dft.ddc
write_test_protocol -output compute_block_scan_dft.spf
write_test_model -output compute_block_scan_dft.ctldb
write -hierarchy -format vhdl -output  comp_block_Scan.vhdl
write -hierarchy -format verilog -output comp_block_Scan.v
```

Etape 12 Rapports de DFT

```
report_scan_path -view existing_dft -chain all
report_scan_path -view existing_dft -cell all
```

Générer les rapports de surface, timing et puissance.

Le module COMPUTE_BLOCK est synthétisé, avec le scan et les points de test, et vérifié (il ne doit plus y avoir de violations).

La suite du TP consiste à récupérer le module CONVERTOR (qui est seulement combinatoire) le compiler, le mapper sur la même technologie C35_CORELIB et le connecter au COMPUTE_BLOCK. Le module top s'appelle CORE. Suivre les étapes suivantes :

Etape 13 Lire et élaborer le circuits CONVERTOR.vhd et CORE.vhd

Etape 14 Se placer au niveau du module hiérarchique supérieur, CORE, imposer la clock et les contraintes de clock normale.

```
current_design CORE
create_clock -name "clk" -period 10 -waveform { 0 5  }  { CLOCK }
```

Etape 15 Propager vers le top (CORE) le modèle de test des modules hiérarchiquement inférieurs et spécifier les signaux de test du niveau supérieur.

```
use_test_model -true CORE

set_dft_signal -view existing_dft -port "CLOCK" -type ScanClock -timing [list 45 55]
set_dft_signal -view existing_dft -type Reset -port RESETN -active_state 0
```

Etape 16 Compiler en incremental

```
Compile – exact_map –incremental_mapping
```

Etape 17 Créer le protocole de test, spécifier les signaux de test, connecter tous les signaux

```
set_dft_signal -view existing_dft -type ScanDataIn -port TEST_SI
set_dft_signal -view existing_dft -type ScanEnable -port TEST_SE
set_dft_signal -view existing_dft -type ScanDataOut -port SPEAKER
set_dft_signal -view existing_dft -type TestMode -port TEST_MODE

create_test_protocol
preview_dft
insert_dft
```

Etape 18 Vérifier la testabilité

```
dft_drc -coverage_estimate --- ici on observe beaucoup de fautes qui passent des détectables à AU (ne pas se préoccuper)
```

Etape 19 Sauvegarder en verilog et VHDL et générer le fichier de test STIL

```
change_names -rules verilog -hierarchy -verbose
write_test_protocol -out alarm_clock.spf
write -hierarchy -format vhdl -output alarm_clock.vhd.vhdl
write -hierarchy -format verilog -output alarm_clock.v
```

Etape 20 Rapports de surface, timing et puissance.

Question 12. Combien d'augmentation de la surface du circuit ?_____ %
Question 13. Quel impact sur le timing ? _____%
Question 14. Quel impact sur la puissance consommée ?_____%
Question 15. Quelle est la couverture de test de ce circuit ?_____%

Attention de ne pas écraser les netlists sauvées avant insertion de scan, elles peuvent servir par la suite.

Annexe Génération de vecteures de test (ATPG) avec l'outil Tetramax (Synopsys) pour le circuit Alarm Clock

Après avoir sourcé le fichier .bashrc_tetramax2013 lancer l'outil Tetramax avec la commande tmax dans le répertoire TP_SCAN_INSERTION.

Lire la bibliothèque ATPG

Onglet Netlist – choisir corelib_pour_ATPG.v. Cocher Library modules et Master Modules, Conservative Mux – None

Lire la netlist du circuit alarm clock

Onglet netlist – netlist_alarm_clock.v (donner le noms que vous avez utilisé lors de la synthèse avec insertion du scan)

Build – Top module CORE. Set Build Options – cocher Master slave into DFF (OK and RUN)

Ne pas tenir compte des warnings si ceux-ci concernent des entrées/sorties/signaux internes non connectés. Même chose pour des warnings relatifs à la règle N16. SINON refaire la synthèse et reprendre au début avec Tmax.

DRC – charger le fichier alarm-clock.spf

ATPG –

General ATPG Settings – Capture cycles, mettre 9

Cocher Enable Full_Seq ATPG

Remove NDetects

Fault models – stuck

Add all faults

Pattern sources – internal

Click on Full Seq Settings (onglet) - set merge efforts low

Colonne RUN clicker sur FULL SEQ (vous devez obtenir une couverture >96%)

Verifier, s'il y en a les fautes UD! Analyser les autres catégories.

Write Patterns

Donner un nom de fichier (patterns_alarm_clock.v)

Cocher pattern source - internal

File format STIL

Scan format- serial

Write test bench

A partir du fichier de patterns généré précédemment, produire un testbench.

Simulation

Cocher Full Sequential

Cocher Pattern Source – Internal

Cocher Insert fault – Scan Cell, donner un numéro de cellule et un modèle de stuckat

Run