



Encadré par Régis Leveugle

Axel Baldacchino

Tom Désesquelle

Pierre-Olivier Guessard

**Analyse et Réalisation d'un
Système Complexe :
Protection contre les perturbations dans un FPGA**

2022-2023

INP Phelma, 3AMT



Remerciements

Nous voulons remercier tout d'abord Régis Leveugle qui malgré des soucis personnels, a pu nous aider et répondre à nos questions tout au long du projet. Nous souhaitons aussi remercier le reste de l'équipe enseignante pour leur disponibilité ainsi que le personnel du laboratoire du CIME qui nous a permis d'effectuer notre première partie de projet dans leur enceinte. Enfin, un grand merci à toute l'équipe pédagogique qui nous a suivi durant ces 3 années à Phelma.

Table des matières

Glossaire	5
1. Introduction.....	6
1.1 Contexte	6
1.2 Objectif	6
1.3 État de l'art.....	6
2. Définition du cahier des charges	8
2.1 Axes de réflexion	8
2.2 Solution retenue.....	8
2.3 Outils employés	9
3. Études et réalisation.....	10
3.1 Approche de la redondance d'un circuit	10
3.1.1 Décomposition du fichier de netlist	10
3.1.2 Ajout manuel de redondance.....	11
3.2 Réalisation	13
3.2.1 Génération du bloc de vote.....	13
3.2.2 Estimation de la place restante dans le FPGA	15
3.2.3 Script de redondance	15
3.2.4 Stratégies de combinaison des LUTs	16
3.2.5 Automatisation.....	19
3.2.6 Interface Graphique	32
4. Résultats	34
5. Plan de validation	37
6. Axes d'amélioration.....	40
7. Gestion de projet.....	41
7.1 Méthodologie de travail	41
7.2 Planning - Répartition des tâches.....	41
7.2.1 Planning prévisionnel	41
7.2.2 Planning réel.....	42
8. Preuve par compétence	43
9. Conclusion	44
10. Bibliographie.....	45

Figure 1: Principe du TMR et table de vérité d'un voteur	7
Figure 2 : Description d'une netlist	11
Figure 3 : Cellule TOP avant redondance	12
Figure 4 : Cellule TOP après redondance	12
Figure 5 : Circuit initial.....	13
Figure 6 : Circuit après redondance	13
Figure 7 : Principe d'implémentation des voteurs	14
Figure 8 : Design d'un voteur à 3 entrées	14
Figure 9 : Algorigramme de voterNGenerator.py	14
Figure 10 : Simulation fonctionnelle d'un voter à 3 entrées.....	15
Figure 11 : Concept du LUT6_2	17
Figure 12 : Possibilité d'utilisation du LUT6_2	17
Figure 13: Algorigramme de CellsCombining.py	18
Figure 14: Algorigramme de internet.py.....	19
Figure 15: Algorigramme de constraintsGenerator.py	20
Figure 16 : Pinout	21
Figure 17: Algorigramme de lecture du pinout	21
Figure 18: Algorigramme du tri des PADs utilisés	22
Figure 19 : Redondance des contraintes de propriétés	23
Figure 20 : Algorigramme de redondance des propriétés	23
Figure 21 : Redondance des contraintes de placement.....	24
Figure 22 : Algorigramme de redondance du placement.....	25
Figure 23 : Redondance des contraintes de timing.....	26
Figure 24 : Algorigramme de redondance du timing	26
Figure 25 : Redondance des signaux du banc de test	27
Figure 26 : Ecriture du DUT après redondance	28
Figure 27 : Redondance des stimuli du banc de test.....	28
Figure 28: Algorigramme de la redondance du banc de test.....	29
Figure 29 : Algorigramme d'automatisation des scripts	31
Figure 30 : Algorigramme de l'interface utilisateur	32
Figure 31 : Interface utilisateur	33
Figure 32 : Simulation fonctionnelle	34
Figure 33 : Simulation fonctionnelle post-implémentation	34
Figure 34 : Simulation de timing post-implémentation	35
Figure 35 : Comparaison des différents résultats d'implémentation	35
Figure 36 : Simulation fonctionnelle sans injection de faute	36
Figure 37 : Simulation fonctionnelle avec injection de faute.....	36
Figure 38 : Diagramme de Gantt prévisionnel	42
Figure 39 : Diagramme de Gantt réel.....	42

Glossaire

EDIF :	Electronic Design Interchange Format
FPGA :	Field-Programmable Gate Array
GUI :	Graphical User Interface
IO :	Input-Output - Entrée/Sortie
LUT :	Look-Up Table
NMR :	N-time Modular Redundancy
TBCFF :	To Be Continued For Friday
TCL :	Tool Command Language
TMR :	Triple Modular Redundancy

1. Introduction

1.1 Contexte

De nos jours, les technologies évoluent sans cesse, mais pour ce faire, cela passe par l'évolution de la complexité des circuits électroniques. Ces circuits électroniques comportent de plus en plus de transistors, tout en ayant comme consigne de diminuer leur taille. Avec cette augmentation de transistors, les fautes ou défaillances ont d'autant plus de chance d'apparaître sur ces circuits, et donc de conduire à une panne. Pour éviter cette panne justement, nous allons vous parler de tolérance aux fautes.

Qu'est-ce que la tolérance aux fautes pour un FPGA ?

La tolérance aux fautes pour un FPGA fait référence à la capacité d'un FPGA à fonctionner de manière fiable malgré la présence de défauts ou de fautes. Cela peut être important dans certaines applications critiques, telles que les systèmes de contrôle d'aviation, où une faute pourrait avoir des conséquences graves si cela conduit à une panne.

La tolérance aux fautes s'applique principalement aux domaines critiques dont la vie humaine peut être en jeu tel que le domaine spatial, aéronautique ou encore l'automobile. Cependant, elle peut être aussi présente dans des domaines bien moins critiques, dans le cas de supercalculateur par exemple, si une ou plusieurs défaillances se présentent lors d'un calcul conséquent, alors il faut pouvoir tolérer ces fautes si elles ne sont pas critiques afin de ne pas stopper ces calculs.

1.2 Objectif

L'objectif du projet est d'étudier tout d'abord les différentes techniques de tolérance aux fautes afin d'en choisir une ou plusieurs qui nous seraient utiles, faire fonctionner un système sans tolérance aux fautes, puis mettre en œuvre une ou plusieurs techniques afin de les implémenter sur FPGA, puis l'optimiser. Nous comparerons alors le nombre de LUTs ajoutés dans le FPGA avec et sans tolérances aux fautes, puis nous le comparerons après optimisation en comparant toujours le nombre de LUTs implantés. Nous étudierons l'influence de ces techniques sur nos systèmes, en analysant par exemple la place occupée sur le FPGA ou encore l'impact sur le timing de la clock.

Nous allons tout d'abord effectuer une étude sur les solutions proposées actuellement dans l'industrie, suite à cela, nous expliquerons pourquoi le TMR a été retenu, et quelle stratégie nous allons utiliser afin de l'implanter. Enfin nous expliquerons de quelle manière fonctionne notre méthode de tolérance aux fautes et nous détaillerons les résultats obtenus.

1.3 État de l'art

Lorsque nous avons commencé ce projet, nous n'avions aucune idée de la méthode que nous allions utiliser. Nous avions vu en cours certaines techniques (Hamming, TMR) cependant nous ne connaissions pas leur spécificité. Nous avons donc dû effectuer des recherches sur ce qui était utilisé sur les FPGA, et quels étaient chacun leurs avantages et leurs inconvénients.

La première solution est la redondance, qui est une méthode qui a pour avantage d'être rapide et efficace.

La redondance consiste à dupliquer les blocs et fonctions logiques ciblées puis générer et comparer leurs sorties. Si une différence est notée entre le bloc originel et le bloc dupliqué, un signal d'alarme est créé.

De même pour Hamming, qui est une méthode de correction d'erreur de transmission de données, il peut détecter une erreur dans un message binaire, le localiser et le corriger à l'aide d'une redondance de code. Cependant, même s'il est infaillible lorsque bien utilisé, le code de Hamming ne permet pas de détecter plus de deux erreurs dans un message, et peut seulement en corriger une.

Enfin, nous avons le TMR (Triple Modular Redundancy) qui est aussi une méthode basée sur la redondance, et qui est l'une des plus utilisées.

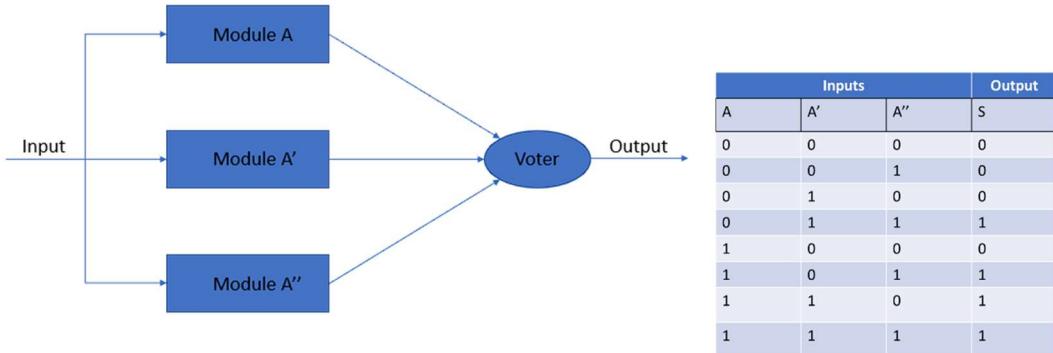


Figure 1: Principe du TMR et table de vérité d'un voteur

Elle consiste à tripler un bloc/une fonction du circuit, les 3 blocs ayant soit la même entrée, soit 3 entrées mais toutes identiques, le voteur décidera alors en fonction du résultat majoritaire quelle sera la sortie.

Elle est souvent utilisée dans les systèmes critiques dont il y a une très forte chance de défauts. L'un de ces avantages majeurs est sa robustesse car avec 3 modules dupliqués et le voteur, il y a très peu de chances que l'erreur ne soit pas détectée.

L'un des désavantages de cette solution est la place prise sur la carte, vu qu'il faut dupliquer les LUTs, on perd beaucoup d'espace et aussi notre timing est plus conséquent. Cependant des optimisations sont possibles afin de limiter cette perte d'espace.

2. Définition du cahier des charges

2.1 Axes de réflexion

Après l'étude des documents scientifiques ([1], [2], [3], [4]) sur les méthodes utilisées pour la protection contre les perturbations dans un FPGA, le TMR fut la méthode retenue car cette méthode de redondance est très fiable et très utilisée sur les FPGA, ce qui veut dire qu'elle est implantable sur un très grand nombre d'entre eux. La redondance quant à elle, même si elle est assez simple à mettre en œuvre, ne permet pas de corriger une faute, et indique simplement si une erreur s'est introduite dans notre circuit à travers un signal d'alarme.

La technique de Hamming fut aussi une piste très intéressante, étant un code correcteur qui peut localiser une erreur dans un message binaire, elle aurait pu être utile en vue d'une comparaison avec le TMR afin de voir quelle méthode est la plus efficace en cas d'erreur injectée. Cependant, nous nous sommes vite rendu compte que la méthode de Hamming ne s'utilise que pour les parties séquentielles du circuit, et dans le cas d'un circuit complet, cette méthode n'est pas aussi efficace qu'une méthode par redondance.

2.2 Solution retenue

Le but de ce projet est de mettre en place une solution permettant d'implémenter un circuit tolérant aux fautes dans un FPGA tout en optimisant l'espace occupé par le circuit final sur la cible. Après analyse de plusieurs techniques de tolérance aux fautes, nous avons retenu la triplication (TMR). Cette technique est très utilisée dans les domaines tels que l'aéronautique ou le spatial ce qui caractérise sa robustesse et sa fiabilité.

Pour mettre en œuvre cette solution le cahier des charges à suivre est le suivant :

- construire un algorithme capable de tripler un circuit complexe aussi bien à partir des fichiers sources du circuit ou de sa netlist
- réduire la taille du circuit implémenté sur FPGA en développant un algorithme d'optimisation de placement du circuit triplié
- automatiser au l'outil de triplication et d'implémentation au maximum, ne nécessitant pas l'intervention humaine. Cela passe par le développement d'une interface graphique pour que l'expérience utilisateur soit la plus agréable possible.
- développer un outils qui s'adapte en fonction de la cible FPGA choisie
- optimiser l'outil en proposant à l'utilisateur de choisir les fonctions de son circuit à rendre tolérante en priorité
- vérifier que le circuit implanté après redondance est fonctionnel avec injection de faute en son sein

Tout l'enjeu du projet a été de prendre en compte l'augmentation de l'espace occupé par le nouveau circuit sur le FPGA, après la phase de triplication pour l'optimiser et le rendre plus compact. En effet, sans optimisation, l'utilisation d'une technique de

tolérance aux fautes comme le TMR, multiplie au minimum par un facteur trois la taille du circuit, si nous négligeons l'ajout des blocs de vote dans le nouveau circuit.

2.3 Outils employés

Concernant l'outil de synthèse/implémentation/simulation, Xilinx Vivado sera utilisé. Par conséquent, les netlist générées par l'outil sont de type EDF (.edf). L'outil de triplication développé dans le cadre de ce projet est alors uniquement basé sur la structure d'une netlist générée par cet outil d'implémentation.

Tous les FPGA de toutes les familles Xilinx peuvent être utilisés pour implémenter un circuit tolérant aux fautes en utilisant l'outil développé et décrit dans ce document.

D'autre part, le langage Python est un langage haut niveau, pratique et simple de prise en main lorsqu'il s'agit de manipulation d'un grand nombre de données ou de parsing de fichiers. C'est pour cette seconde raison que les scripts de triplication, d'optimisation d'espace et de génération de fichiers ont été développés exclusivement en Python.

Le logiciel Xilinx Vivado peut être utilisé en ligne de commande en mode TCL. Ce langage sera utilisé dans la phase d'automatisation de l'outil.

3. Études et réalisation

Cette partie décrit les processus de recherche et de réalisation menés pour réaliser l'implémentation de redondance dans un circuit FPGA. La phase d'étude vise à comprendre comment la redondance va pouvoir être réalisée. La phase de réalisation vise à automatiser la redondance.

3.1 Approche de la redondance d'un circuit

Cette partie décrit les processus de recherches qui permettent de confirmer que la redondance du circuit est réalisable. La redondance peut être effectuée au niveau RTL et au niveau netlist. Réaliser la redondance d'un circuit décrit sous forme de netlist permet d'appliquer la redondance à tous les circuits, sans nécessité de posséder les fichiers RTL. La suite de cette partie décrit comment est composée une netlist et comment l'éditer pour ajouter de la redondance au circuit final.

3.1.1 Décomposition du fichier de netlist

Une première étape consiste à comprendre comment est composée une netlist. Pour ce faire, il faut préalablement réaliser un circuit RTL, puis générer sa netlist. Dans cette approche, il paraît nécessaire de partir d'un circuit RTL pour maîtriser le design pour comprendre plus facilement la composition de la netlist : plus le circuit RTL est simple, plus la netlist sera simple à décomposer. La génération de la netlist depuis le logiciel Vivado de Xilinx se fait avec la commande TCL : `write_edif PATH/top.edf`. La netlist est écrite au format EDIF (Electronic Design Interchange Format), couramment utilisé pour décrire des circuits intégrés. La netlist se décompose comme le démontre la Figure 2.

Le fichier EDIF se compose de plusieurs sections, chacune décrivant une partie différente de la conception du circuit.

Tout d'abord, la section “**EDIF**” contient des informations générales sur le fichier, comme la version d'EDIF utilisée et les informations sur l'auteur. Ensuite, il y a une section qui décrit les cellules de la **librairie HDI_PRIMITIVES** qui vont être utilisées dans le design. Ces cellules sont principalement des Look-Up Tables (LUT) de différentes tailles et des bascules.

Ensuite, une fois toutes les cellules de HDI_PRIMITIVES déclarées, la netlist décrit toutes les cellules de la **librairie WORK**. Ces cellules correspondent aux fichiers RTL qui ont été utilisés dans la réalisation du design. La description des cellules comporte plusieurs sections :

- une section “**INTERFACE**” contient des informations sur les ports d'entrée et de sortie de la cellule décrite dans le fichier. Cette section décrit les ports de signal de chaque composant et leur direction (entrée ou sortie). Elle peut également inclure des informations sur les propriétés électroniques des ports, telles que la tension d'alimentation.
- une section “**CONTENTS**” contient des informations détaillées sur les composants utilisés dans le circuit. Elle contient les informations relatives aux **instances** de composants, et peut également inclure des informations sur les

propriétés des composants telles que les caractéristiques de tension et de courant, ou leur initialisation. Cette section contient également des informations sur les relations entre les composants, comme les **connexions** électroniques. Les connexions électroniques décrivent les connexions physiques entre les différents composants du circuit.

- Enfin, une section “**DESIGN**” contient des informations générales sur la conception du circuit. Elle contient principalement l’information sur le nom du circuit.

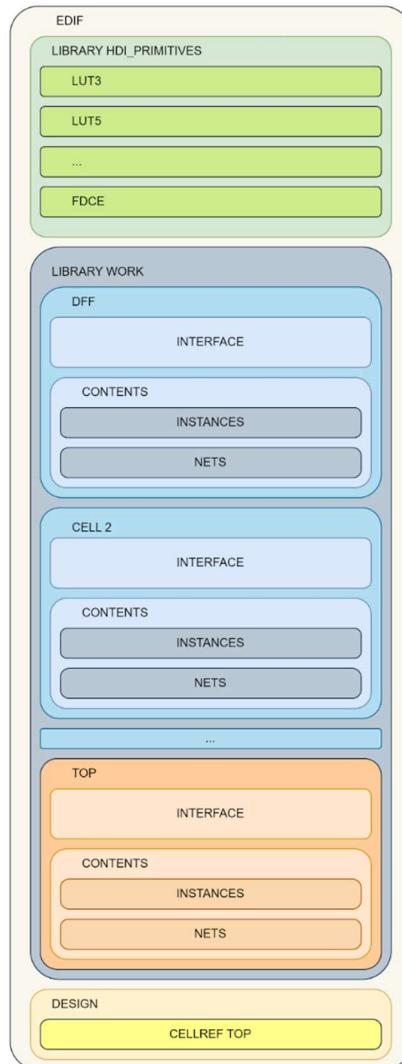


Figure 2 : Description d'une netlist

3.1.2 Ajout manuel de redondance

L’objectif de cette approche est de vérifier qu’il est possible d’éditer le fichier de netlist pour y ajouter de la redondance dans le circuit initialement décrit.

Pour s’y prendre, il faut éditer la cellule de référence du design (Figure 2). Cette cellule se trouve dans la librairie WORK. C’est cette cellule (Figure 3) qui contient tous les sous-circuits du design réalisé.

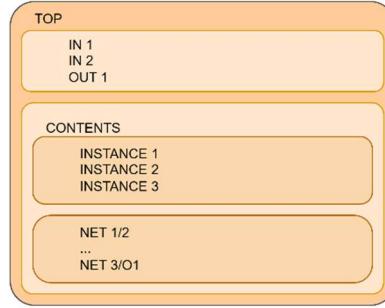


Figure 3 : Cellule TOP avant redondance

Il faut commencer par éditer la section “INTERFACE” triplant les entrées et sorties de la cellule, en modifiant leurs noms. Une façon astucieuse de nommer les interfaces triplées est d’ajouter une marque de copie comme `_TMRi` au nom initial pour s’assurer que ce label n’existe pas dans le design initial. Dans un second temps, il faut tripler les instances dans la section qui convient. Enfin, il faut aussi tripler les connexions entre les cellules. Il est important de veiller à éditer le nom des instances et des connexions triplées, sans quoi la redondance n’a pas de sens (plusieurs cellules/plusieurs connexions ne peuvent pas être décrites par un même label).

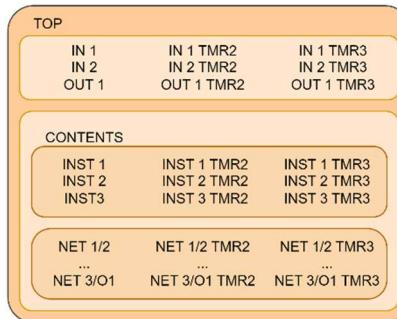


Figure 4 : Cellule TOP après redondance

De cette manière (Figure 4), deux copies du circuit initial ont été ajoutées. Pour vérifier que la redondance est effective dans le fichier de netlist édité, il est nécessaire de créer un projet Vivado post-synthèse. Dans ce projet post-synthèse, il suffit de spécifier la netlist éditée et d’ouvrir le schéma correspondant au SYNTHESIZED DESIGN. Si la copie a correctement été effectuée, le design affiché doit correspondre à trois fois le design initial.

Pour illustrer ce processus d’approche de redondance, la Figure X5 décrit un circuit initial.

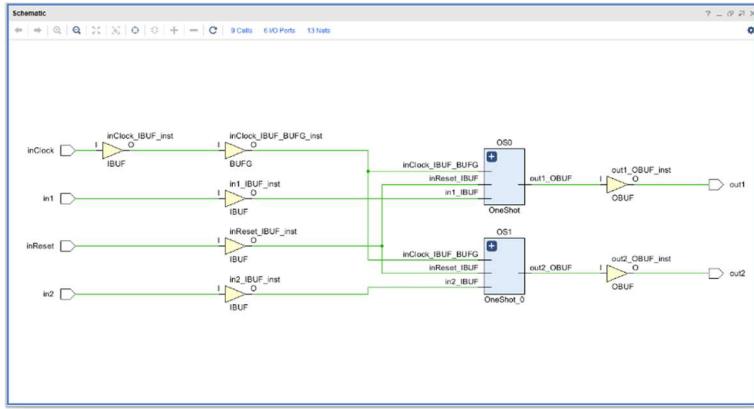


Figure 5 : Circuit initial

Après édition de la netlist correspondant au circuit décrit en Figure 5, la Figure 6 est obtenue en ouvrant le schéma de synthèse. Cette figure montre que la redondance du circuit a été effectuée.

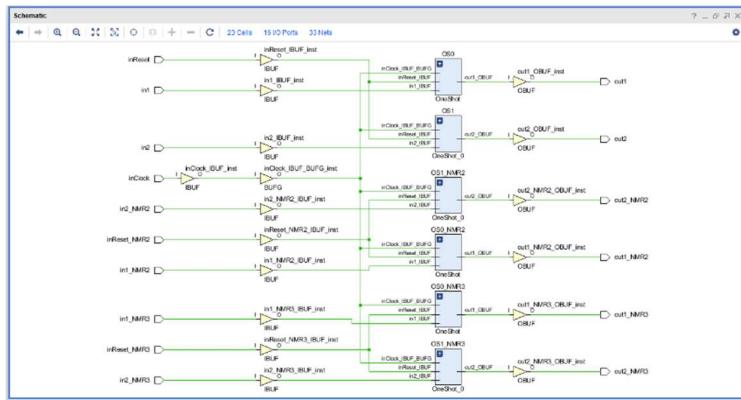


Figure 6 : Circuit après redondance

Cette phase d'étude permet de conclure que la modification d'un fichier de netlist va permettre d'ajouter de la redondance dans le circuit.

3.2 Réalisation

Désormais, il est question d'automatiser le processus de redondance du circuit. Pour ce faire, plusieurs scripts ont été réalisés pour automatiser différentes tâches dans le processus de redondance. Ces différents scripts vont permettre, entre autres, de générer un voteur à N entrées dans le cadre de NMR (N-Module Redundancy), d'effectuer la redondance du circuit, d'optimiser l'espace disponible dans le FPGA ou encore d'automatiser toute l'exécution du processus de redondance.

3.2.1 Génération du bloc de vote

Lors de la redondance d'une cellule du circuit, il faut voter majoritairement la sortie de chacune ces cellules. Ce vote majoritaire (Figure 7) a pour effet de masquer une potentielle faute en amont du voteur.

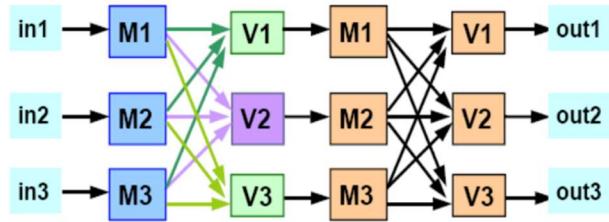


Figure 7 : Principe d'implémentation des voteurs

Dans l'optique d'automatiser le processus de redondance d'un circuit, il est utile de connaître au préalable le résultat de synthèse d'un voteur de taille N. Ce résultat de synthèse va être extrait pour être ajouté dans la netlist du circuit à modifier.

Le principe du voteur est de comparer (porte AND) une à une ces entrées, puis spécifier (porte OR) la valeur majoritaire. À titre d'illustration, la Figure 8 décrit le design d'un voteur de taille 3.

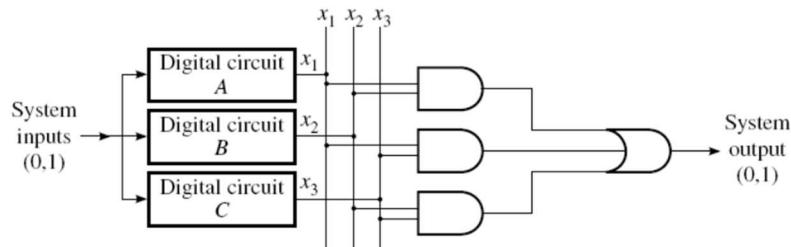


Figure 8 : Design d'un voteur à 3 entrées

Un script Python ‘voterNGenerator.py’ permet donc de générer un fichier VHDL décrivant le fonctionnement d'un voteur à N entrées, avec N un nombre impair. Ce script prend 2 arguments : le fichier *.vhd dans lequel la description matérielle doit être écrite et N, le nombre d'entrées du voteur.

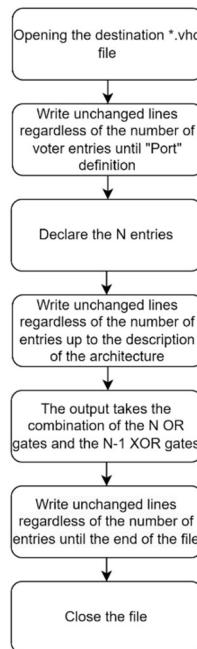


Figure 9 : Algorigramme de voterNGenerator.py

À la suite de l'exécution de ce script (Figure 9), réaliser une synthèse de ce design permet de connaître les cellules qui composent un voteur de taille N. Ces éléments de synthèse doivent être récupérés et inclus dans le design à modifier.

Auparavant, le voteur a été validé en simulation fonctionnelle (Figure 10). Les entrées du voteur sont représentées par les signaux `in1`, `in2` et `in3`. La sortie du voteur est représentée par le signal `V`. Un signal `A` contenant le résultat attendu a été ajouté au testbench, pour comparer avec le signal de sortie du voteur

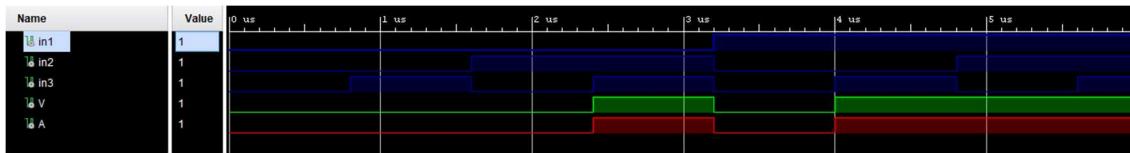


Figure 10 : Simulation fonctionnelle d'un voter à 3 entrées

La sortie du voteur correspond en tous points aux résultats attendus en fonction des différentes valeurs d'entrées du voteur. Le fonctionnement du voteur est donc validé, et il peut être utilisé pour réaliser la redondance d'un circuit.

3.2.2 Estimation de la place restante dans le FPGA

En utilisant la méthode de la redondance pour rendre un circuit tolérant aux perturbations, la place occupée par celui-ci va augmenter. Dans notre cas, si les optimisations de placement sont négligées, la place occupée par le circuit généré par la redondance est trois fois supérieure à la place occupée par le circuit initial. Sachant ça, le but de cet algorithme est de savoir, avant de construire le circuit redondant aux perturbations, si la place restante dans le FPGA est suffisante pour accueillir le nouveau circuit.

Cet algorithme se base sur le rapport d'utilisation des slices du FPGA.

3.2.3 Script de redondance

Le script de redondance a pour but d'implémenter la solution de TMR souhaitée. Ce script reprend le principe étudié dans la partie **3.1.2 Ajout manuel de redondance**. Le script sait détecter les différentes sections de la netlist (Figure 2).

Il commence par récupérer le nom du circuit et le stocke dans une variable `top`. Ensuite, il lit entièrement la librairie `HDI_PRIMITIVES`. Si cette librairie ne contient pas la cellule utile pour le voteur (LUT3 dans le cas du TMR), il ajoute la déclaration de la cellule correspondante dans cette librairie.

Par la suite, il lit la librairie `WORK`. Pour chaque cellule, le script regarde s'il s'agit de la cellule comportant le circuit final. Si ce n'est pas le cas, il stocke les informations utiles de la cellule en cours de lecture. Les informations stockées correspondent aux noms de la cellule, aux noms de ces entrées/sorties, au nombre de sorties de la cellule.

S'il s'agit de la cellule comportant le circuit final, le script stocke les informations sur les entrées/sorties de cette cellule et effectue une redondance dessus en appliquant

une marque de redondance telle que ‘_NMRi’. Ensuite, le script gère les instances de la cellule. Pour chacune d’elles, il sauvegarde dans une liste le nom de l’instance, puis effectue une redondance de l’instance en ajoutant la marque de redondance et ajoute des voteurs en fonction du nombre T de sorties de l’instance (tels que le nombre de voteurs ajoutés pour une instance vaille $N \times T$). Enfin, le script décrypte les connexions entre les différentes cellules. Pour chaque connexion, il détecte quelles sont les deux cellules interconnectées et le nom du signal qui les connecte, sépare ces deux cellules et ajoute une connexion avec un voteur entre elles. Par la même occasion, il fait de la redondance sur les connexions avec les instances ajoutées. Une fois la fin de la librairie WORK atteinte, l’étape de redondance est terminée et le script se termine en recopiant les dernières lignes telles qu’elles sont.

3.2.4 Stratégies de combinaison des LUTs

La réalisation de TMR est très coûteuse en termes de surface. Pour une surface initiale donnée, le TMR requiert, en théorie, une surface plus que trois fois plus grande (trois fois le même circuit + voteurs). Cependant, ce constat peut être amélioré. Pour diminuer la surface occupée par le circuit après redondance, il est possible de compacter les LUTs entre eux.

L’étape de synthèse permet de regrouper des LUTs ensemble pour économiser de l’espace. La synthèse regroupe, par défaut, les éléments qui partagent les mêmes signaux. Pour économiser encore plus d’espace, il existe également des techniques de ‘packing’ pour combiner des LUTs de manière plus fine.

Pour utiliser une des techniques de ‘packing’, il est nécessaire de récupérer les informations sur les cellules primitives (LUT, FDCE, etc.) qui constituent le design. Ces informations sont disponibles à l’ouverture du design dans Vivado. Ces informations doivent être stockées dans un fichier temporaire pour ajouter des contraintes sur la combinaison des LUTs dans le design.

3.2.4.1 Étude des LUTs

Chaque slice du FPGA est composé de quatre LUT6_2. Ce type de LUT (Figure 11) dispose de 6 entrées et de 2 sorties. En regardant plus en détail sa composition, il s’agit en fait de deux LUT5 qui partagent les mêmes entrées (I0, I1, I2, I3, I4) et ont une sortie différente (O5, O6) chacun, pilotée par une entrée sélective I5.

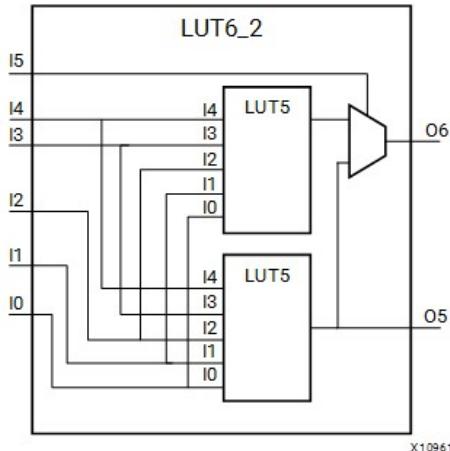


Figure 11 : Concept du LUT6_2

Aussi, il est bon de savoir que ces deux LUT5 n'ont pas l'obligation d'utiliser toutes leurs entrées. C'est-à-dire qu'ils peuvent être configurés comme des LUTs avec un nombre d'entrées inférieures. Il est alors possible de compacter deux LUTs qui n'ont pas de signaux en commun ensemble, pourvu que le nombre total d'entrées de ces deux LUTs soit strictement inférieur à six. La Figure 12 décrit, à titre d'exemple, le 'packing' d'un LUT3 et d'un LUT2 dans une même cellule LUT.

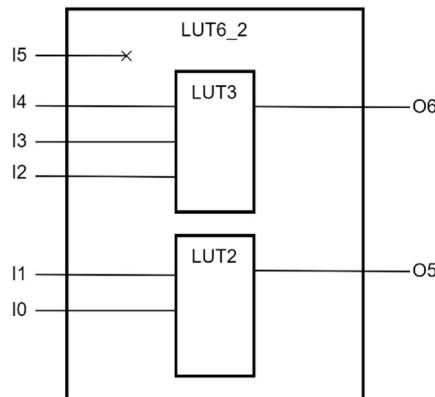


Figure 12 : Possibilité d'utilisation du LUT6_2

Pour compacter deux LUTs entre eux, il faut spécifier les contraintes suivantes dans un fichier de contraintes :

- `set_property LUTNM LUT_group%i% [get_cells PRIMITIVE1]`
- `set_property LUTNM LUT_group%i% [get_cells PRIMITIVE2]`

La suite propose une façon de générer ces contraintes de 'packing' pour optimiser l'espace occupé dans le FPGA par la redondance du circuit.

3.2.4.2 Combiner les LUTs en maximisant les LUT5

La stratégie d'optimisation des LUTs consiste à regrouper des LUTs deux à deux en essayant de former un maximum de LUT6_2 avec 5 entrées.

Tout d'abord, le `script` va récupérer les informations sur les primitives qui constituent le circuit avec redondance et initialiser un dictionnaire qui contiendra le type de cellules

utilisées et leurs noms. Ensuite, le **script** va lire le fichier temporaire contenant les types de cellules et leurs noms pour classer ces informations dans le dictionnaire créé. Le **script** poursuit en supprimant du dictionnaire toutes les cellules qui ne sont pas des LUTs, et qui sont des LUT6_2, des LUT6 et des LUT5 : il n'est pas possible d'optimiser ces cellules. Le **script** peut enfin procéder à l'optimisation des LUTs. Il commence par traiter les LUT4 qui doivent être associés avec des LUT1 dans la mesure du possible. Dès qu'il n'est plus possible d'associer des LUT4 avec des LUT1, le **script** traite les LUT3 qui doivent être associés avec des LUT2, puis avec des LUT1 s'il n'y a plus de LUT2 disponibles. Dès que les LUT3 ne peuvent plus être optimisés, le **script** tente d'associer des LUT2 entre eux, puis avec un LUT1 si possible. Finalement, le **script** tente d'associer les LUT1 deux à deux. La Figure 13 schématisé l'algorithme décrit. À chaque association de deux LUTs une contrainte les reliant est écrite dans un fichier de contraintes qui sera utilisé pour l'implémentation.



Figure 13: Algorigramme de CellsCombining.py

3.2.5 Automatisation

L'une des contraintes que nous nous sommes fixée est que l'outil développé doit être le plus user friendly possible. Cela passe par une automatisation de tous les processus du flot de l'outil développé.

3.2.5.1 Récupération du pinout du FPGA

L'une des contraintes fixées par le cahier des charges est d'adapter l'outil de redondance et d'optimisation de l'espace du circuit lors de l'implémentation en fonction d'un grand nombre de cibles FPGA. Or pour pouvoir automatiser l'implémentation, quel que soit la cible choisie par l'utilisateur, il faut que l'outil ait dans sa base de données les pinouts de tous les FPGA pris en compte par l'outil.

Pour éviter de stocker un grand nombre de fichiers, en fonction de la cible choisie, un script Python extrait à partir de la documentation Xilinx, le pinout associé au FPGA choisi pour l'implémentation. Le fonctionnement de ce script est détaillé ci-dessous. Évidemment cette étape de l'automatisation requiert une connexion internet pour pouvoir récupérer le pinout. Si aucune connexion internet n'est possible pour diverses raisons, au moment de l'utilisation de l'outil de redondance, il est toujours possible de renseigner le pinout du FPGA choisi en renseignant son emplacement.

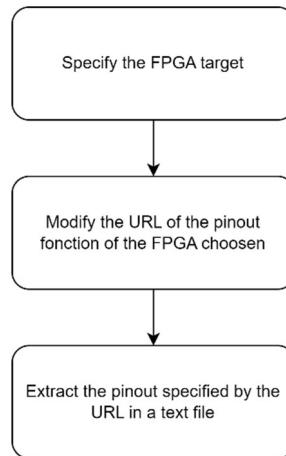


Figure 14: Algorigramme de internet.py

3.2.5.2 Générateur de fichiers de contraintes

Pour implémenter un circuit sur cible FPGA, il est nécessaire de renseigner à l'outil d'implémentation des contraintes de placement et de temps. Ces contraintes sont décrites dans des fichiers spécifiques propres à l'outil utilisé. Dans le cas de l'utilisation du logiciel Xilinx Vivado, les fichiers de contraintes sont décrits sous l'extension XDC (.xdc).

De nombreuses contraintes peuvent être spécifiées en fonction de l'application voulue, et du cahier des charges. A minima, il est nécessaire pour tout circuit de décrire des contraintes sur les types de PADs utilisés et leur placement, tant pour les signaux d'entrée/sorties. Il est nécessaire de construire le signal d'horloge utilisé dans les

circuits séquentiels, pour qu'il puisse correctement être distribué lors de l'implémentation de distribution du signal d'horloge dans le circuit pour les circuits séquentiels.

Le but du projet est de développer une solution permettant de rendre un circuit tolérant aux fautes sans modifier son comportement. Il est donc obligatoire d'adapter les fichiers de contraintes du circuit initial, au circuit généré par la technique du TMR.

À partir des fichiers de contraintes du circuit initial et du fichier source du top module, l'automatisation de la génération des fichiers de contrainte permet de construire les trois nouveaux fichiers de contraintes :

- *IO_properties.xdc*
- *pad_location.xdc*
- *timing.xdc*

Le script d'automatisation de la génération des fichiers de contrainte se décompose en plusieurs parties (Figure ci-dessous).

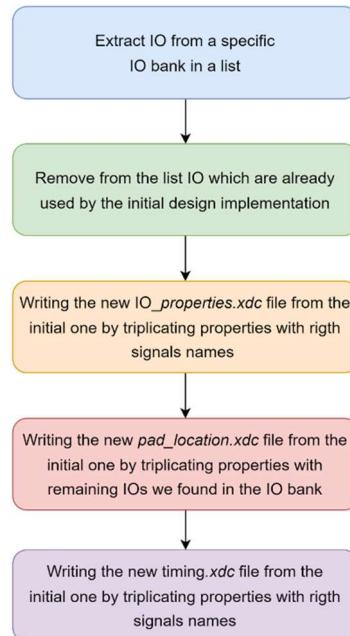


Figure 15: Algorigramme de constraintsGenerator.py

La première étape décrite dans ce script d'automatisation de la génération des fichiers de contraintes extrait, à partir du pinout de la cible FPGA choisie par l'utilisateur, toutes les banques de pad d'IO.

Remarque : Dans le fichier de pinout, les pins sont organisés en fonction de leur emplacement sur le FPGA. Un regroupement de pins situés dans une certaine région du FPGA est regroupé sous forme de banque de pins. Pour identifier tous les pins appartenant à une même banque d'IOs, un suffixe est ajouté au nom de chaque pin. La figure ci-dessous illustre des pins de deux banques de pins différentes.

Pin	Pin Name	Memory	Byte Group	Bank
AD25	IO_L23P_T3_12	3		12
AE25	IO_L23N_T3_12	3		12
AE22	IO_L24P_T3_12	3		12
AF22	IO_L24N_T3_12	3		12
Y20	IO_25_12	NA		12
N16	IO_0_13	NA		13
K25	IO_L1P_T0_13	0		13
K26	IO_L1N_T0_13	0		13

Figure 16 : Pinout

Le schéma de cette partie du script est illustré ci-dessous :

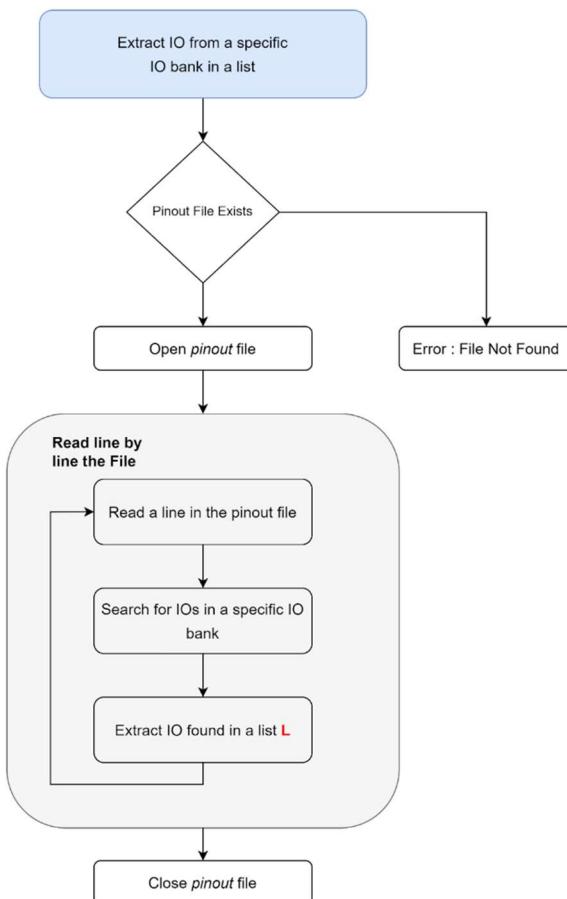


Figure 17: Algorigramme de lecture du pinout

Une fois que toutes les pins du pinout sont extraites dans une liste, il faut vérifier la disponibilité de chacune d'elles. Cela revient à vérifier pour chaque pin, si elle a déjà été mappée à un signal lors de l'implémentation du circuit initial.

Le fichier de contrainte *pad_location.xdc* permet justement de décrire le mapping, l'attribution des pins/pads aux différents signaux d'entrée/sortie du circuit.

La seconde partie du script, illustrée par le schéma ci-dessous, permet donc de vérifier quels pins sont disponibles à partir de la liste des pins extraites à partir du pinout et du fichier de contrainte *pad_location.xdc*.

Tous les pins déjà mappés à des signaux du circuit sont retirés de la liste pour qu'au moment de l'écriture du nouveau fichier de contrainte *pad_location.xdc*, les nouveaux signaux d'entrées/sorties engendrés par l'algorithme de TMR, soient assignés à des pins vierges de tout signaux.

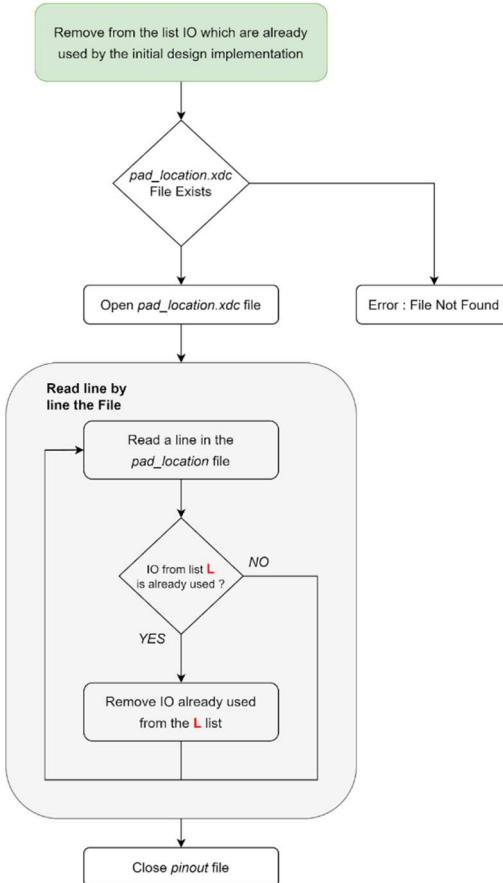


Figure 18: Algorigramme du tri des PADs utilisés

Les trois dernières étapes du script de génération des fichiers de contrainte reviennent à l'écriture de chaque fichier :

- *IO_properties.xdc*
- *pad_location.xdc*
- *timing.xdc*

Le fichier de contrainte *IO_properties.xdc* est un fichier qui permet de spécifier pour chaque signaux d'entrée/sortie le type d'IO ou encore son niveau de tension. Pour générer le nouveau fichier de contrainte *IO_properties.xdc* correspondant au nouveau circuit, il suffit de parcourir le fichier *IO_properties.xdc* initial et de tripler les lignes qui définissent les IO des signaux en ajoutant un suffixe au nom du signal, de la même façon que dans le script de redondance du circuit. La figure ci-dessous est une comparaison entre un morceau du fichier d'entrée et celui de sortie.

```

set_property CFGBVS VCCO [current_design]
set_property CONFIG VOLTAGE 3.3 [current_design]
set_property IOSTANDARD LVCMOS33 [get_ports {inClock}]
set_property IOSTANDARD LVCMOS33 [get_ports {inReset}]

set_property CFGBVS VCCO [current_design]
set_property CONFIG VOLTAGE 3.3 [current_design]
set_property IOSTANDARD LVCMOS33 [get_ports {inClock}]
set_property IOSTANDARD LVCMOS33 [get_ports {inClock_NMR2}]
set_property IOSTANDARD LVCMOS33 [get_ports {inClock_NMR3}]
set_property IOSTANDARD LVCMOS33 [get_ports {inReset}]
set_property IOSTANDARD LVCMOS33 [get_ports {inReset_NMR2}]
set_property IOSTANDARD LVCMOS33 [get_ports {inReset_NMR3}]

```

Figure 19 : Redondance des contraintes de propriétés

Le fonctionnement du script de génération du fichier IO_properties.xdc est décrit comme suit.

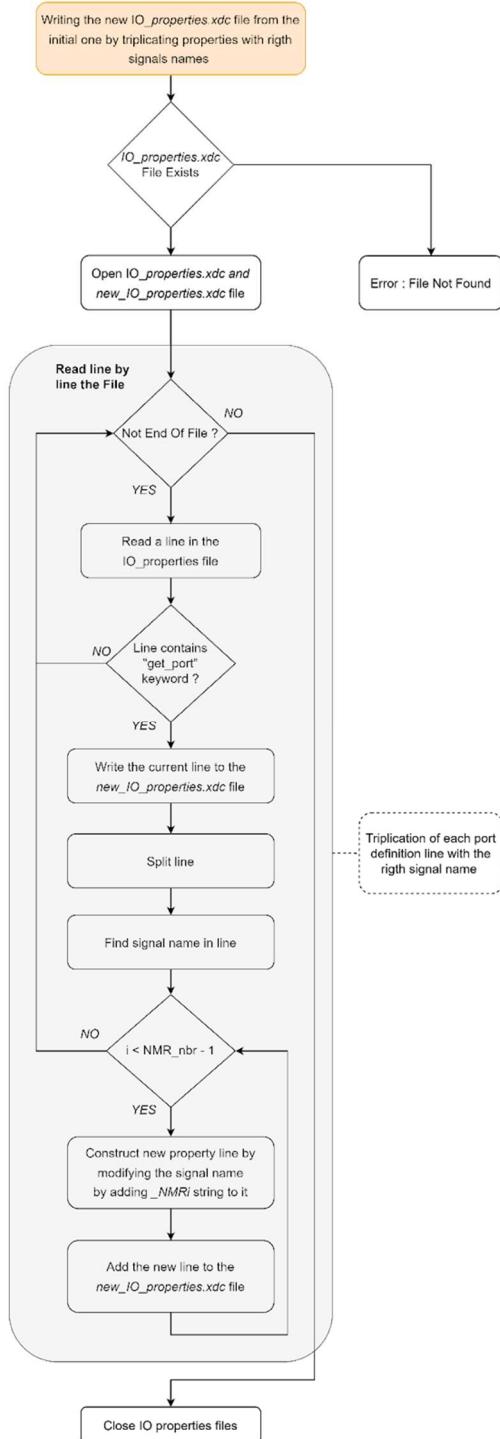


Figure 20 : Algorigramme de redondance des propriétés

Le fichier de contrainte *pad_location.xdc* est le fichier qui permet d'assigner les pins disponibles sur le FPGA, aux signaux d'entrée/sortie du circuit. Le nouveau fichier de contrainte *pad_location.xdc* est construit à partir du fichier *pad_location.xdc* initial. Le but est d'assigner un pin libre par signal, en parcourant la liste des pins libres construite en amont. La liste des signaux à mapper correspond à la concentration des signaux d'entrées/sorties du circuit initial et de leurs signaux triplés.

La figure ci-dessous est une comparaison entre un morceau du fichier d'entrée et celui de sortie.

```
set_property PACKAGE_PIN D18 [get_ports {inReset}]\nset_property PACKAGE_PIN H17 [get_ports {inData1}]\n\nset_property PACKAGE_PIN 018 [get_ports {inReset}]\nset_property PACKAGE_PIN J19 [get_ports {inReset_NMR2}]\nset_property PACKAGE_PIN B19 [get_ports {inReset_NMR3}]\nset_property PACKAGE_PIN H17 [get_ports {inData1}]\nset_property PACKAGE_PIN H18 [get_ports {inData1_NMR2}]\nset_property PACKAGE_PIN C19 [get_ports {inData1_NMR3}]
```

Figure 21 : Redondance des contraintes de placement

Pour parvenir à ce résultat, ci-dessous le fonctionnement du script de génération du fichier *pad_location.xdc*.

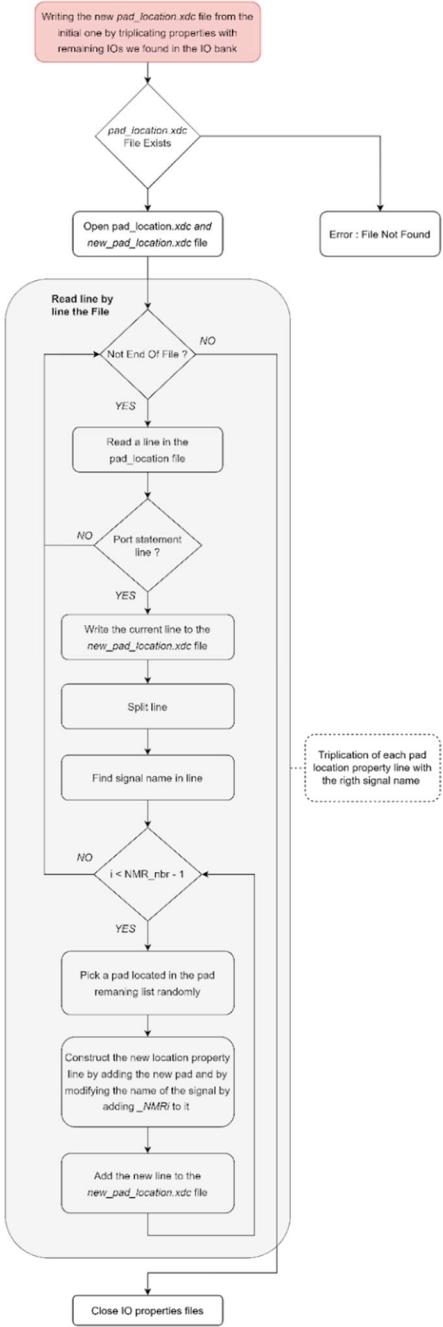


Figure 22 : Algorigramme de redondance du placement

Le fichier de contrainte `timing.xdc` correspond est le fichier qui permet notamment de générer le signal d'horloge du circuit. Il est aussi possible de décrire des contraintes sur les chemins critiques. Lors de la génération du circuit tolérant aux fautes, le signal d'horloge est aussi triplé. En effet le signal d'horloge est un des signaux les plus critiques du circuit car il est distribué dans tout le FPGA. Si un bit flip intervient sur le signal d'horloge c'est le fonctionnement de tout le système qui peut être affecté, d'où la nécessité de tripler le signal d'horloge.

La figure ci-dessous est une comparaison entre le fichier de contrainte de timing d'entrée et celui de sortie.

```

create_clock -period 20.833 -name ckin [get_ports inClock]
set_false_path -from [get_ports inReset]

create_clock -period 20.833 -name ckin [get_ports inClock]
create_clock -period 20.833 -name ckin_NMR2 [get_ports inClock_NMR2]
create_clock -period 20.833 -name ckin_NMR3 [get_ports inClock_NMR3]
set_false_path -from [get_ports inReset]
set_false_path -from [get_ports inReset_NMR2]
set_false_path -from [get_ports inReset_NMR3]

```

Figure 23 : Redondance des contraintes de timing

Pour parvenir à ce résultat, ci-dessous le fonctionnement du script de génération du fichier *timing.xdc*.

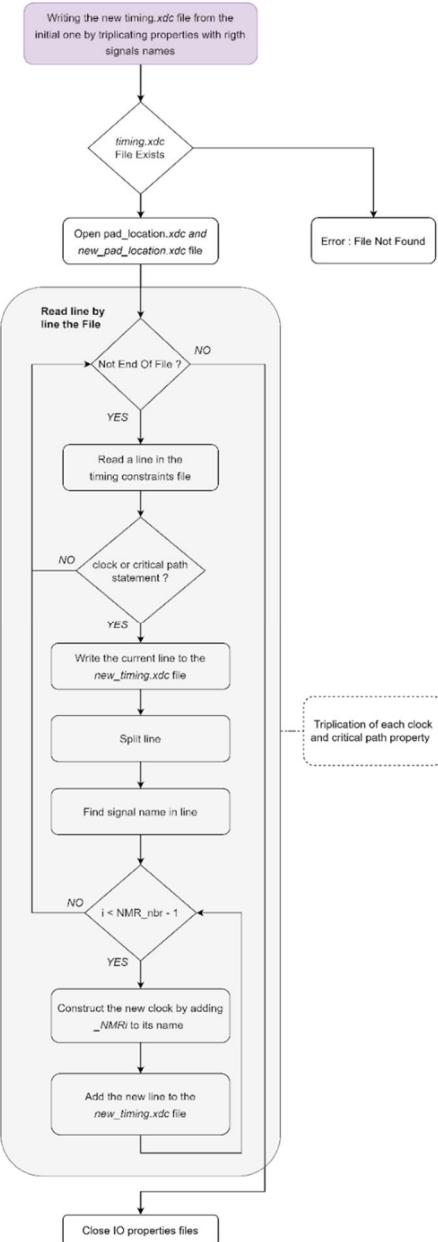


Figure 24 : Algorigramme de redondance du timing

3.2.5.3 Générateur de banc de test

L'outil développé pour rendre le circuit tolérant aux fautes ne doit en aucun cas affecter le comportement fonctionnel du circuit. Pour tester le circuit en post TMR, il faut adapter le banc de test initial à ce nouveau circuit. En effet la redondance du circuit a

engendré la redondance de tous les signaux d'entrée/sortie. Par conséquent, il faut pouvoir contrôler ces nouveaux signaux de la même manière que les signaux d'entrée/sortie initiaux du circuit.

De plus, pour vérifier que le circuit est tolérant aux fautes, il est nécessaire d'adapter le banc de test initial au circuit généré par l'algorithme pour simuler des injections de fautes.

Le script de génération du banc de test prend en entrée le banc de test initial ainsi que le fichier source du top module du circuit. Le banc de test généré en sortie est donc propre au circuit généré par la technique du TMR.

Le principe de fonctionnement du script python est le même que pour les scripts de génération des fichiers de contrainte présentés précédemment. Le banc de test initial est lu ligne par ligne. On distingue trois parties dans les bancs de test RTL, linstanciation du DUT, la déclaration des signaux utilisés ainsi que la description des stimuli de test. Seule la génération de bancs de tests VHDL est prise en compte dans le script.

Le script est donc décomposé en trois parties bien distinctes. Lorsque la ligne lue correspond à la déclaration d'un "signal", elle est triplée et un suffixe est ajouté au nom de ces nouveaux signaux de sorte à les différencier. La figure ci-dessous correspond au résultat de la redondance des signaux du banc de test.

```

architecture Behavioral of tb_top is
begin
  constant ClockPeriod : time := 20 ns;
  signal inClock : STD_LOGIC := '0';
  signal inReset : STD_LOGIC := '0';
  signal inData1 : STD_LOGIC := '0';
  signal inData2 : STD_LOGIC := '0';
  signal S1 : STD_LOGIC;
  signal S2 : STD_LOGIC;
begin
  constant ClockPeriod : time := 20 ns;
  signal inClock : STD_LOGIC := '0';
  signal inClock_NMR2 : STD_LOGIC := '0';
  signal inClock_NMR3 : STD_LOGIC := '0';
  signal inReset : STD_LOGIC := '0';
  signal inReset_NMR2 : STD_LOGIC := '0';
  signal inReset_NMR3 : STD_LOGIC := '0';
  signal inData1 : STD_LOGIC := '0';
  signal inData1_NMR2 : STD_LOGIC := '0';
  signal inData1_NMR3 : STD_LOGIC := '0';
  signal S2 : STD_LOGIC;
  signal S2_NMR2 : STD_LOGIC;
  signal S2_NMR3 : STD_LOGIC;
begin

```

Figure 25 : Redondance des signaux du banc de test

Si la ligne lue contient le mot-clé 'map', on doit instancier la nouvelle version du DUT. Au préalable, lors de la post-synthèse, un fichier Verilog a été généré : il décrit les IOs du DUT. Parser ce fichier permet d'accéder à toutes les IOs du nouveau DUT. Il est alors possible d'instancier le nouveau DUT dans le testbench. La figure ci-dessous illustre le résultat de linstanciation du nouveau DUT, avec toutes ses IO triplés, dans le banc de test.

```

-----  

-- TOP module instance  

-----  

TOP : entity work.topM  

Port Map(  

    inClock => inClock ,  

    inReset => inReset ,  

    inData1 => inData1 ,  

    S2      => S2  

);
-----  

-----  

-- TOP module instance  

-----  

TOP : entity work.topM  

Port Map(  

    inClock      => inClock      ,  

    inClock_NMR2 => inClock_NMR2 ,  

    inClock_NMR3 => inClock_NMR3 ,  

    inReset      => inReset      ,  

    inReset_NMR2 => inReset_NMR2 ,  

    inReset_NMR3 => inReset_NMR3 ,  

    inData1      => inData1      ,  

    inData1_NMR2 => inData1_NMR2 ,  

    inData1_NMR3 => inData1_NMR3 ,  

    S2          => S2          ,  

    S2_NMR2     => S2_NMR2     ,  

    S2_NMR3     => S2_NMR3
);
-----  


```

Figure 26 : Ecriture du DUT après redondance

La dernière partie du banc de test correspond à la description du stimuli. Cette partie du fichier débute dès la lecture du mot clé ‘process’. Le but ici est de tripler tous les stimuli comme sur la figure ci-dessous.

```

inDataProcess : process
begin
    inData1 <= '0';
    wait for 6*ClockPeriod;
    inData1 <= '0';
    wait for 4*ClockPeriod;
    inData1 <= '1';
    wait for 4*ClockPeriod;
    inData1 <= '0';
    wait for 4*ClockPeriod;
    inData1 <= '1';
    wait for 4*ClockPeriod;
    inData1 <= '0';
    wait;
end process;

inDataProcess : process
begin
    inData1      <= '0';
    inData1_NMR2 <= '0';
    inData1_NMR3 <= '0';
    wait for 6*ClockPeriod;
    inData1      <= '0';
    inData1_NMR2 <= '0';
    inData1_NMR3 <= '0';
    wait for 4*ClockPeriod;
    inData1      <= '1';
    inData1_NMR2 <= '1';
    inData1_NMR3 <= '1';
    wait for 4*ClockPeriod;
    inData1      <= '0';
    inData1_NMR2 <= '0';
    inData1_NMR3 <= '0';
    wait for 4*ClockPeriod;
    inData1      <= '1';
    inData1_NMR2 <= '1';
    inData1_NMR3 <= '1';
    wait for 4*ClockPeriod;
    inData1      <= '0';
    inData1_NMR2 <= '0';
    inData1_NMR3 <= '0';
    wait;
end process;

```

Figure 27 : Redondance des stimuli du banc de test

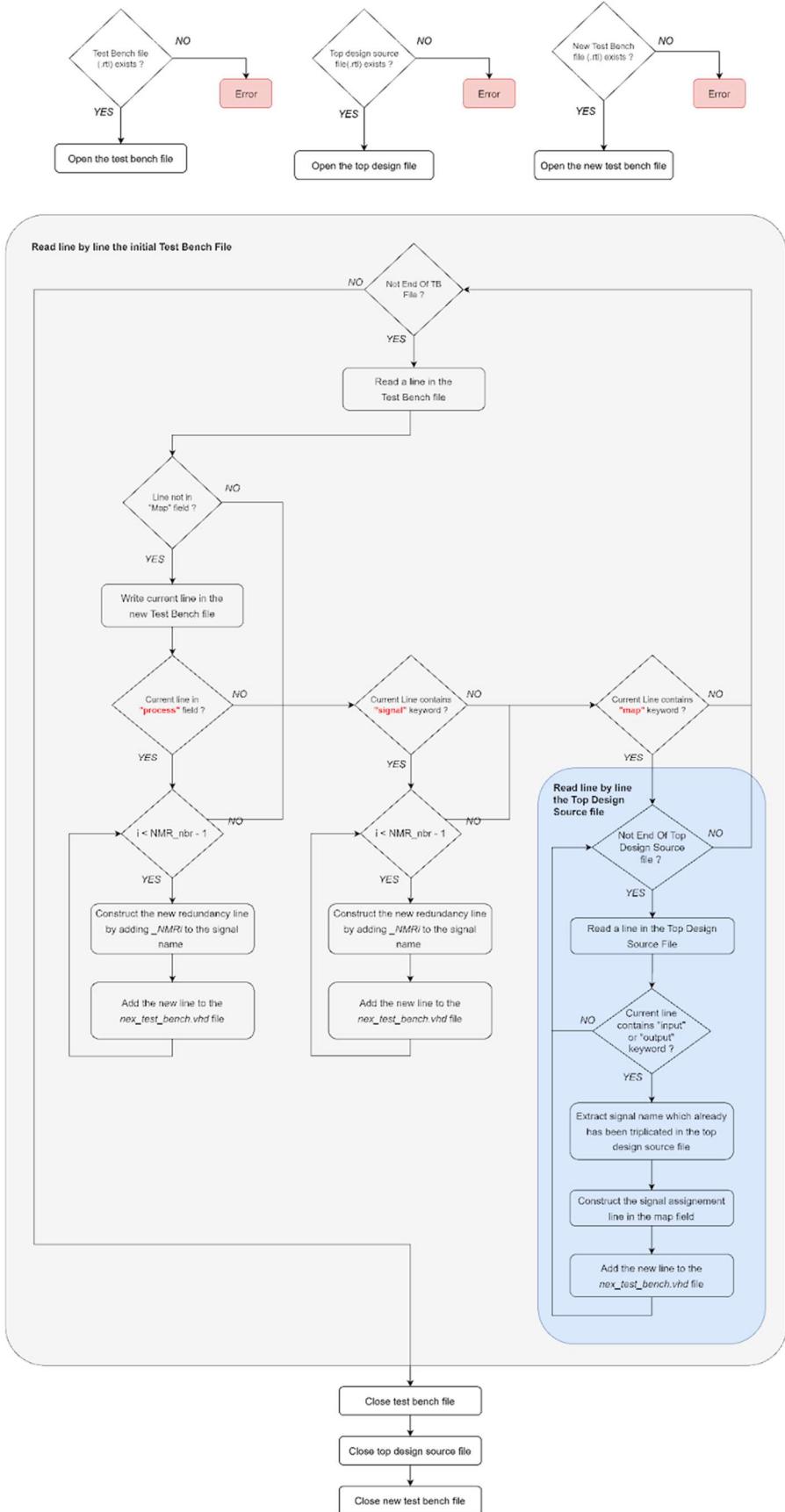


Figure 28: Algorigramme de la redondance du banc de test

3.2.5.4 Scripts TCL

L'automatisation de la redondance d'un circuit nécessite l'utilisation du logiciel Vivado. Le logiciel peut être lancé sans interface graphique et exécuter des commandes reçues à travers un script TCL.

3.2.5.4.1 Ouverture de projet et génération d'une netlist

Un premier script permet d'ouvrir un projet existant. À partir de ce projet, le module en haut de la hiérarchie est redéfinie et l'ordre de compilation des fichiers est mis à jour. La synthèse de ce projet est ensuite effectuée, puis la netlist correspondant au résultat de cette synthèse est écrite dans un fichier .EDF (format EDIF). Le projet est finalement fermé.

3.2.5.4.2 Création d'un projet à partir d'une netlist et récupération des informations principales du circuit

Un second script permet de créer un nouveau projet à partir d'une netlist. Ce script récupère aussi la cible FPGA visée. Après configuration des fichiers du projet, le design correspondant au résultat de synthèse est ouvert, et les propriétés concernant les primitives utilisées dans ce design sont renseignées dans un fichier dédié. Le script en profite pour extraire dans un fichier Verilog les IOs de ce design, avant de fermer le projet.

3.2.5.4.3 Contraintes du circuit, implémentation, simulations post-implémentation

Un troisième script permet d'ouvrir le projet post-synthèse précédemment créé. Le script permet d'ajouter les fichiers de contraintes et de simulation. Le script lance ensuite une implémentation, puis exécute une simulation fonctionnelle post-implémentation. À la fin de l'exécution de la simulation, l'interface graphique utilisateur se lance pour vérifier que la fonctionnalité du circuit initial n'a pas été modifiée.

3.2.5.5 Script de commande

Un script de commande permet de cadencer tous les scripts décrits auparavant. Les scripts sont exécutés dans l'ordre suivant (Figure 29) :

- Premier script TCL
- internet.py
- constraintsGenerator.py
- NMR_TOP.py
- Second script TCL
- CellsCombining.py
- tbenchGenerator.py
- Troisième script TCL

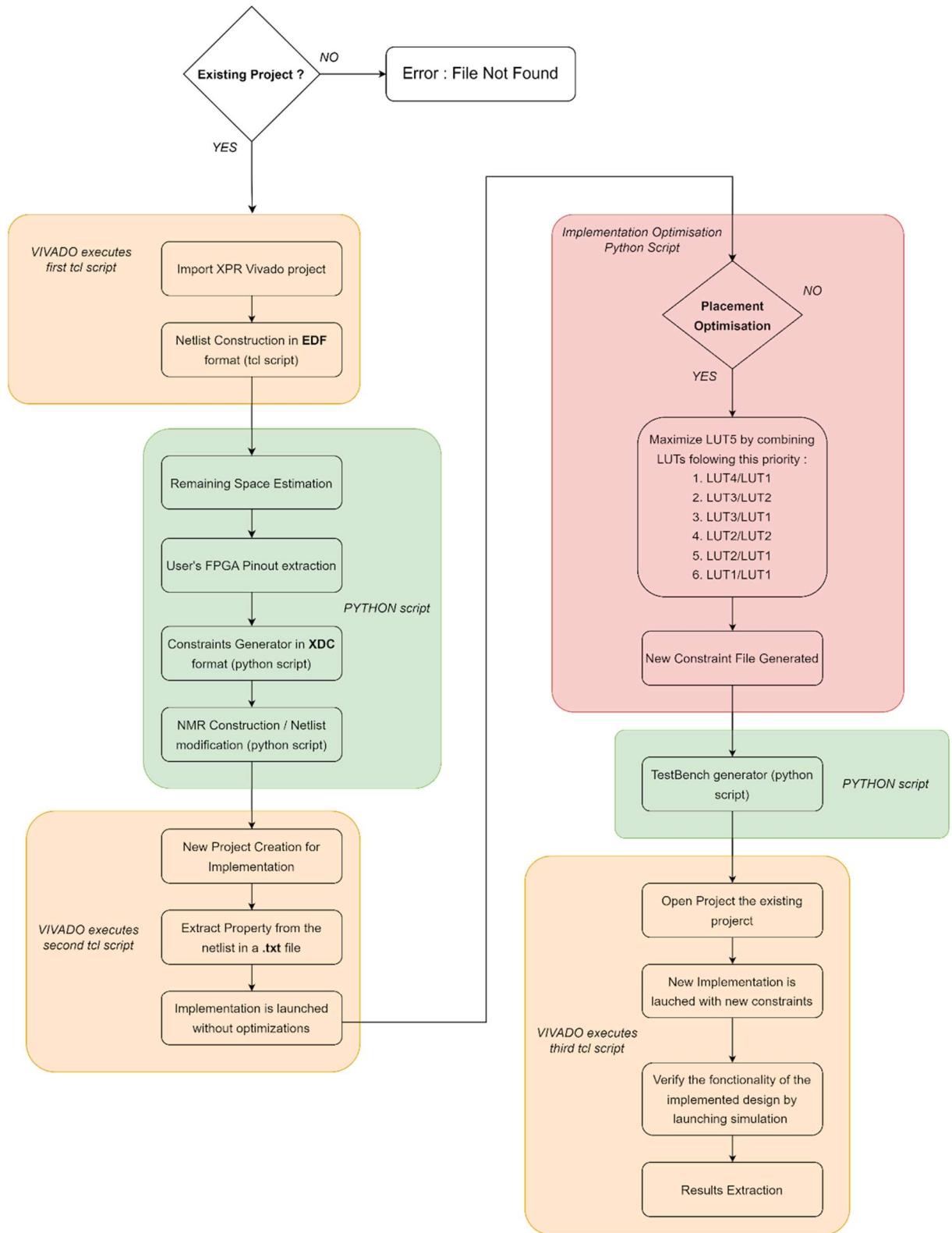


Figure 29 : Algorigramme d'automatisation des scripts

3.2.6 Interface Graphique

L'automatisation de l'algorithme de redondance et d'optimisation de l'implémentation est managée par un script de commande qui exécute tour à tour les différents scripts python ou TCL. Le problème d'une automatisation avec ce type de gestion n'est pas ergonomique et nécessite l'intervention de l'utilisateur dans plusieurs des scripts exécutés pour renseigner l'emplacement d'un ou plusieurs fichiers.

Par exemple, le script d'automatisation de la génération des fichiers de contraintes nécessite en entrée l'emplacement de tous les fichiers de contraintes initiaux du projet ainsi que du fichier RTL source du top module du circuit.

L'ajout d'une interface graphique au processus d'automatisation du système permet à l'utilisateur de renseigner toutes les informations nécessaires avant exécution du système sans passer par les scripts sources. Cela permet aussi à l'utilisateur d'avoir accès aux principaux résultats de l'optimisation d'espace à l'implémentation par exemple.

L'interface graphique a été développée en python via la librairie TkInter. Elle correspond au fichier mère de l'automatisation. En effet c'est à partir de ce fichier et uniquement une fois que tous les champs obligatoires ont été renseignés par l'utilisateur, que le fichier de commande décrit en amont est exécuté.

Pour exécuter le système, il faut renseigner l'emplacement des fichiers suivants :

- Le fichier du projet existant sous format .xpr (format Xilinx Vivado)
- Les fichiers de contraintes
- Le banc de test du circuit fourni en entrée

La figure ci-dessous résume les fichiers à renseigner par l'utilisateur avant exécution du système.

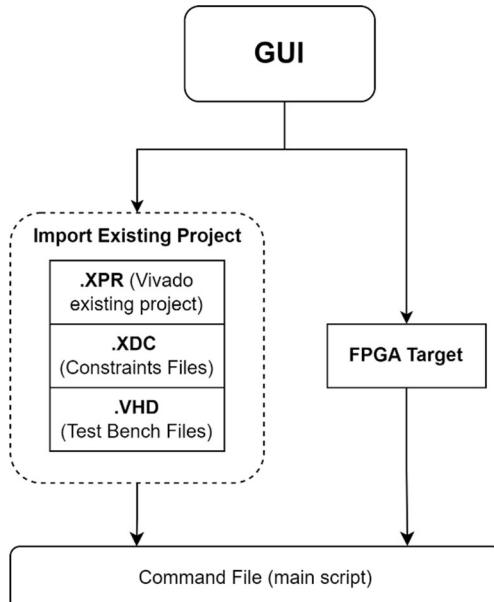


Figure 30 : Algorigramme de l'interface utilisateur

De plus, il faut obligatoirement que l'utilisateur renseigne la cible FPGA sur laquelle il travaille car comme décrit précédemment, le type de FPGA est une information essentielle lors de la génération des fichiers de contraintes.

Il est également possible d'implémenter le circuit dans le FPGA sans optimisation de placement. Cela peut être utile pour comparer l'espace utilisé, la consommation et les chemins critiques du circuit implémenter sans optimisation et celui avec.

L'interface graphique est illustrée ci-dessous.

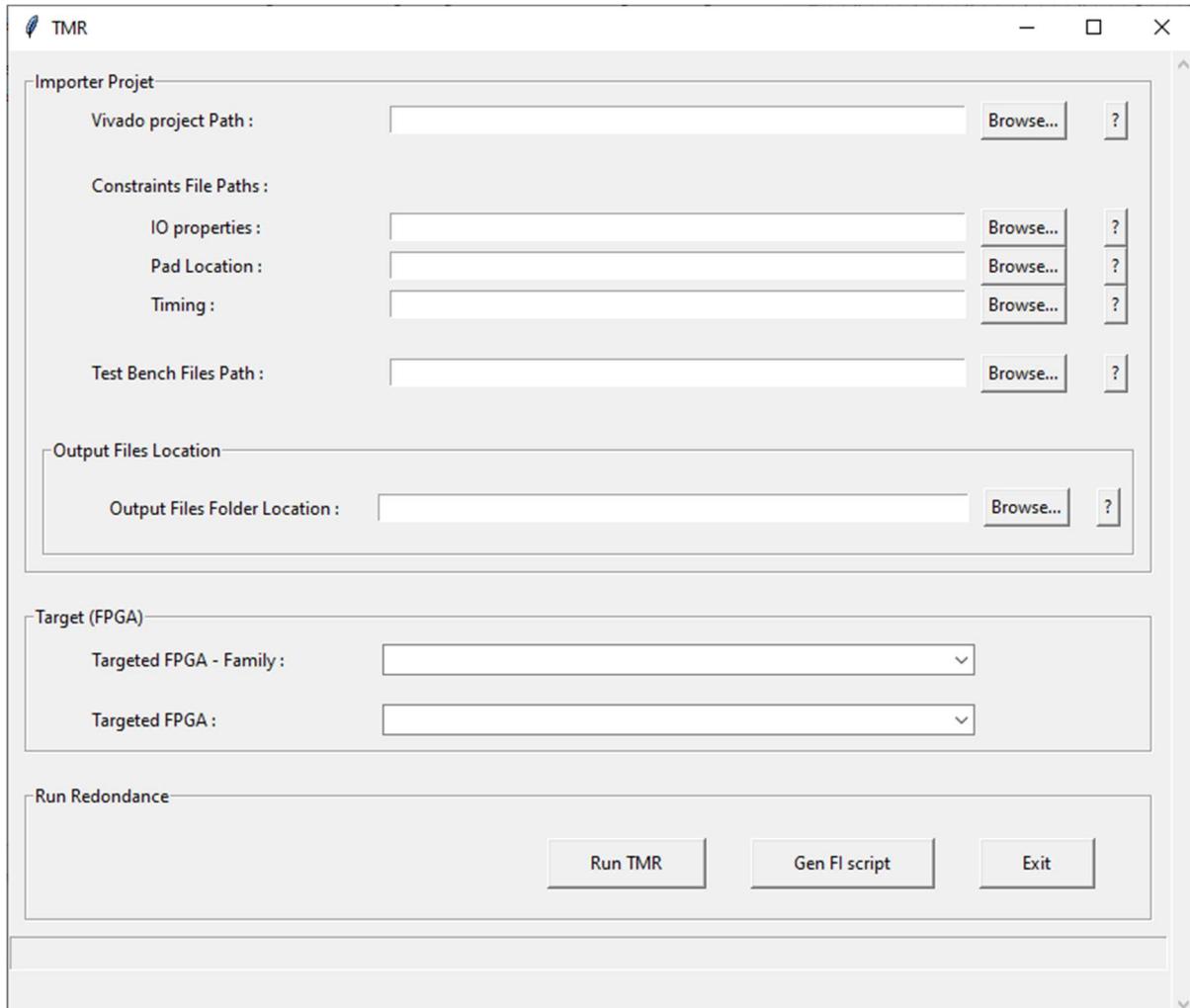


Figure 31 : Interface utilisateur

Pour vérifier que le circuit généré par l'algorithme de redondance est bien tolérant aux perturbations, il faut observer son comportement en simulation, avant de passer sur la carte. Pour cela, Vivado permet de forcer des signaux pendant les simulations. Le but est de forcer aléatoirement un signal du circuit à une valeur constante. Si le circuit est tolérant aux fautes, il ne devrait pas prendre en compte la perturbation, et ainsi avoir le comportement attendu.

Toujours dans une optique d'automatisation, le script TCL permettant de lancer la simulation avec une faute injectée est généré par l'appui sur le bouton Gen FI script de l'interface graphique. A l'appui, un signal du circuit est choisi aléatoirement pour être forcé aléatoirement à 0 ou 1.

4. Résultats

À la fin de l'exécution du batch lancé par l'interface graphique, Vivado se lance et affiche le résultat d'une simulation fonctionnelle post-implémentation. Dans un premier temps, au regard de ce résultat de simulation, il est nécessaire que le fonctionnement du circuit soit identique au fonctionnement du circuit initial.

La Figure 32 correspond au résultat de simulation fonctionnelle d'un circuit lambda avant synthèse tandis que la Figure 33 correspond au résultat de simulation fonctionnelle post-implémentation du même circuit après redondance.

Le fonctionnement des trois circuits après redondance est identique au fonctionnement du circuit initial. La réalisation de la redondance est donc validée.

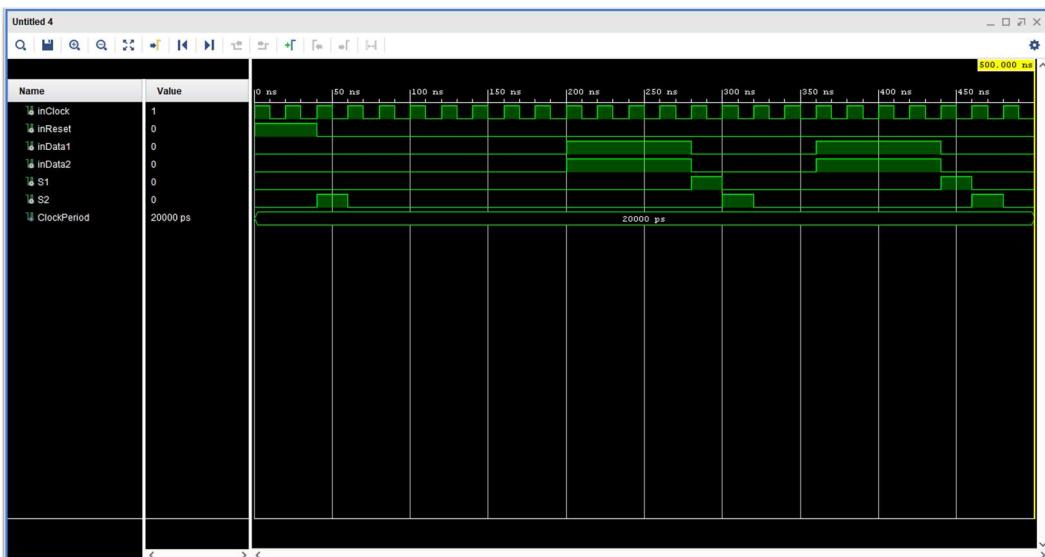


Figure 32 : Simulation fonctionnelle

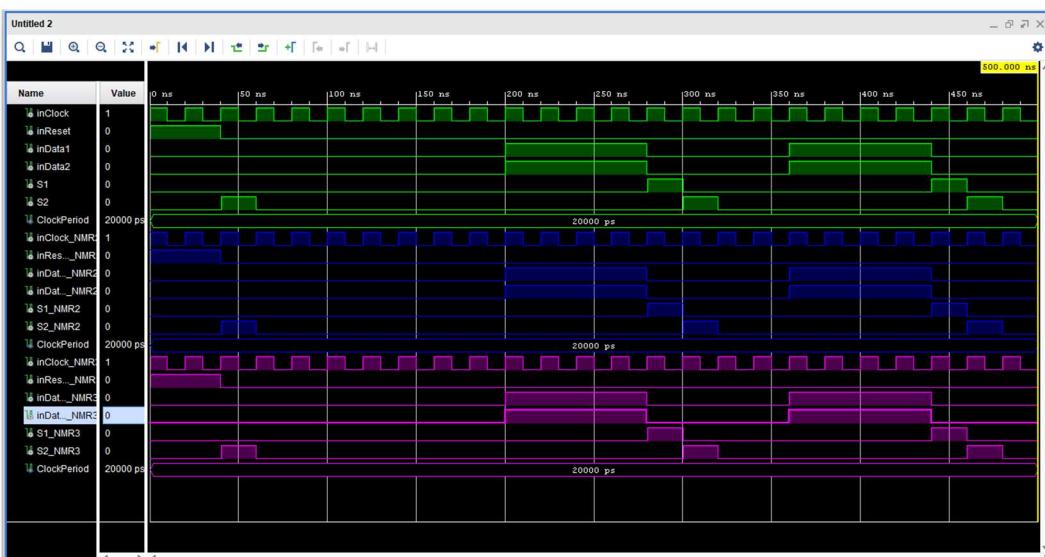


Figure 33 : Simulation fonctionnelle post-implémentation

Dans un second temps, il peut être intéressant de vérifier avec une simulation de timing post-implémentation que le circuit fonctionne toujours. La Figure 34 correspond au résultat de simulation de timing post-implémentation.

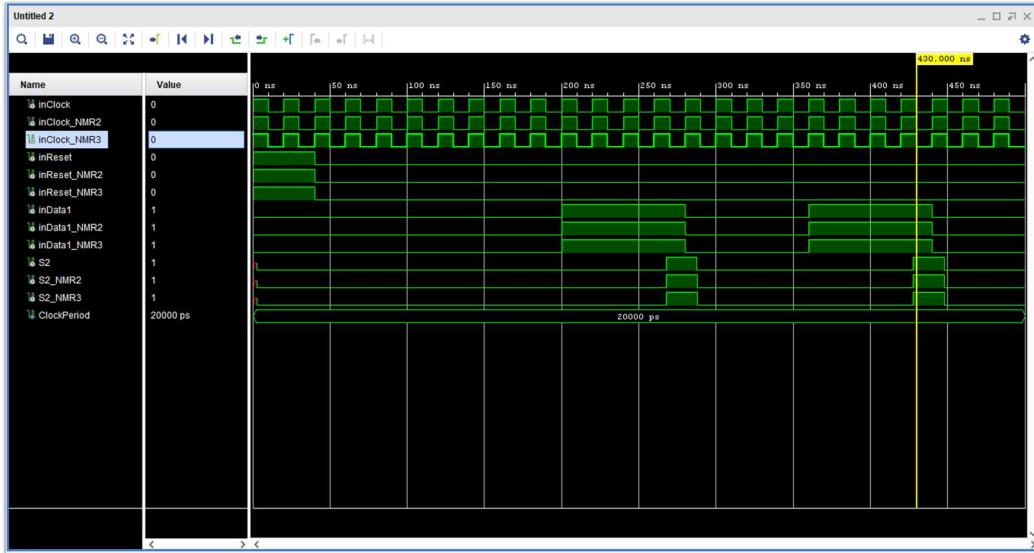


Figure 34 : Simulation de timing post-implémentation

Le timing est respecté : pour une horloge de fréquence 48 MHz ($T = 20,833 \text{ ns}$), le chemin critique est inférieur à une période d'horloge (17,544).

Cette information se trouve aussi dans le rapport post-implémentation sous l'appellation WNS (Worst Negative Slack).

Dans troisième temps, l'optimisation de l'espace occupé dans le FPGA est un facteur clé. On peut donc évaluer l'espace occupé par le circuit dans différentes configurations (Figure 35):

- sans TMR, sans optimisation de placement
- avec TMR, sans optimisation de placement
- avec TMR, avec optimisation de placement

La configuration sans TMR, avec optimisation de placement pourrait aussi être étudiée.

Design Runs														
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
synth_1	constrs_1	synth_design Complete!								4	8	0.00	0	0
impl_1	constrs_1	route_design Complete!	20.015	0.000	0.146	0.0...	0.000	0.114	0	4	8	0.00	0	0

Design Runs														
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
impl_1 (active)	constrs_1	Implementation Out-of-date	17.544	0.000	0.118	0.000	0.000	0.114	0	12	24	0.00	0	0
impl_2	constrs_1	route_design Complete!	18.062	0.000	0.183	0.000	0.000	0.114	0	21	24	0.00	0	0

Figure 35 : Comparaison des différents résultats d'implémentation

Le circuit est relativement petit, il est donc impossible de conclure sur la réduction de l'espace. Cependant, entre les deux designs après redondance, l'optimisation réduit presque de moitié le nombre de LUTs.

Pour vérifier fonctionnellement que le circuit est tolérant aux fautes, nous lui avons injecté aléatoirement une faute sur un de ces signaux et vérifier que la sortie du circuit n'était pas perturbée par cette perturbation. La figure ci-dessous correspond à la simulation sans injection de faute d'un circuit contenant quatre one-shots interconnectés.

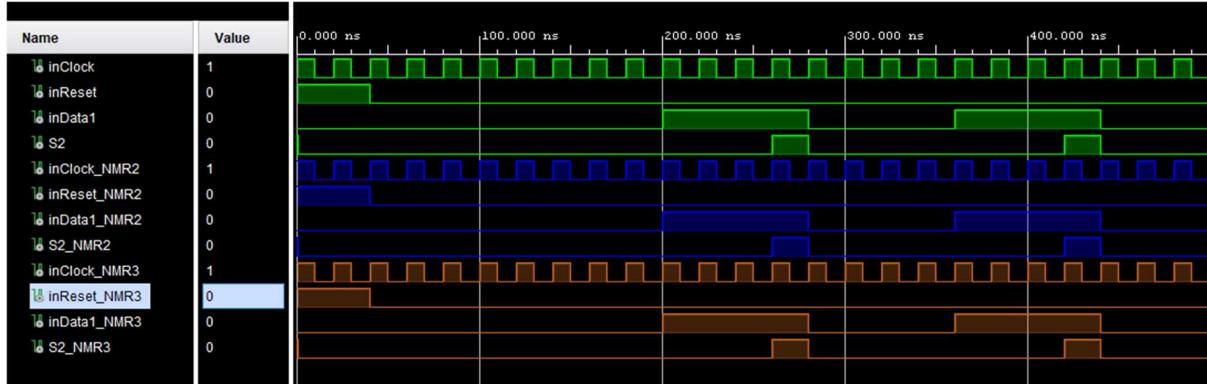


Figure 36 : Simulation fonctionnelle sans injection de faute

Sur la figure ci-dessous le signal InData1_NMR3 a été forcé à 1.

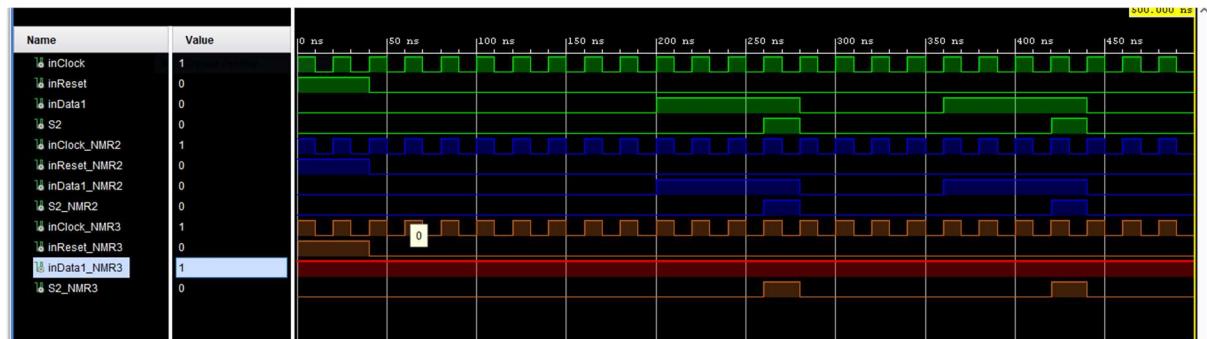


Figure 37 : Simulation fonctionnelle avec injection de faute

Nous pouvons voir que les signaux de sortie ne sont pas affectés par la perturbation.

5. Plan de validation

Le plan de validation permet de démontrer que notre produit répond aux contraintes du cahier des charges. La validation va permettre de vérifier toutes les étapes du développement.

Test n°1	Date :	
TCL 1	Résultats attendus	Résultat
<ul style="list-style-type: none"> Ouvrir le projet Vivado spécifié en arg Exécuter une synthèse Générer la netlist dans un fichier spécifié en arg 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Netlist disponible dans le fichier spécifié 	OK

Test n°2	Date :	
internet.py	Résultats attendus	Résultat
<ul style="list-style-type: none"> Récupérer le pinout d'un FPGA spécifié en arg Stocker ce pinout dans un fichier spécifié 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Pinout disponible dans le fichier spécifié 	OK

Test n°3	Date :	
constraintsGenerator.py	Résultats attendus	Résultat
<ul style="list-style-type: none"> Appliquer la redondance sur les fichiers initiaux Écrire la redondance dans les fichiers spécifiés en arg 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Nouveaux fichiers de contraintes contiennent la redondance 	OK

Test n°4	Date :	
NMR_TOP.py	Résultats attendus	Résultat
<ul style="list-style-type: none"> Appliquer la redondance sur la netlist initiale Ecrire la nouvelle netlist 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Nouvelle netlist disponible dans le fichier spécifié 	OK

Test n°5	Date :	
TCL 2	Résultats attendus	Résultat
<ul style="list-style-type: none"> Créer un projet post-synthèse Spécifier le FPGA cible Extraire les cellules primitives Générer le fichier Verilog contenant la description des IOs 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Projet créé Cellules primitives extraites Fichier Verilog disponible 	OK

Test n°6	Date :	
CellsCombining.py	Résultats attendus	Résultat
<ul style="list-style-type: none"> Ouvrir le fichier contenant les cellules primitives Créer un fichier de contraintes pour optimiser l'espace dans le FPGA 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Fichier de contraintes disponible 	OK

Test n°7	Date :	
tbenchGenerator.py	Résultats attendus	Résultat
<ul style="list-style-type: none"> Ouvrir le tbench initial Ouvrir le fichier Verilog contenant les infos sur les IOs Générer un nouveau tbench pour le circuit TMR 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Nouveau tbench disponible pour le circuit TMR 	OK

Test n°8	Date :	
TCL 3	Résultats attendus	Résultat
<ul style="list-style-type: none"> Ouvrir le projet post-synthèse Ajouter les fichiers de contraintes Ajouter le tbench Faire l'implémentation Lancer la simulation fonctionnelle post-implémentation 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Implémentation réussie Simulation se lance sans encombre 	OK

Test n°9	Date :	
BATCH	Résultats attendus	Résultat
<ul style="list-style-type: none"> Automatiser tous les scripts 	<ul style="list-style-type: none"> Le script s'exécute sans erreur 	OK

Test n°10	Date :	
TMR_GUI.py	Résultats attendus	Résultat
<ul style="list-style-type: none"> Afficher une interface graphique Lancer le script d'automatisation 	<ul style="list-style-type: none"> Le script s'exécute sans erreur Automatisation lancée 	OK

6. Axes d'amélioration

Les points que nous aurions souhaités améliorer/ajouter sont :

- Faire la redondance seulement sur les bascules
- GUI qui fonctionne sur Linux
- Pouvoir créer un projet depuis l'interface graphique et non pas seulement l'importer.

7. Gestion de projet

7.1 Méthodologie de travail

Dès le début du projet, nous avons créé un répertoire de projet sous GitLab. Le principal objectif de cette mise en place est de versionner le travail de chacun à chaque évolution majeure du code. L'intérêt de mettre en place un répertoire partagé en ligne permet de toujours travailler sur des fichiers à jour, comportant toutes les modifications faites ultérieurement par un autre membre du groupe.

Pour travailler sur des versions différentes du code simultanément, nous avons créé plusieurs branches au projet. Dès lors qu'une étape majeure était franchie, la branche principale est actualisée.

L'autre intérêt d'utiliser un répertoire partagé et de toujours être au courant de l'avancée des autres membres du groupe, utile pour le debug par exemple.

Concernant l'organisation au niveau matériel de notre environnement de travail, dans un premier temps nous travaillions depuis le CIME en utilisant les licences Vivado du laboratoire jusqu'à ce que le CIME soit contraint de fermer l'accès à ses machines pour cause de hacking des serveurs de l'université.

Nous avons donc dû nous adapter en étant contraints de travailler depuis chez nous. Par chance nous avons pu installer le même environnement de travail que celui que nous utilisions au CIME, à savoir le logiciel Xilinx Vivado ainsi qu'un IDLE Python. Cela a pris un peu de temps à remettre en place l'environnement de travail mais nous avons ainsi finalement pu continuer notre projet dans de bonnes conditions.

7.2 Planning - Répartition des tâches

7.2.1 Planning prévisionnel

Dans le planning prévisionnel, nous avions prévu plusieurs grandes parties tout au long du projet. La première étant la partie d'analyse, où nous nous mettons d'accord sur le cahier des charges, les outils que nous allions utilisés et la prise en main de Vivado.

Vient ensuite la partie réalisation de notre projet avec la mise en œuvre des algorithmes de votes, de redondance, à travers la création des lexer/parser de Netlist et des fichiers d'information sur les slices du FPGA.

Enfin la partie validation afin de valider notre système et la mise en forme des résultats. Si le temps le permettait, nous avions prévu de faire des études sur d'autres méthodes de protection contre les perturbations et effectuer des comparaisons entre eux.

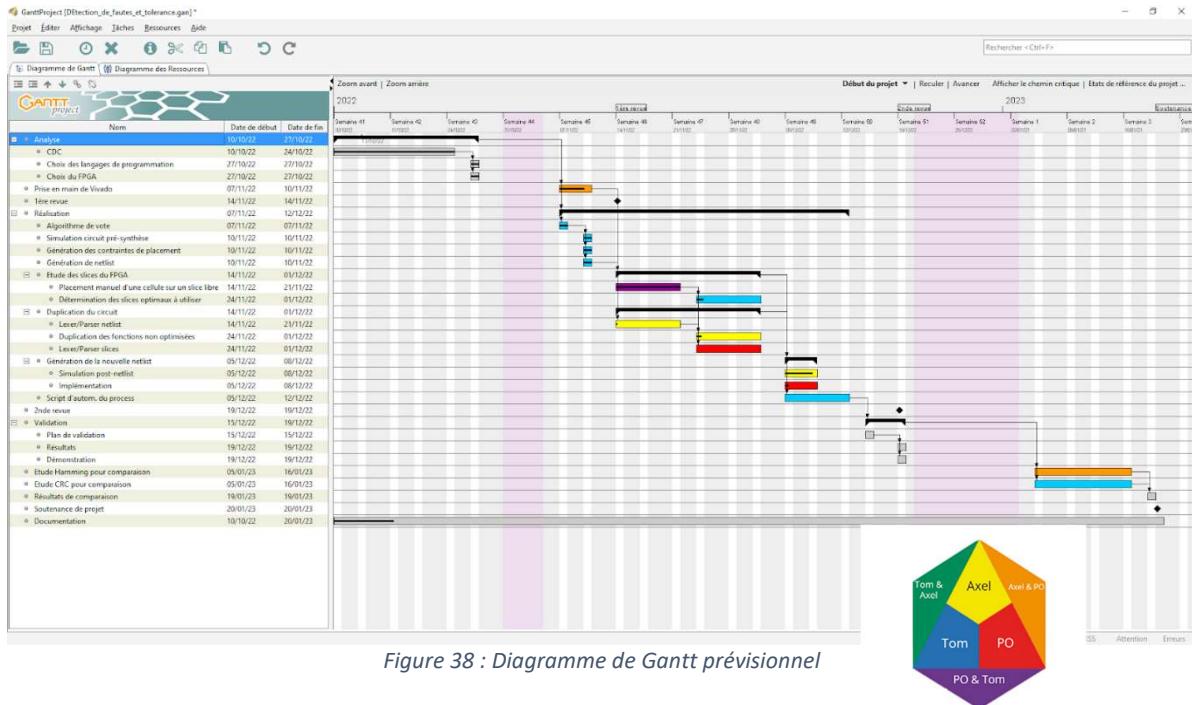


Figure 38 : Diagramme de Gantt prévisionnel

7.2.2 Planning réel

Les différences notables avec le planning prévisionnel et le réel est que nous avons ajouté une interface graphique à notre projet. Nous avons dépassé le temps estimé pour réaliser les scripts d'automatisation du circuit, ce qui nous a amené à dépasser les dates estimées. Cependant cette perte de temps n'aura pas affecté la date de rendu de fin de projet.

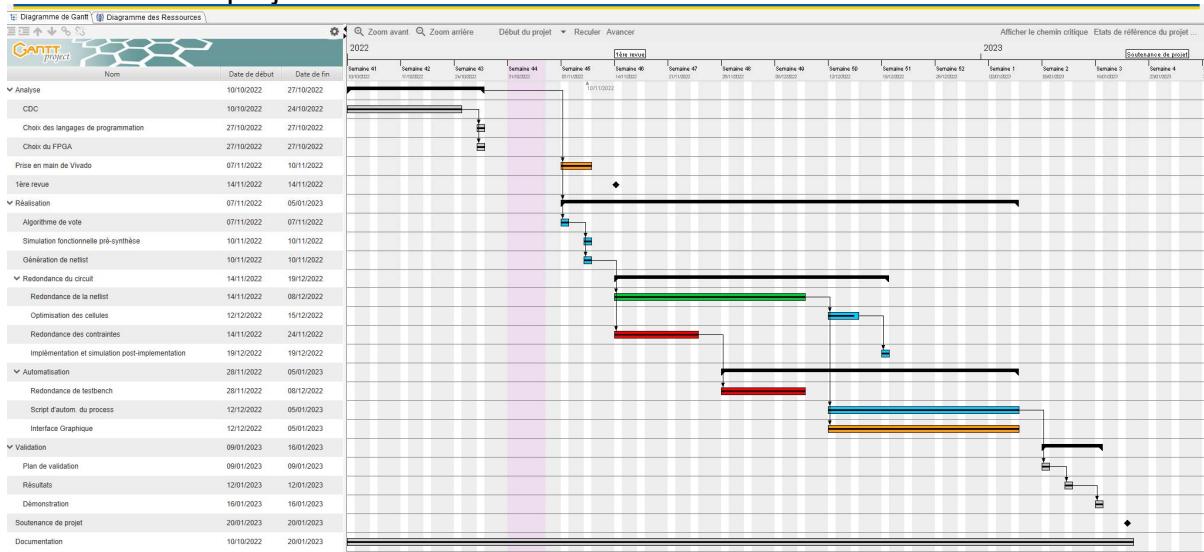


Figure 39 : Diagramme de Gantt réel

8. Preuve par compétence

Au cours des séances de ce projet, nous avons renforcé plusieurs compétences. En effet, lors de ce projet nous avons proposé une solution technique répondant à la problématique d'optimisation de la place occupée dans un FPGA par un circuit tolérant aux fautes. Le projet est rendu complexe de par sa diversité. Premièrement par la mise en place d'un algorithme permettant de rendre un circuit tolérant aux perturbations sans affecter son fonctionnement. Dans un second temps, optimiser l'espace occupé par le circuit lors de son implémentation dans un FPGA choisi par l'utilisateur. Enfin, la mise en place d'une automatisation efficace et la plus simple d'utilisation.

Toutes ces tâches nous ont permis de mettre en œuvre un grand nombre de compétences techniques que nous avons développé lors de nos différents cours à PHELMA. Nous pensons notamment à la capacité de construction d'algorithmes complexes, quel que soit le langage de programmation utilisé. Nous avons vu dans ce rapport que nous avons aussi bien utilisé des langages tels que python, TCL, ou bash, ce qui a par ailleurs renforcé notre capacité d'adaptation. En effet, nous n'avons pas développé nos scripts en fonction des langages que nous maîtrisions mais bel et bien en fonction de leurs spécificités. Pour illustrer nos propos, ni le python ou le TCL ne sont enseignés au cours de notre formation.

Nous pensons également au cours de vérification qui nous a donné des bases et un angle de vue permettant de mettre en œuvre un plan de vérification efficace permettant de couvrir un maximum de fonctionnalités du système développé.

Un autre exemple de compétences que nous avons mis en œuvre lors de la réalisation de ce projet sont les connaissances acquises lors des cours de conception de circuits logiques qui nous ont permis d'analyser les résultats d'optimisation d'espace, de consommation et de timing obtenus en post implémentation.

En prenant du recul, nous aurions dû accorder plus de temps pour l'étude des différentes méthodes de protection contre les perturbations existantes. Notre choix de travailler sur le TMR a été orienté par le cours de VLSI avancée que nous avions suivi cette année. D'autre part, ce que nous avons trouvé difficile au niveau gestion de projet, c'est la répartition des tâches entre les membres du groupe. En effet, nous avions beaucoup de tâches à réaliser, mais un grand nombre d'entre elles était mineures en termes de temps de travail.

Enfin au travers de cet enseignement, nous avons été amenés à travailler dans une totale autonomie pour comprendre les systèmes complexes à l'étude. Cela nous a aussi demandé de chercher par nous-même des solutions non triviales.

9. Conclusion

Pour conclure, nous avons mis en place une solution automatique de redondance afin de protéger un FPGA des perturbations.

Pour ce faire, nous avons réalisé une interface graphique pour exécuter un batch permettant de lancer toutes les étapes nécessaires pour faire la redondance d'un circuit.

En plus de nos connaissances personnelles sur les circuits numériques, nous avons appris à étudier des fichiers de netlist.

Pour tester notre circuit, nous avons commencé à regarder l'injection de fautes dans le simulateur xsim de Vivado qui permet d'évaluer la robustesse de notre solution face à des fautes transitoires. Enfin, nous avons prévu d'implémenter une solution de TMR sur une carte FPGA d'ici la soutenance orale.

10. Bibliographie

- [1] J. B. Ferron, L. Anghel and R. Leveugle, "Towards low-cost soft error mitigation in SRAM-based FPGAs: A case study on AT40K," *2012 IEEE 3rd Latin American Symposium on Circuits and Systems (LASCAS)*, Playa del Carmen, Mexico, 2012, pp. 1-4, doi: 10.1109/LASCAS.2012.6180356.
- [2] M. Ben Jrad and R. Leveugle, "Automated design flow for no-cost configuration error detection in sram-based FPGAs," *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 2013, pp. 1-6, doi: 10.1109/ReConFig.2013.6732272.
- [3] R. Leveugle and M. Ben Jrad, "On improving at no cost the quality of products built with SRAM-based FPGAs," *Fifth Asia Symposium on Quality Electronic Design (ASQED 2013)*, Penang, Malaysia, 2013, pp. 295-301, doi: 10.1109/ASQED.2013.6643603.
- [4] M. Ben Jrad and R. Leveugle, "Evaluating a low cost robustness improvement in SRAM-based FPGAs," *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, Chania, Greece, 2013, pp. 173-174, doi: 10.1109/IOLTS.2013.6604072.
- [5] Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide (UG953) [user guide]. https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries/LUT6_2 (06/01/2022)
- [6] Lorena Anghel, L.A. (2022). course_Robustesse_DevDurable2022 [Diapositives]