BORDEAUX INP Enseirb-Matmeca

Projet avancé en Systèmes Embarqués

Implémentation matérielle de l'algorithme de Viterbi sur carte FPGA

Axel Veyssière & Lucas Clairet & Théo Le Mestre & Thibaut Guittet

Tuteur : Camille Leroux

27 octobre 2023 - 26 janvier 2024

Troisième année, Filière Électronique, Option Systèmes Embarqués

Table des matières

Introduction

Lors de la transmission de données d'un système à autre, des erreurs peuvent apparaître sur les données qui peuvent compromettre la transmission toute entière. Ces erreurs peuvent être dues à différents problèmes tels qu'un rapport signal à bruit trop faible, une perturbation du système lors de l'envoi, etc. Il existe, pour des communications filaires et même sans-fil des moyens de diminuer le risque d'erreurs et même des moyens de les corriger grâce à des algorithmes appelés *codes correcteurs d'erreurs*.

L'algorithme de Viterbi est un de ces codes correcteurs d'erreurs. Ce code est dit de type convolutif ce qui signifie que la sortie d'un codeur tel que Viterbi dépend non seulement de la valeur du symbole (un bit dans notre cas) à transmettre mais également des valeurs des symboles transmis précédemment et des valeurs de codage associées.

L'objectif de ce projet est d'implémenter sur une puce reprogrammable FPGA un système de décodage de l'algorithme de Viterbi afin de décoder un message reçu par UART sur la carte FPGA. Pour cela, le projet est divisé en deux parties : une partie logicielle et une partie matérielle.

La première partie logicielle porte sur la création d'un encodeur et décodeur Viterbi en Python à l'aide du logiciel *AFF3CT*, spécialisé dans la simulation et les tests de transmissions utilisant des codes correcteurs d'erreurs. Afin d'utiliser celui-ci, le *wrapper* (ou enveloppe en français) Python nommé *Py_AFF3CT* est utilisé, permettant l'utilisation du langage Python.

Dans la deuxième partie matérielle, un système de décodage de l'algorithme de Viterbi est implémenté sur carte FPGA à l'aide du logiciel *Vivado* d'*AMD*. Les langages VHDL et System Verilog sont utilisés pour créer et implémenter les modules nécessaires.

L'objectif final est de pouvoir encoder des données d'une image par exemple, de simuler leur transfert dans un canal de transmission grâce à *AFF3CT* avant de décoder ces données par la partie matérielle et logiciel pour ensuite comparer les résultats des deux décodages et ainsi valider le fonctionnement de la partie matérielle (voir Figure 1).

Une partie des outils Python et des modules SystemVerilog utilisés ont été fourni au début du projet.

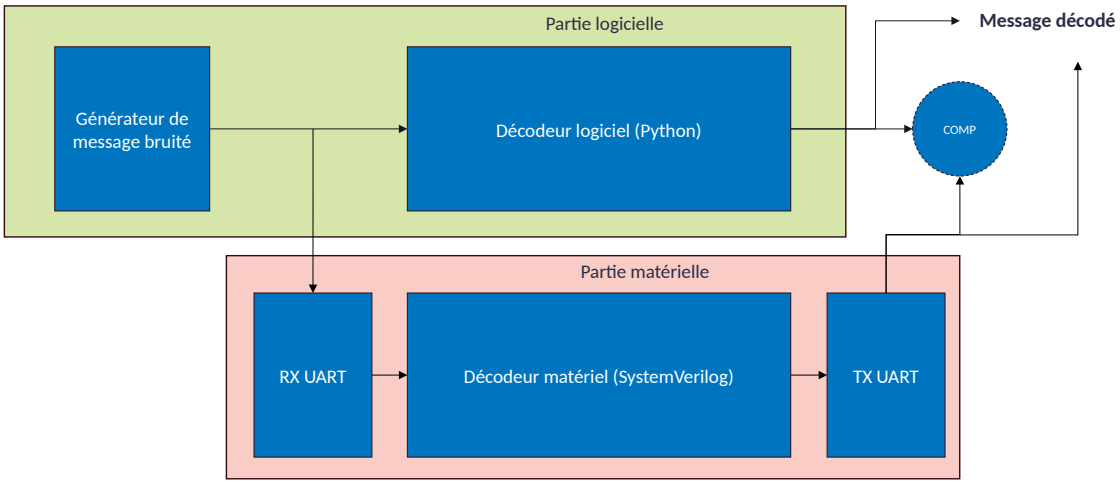

Figure 1 – Structure du projet réalisé

1 Algorithme de Viterbi

Comme expliqué en introduction, l'algorithme de Viterbi est un code correcteur d'erreur de type convolutif. Il permet de décoder des caractéristiques reçues et d'en déduire la valeur du bit transmis avec ses caractéristiques. Cet algorithme s'emploie donc conjointement avec un système permettant d'encoder les bits à transmettre. La probabilité d'avoir une erreur ou bien la *vitesse* à laquelle l'algorithme peut décoder les bits transmis sont tous deux dépendant de certains paramètres de l'encodeur.

Ainsi, pour diminuer le risque d'erreur de transmission, il est possible d'augmenter le nombre de bits de la signature (ou caractéristique) du bit à transmettre : n bits de signature divisant le rendement par n.

Avec l'algorithme de Viterbi, la valeur d'un bit décodé repose sur la convergence du système vers une valeur donnée. Ainsi, à mesure que des données sont reçues, la probabilité que les bits décodés précédemment aient une valeur correcte augmente. Il est possible d'*accélérer* cette convergence en diminuant la longueur de contrainte, c'est à dire le nombre de bits servant à l'encodage de la valeur à envoyer. En effet, bien qu'en théorie le décodeur peut ne jamais converger vers une unique solution, en pratique il suffit d'attendre le décodage de $6 \times K$ bits où K est la longueur de contrainte pour que le chemin parcouru converge à un moment donnée en remontant les états. En revanche, diminuer la longueur de contrainte revient également à augmenter le risque qu'un des bits décodé soit faux. Le principe de convergence signifie que les derniers bits décodés seront sûrement faux et les premiers auront le plus de probabilité d'être corrects.

C'est pourquoi il est nécessaire de trouver un équilibre entre ces deux paramètres de l'encodeur afin que l'algorithme soit le plus efficace possible. La Figure 2 montre un exemple d'encodeur, celui-ci a une longueur de contrainte de 3 et 2 bits de signature sont générés.

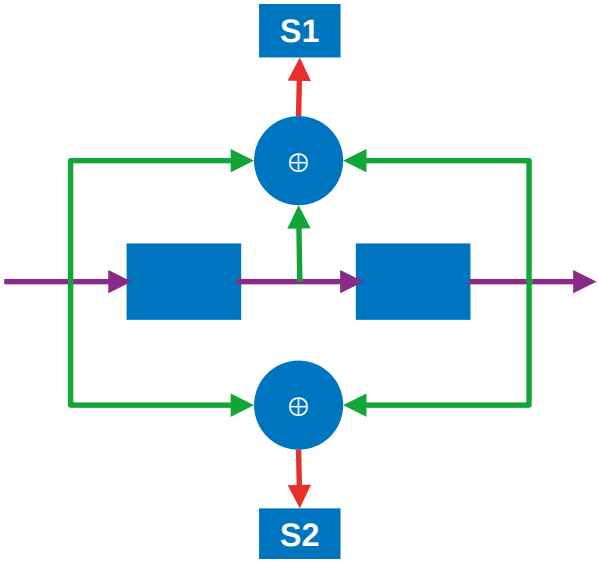

Figure 2 – Encodeur de données générant deux signatures.

Afin de représenter graphiquement l'évolution de l'état du codeur à chaque bit transmis, un treillis est créé. Chaque état du treillis comporte un nombre de nœuds défini, le nombre de nœuds est de 2^{K-1} où K est la longueur de contrainte. Pour chaque nœud, seulement deux chemins sont possibles pour passer à l'état du codeur suivant, selon que le bit envoyé soit un 1 ou un 0 (voir Figure 3).

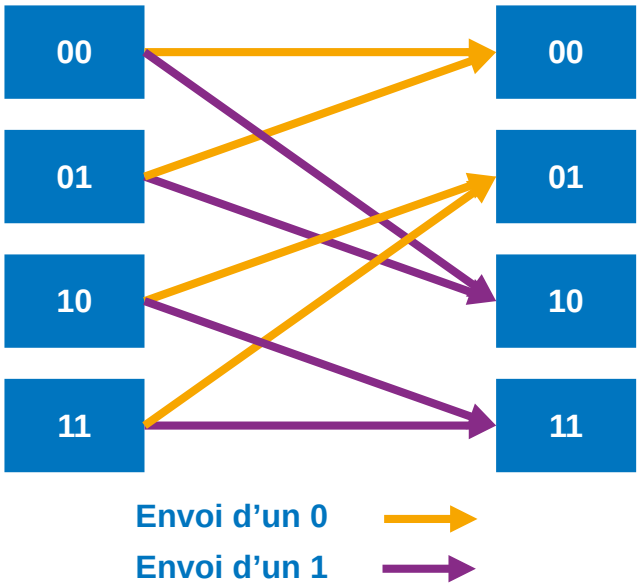

Figure 3 – Treillis représentant graphiquement l'évolution de l'encodeur selon le bit envoyé.

À la réception, ce n'est pas le bit de la donnée qui est reçu mais sa caractéristique, dépendant des états précédents de l'encodeur. Ainsi, l'envoi d'un 1 ou un 0 correspond à une signature reçue (voir Figure 4). Comme il y a plus de signatures possibles que de chemins possibles pour chaque nœud, certains chemins sont théoriquement *impossibles* suivant les signatures reçues, témoignant alors d'une erreur.

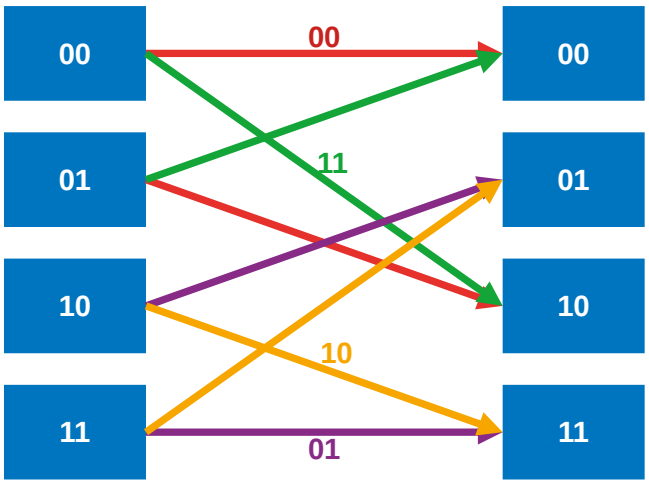

Figure 4 – Treillis représentant graphiquement l'évolution de l'encodeur selon la signature reçue.

La caractéristique d'un bit envoyé est modulé avant d'être envoyé, cette modulation équivaut à un

passage dans le monde analogique et donc à l'introduction de bruit sur la caractéristique. Ainsi, lors de la démodulation, le bit reçu est quantifié selon sa valeur analogique. Ensuite, la distance entre cette valeur quantifiée et les valeurs des 0 et 1 *idéaux* quantifiés est calculée (Figure 5). Plus la distance calculée par rapport à un bit idéal est élevée, plus la probabilité est faible que le bit initialement transmis soit celui-là.

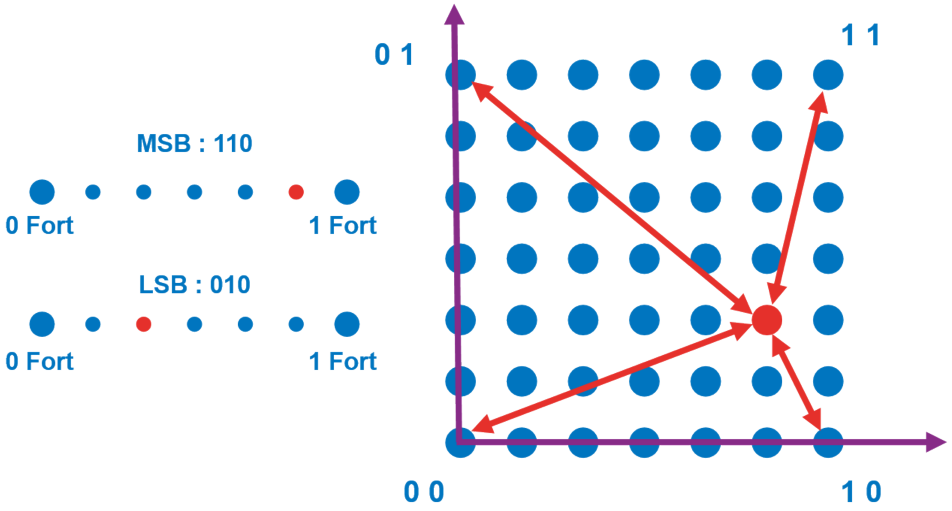

Figure 5 – Représentation graphique de la distance entre une caractéristique idéale théorique et une caractéristique reçue.

À partir des distances calculées, on peut attribuer une métrique à chaque branche correspondant à l'addition des distances de chaque bit de la signature. Cette métrique de branche est additionnée à la métrique de nœud de l'état précédent, puis, chaque nœud de l'état actuel récupère la valeur du chemin venant vers elle avec la métrique la plus faible. Chaque nœud d'un état conserve ainsi l'origine de l'état précédent le plus probable. Après un nombre d'état assez important correspondant à la longueur de convergence, tous les chemins finissent par converger en un seul chemin survivant (voir Figure 6) remontant jusqu'au premier état correspondant au début de la transmission. Ce chemin survivant permet ainsi de déterminer le message initialement envoyé, avec un risque d'erreur réduit.

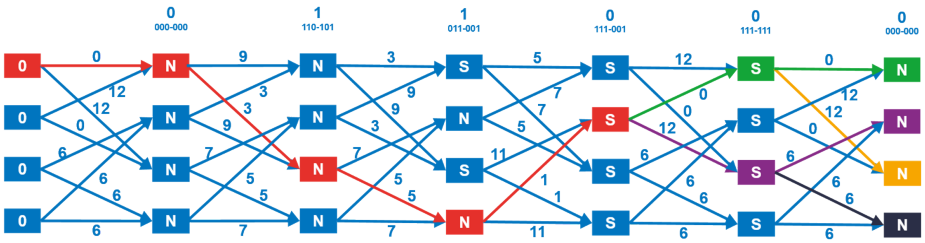

Figure 6 – Représentation graphique du chemin survivant, convergeant après avoir remonté deux états.

2 Développement logiciel

Comme expliqué dans l'introduction, la partie logicielle est réalisée en langage Python avec l'aide de la librairie *PY_AFF3CT*. Cette librairie fournie directement des fonctions afin de générer des trames de données aléatoires, de les bruiter, de les moduler et les démoduler.

2.1 Encodage des données

Le script Python permettant de faire l'encodage des données étant déjà fourni, celui-ci n'a pas été modifié. L'équivalent matériel est donnée Figure 7. La longueur de contrainte de cet encodeur est de $K = 4$. Ainsi, pour le décodeur, $2^{K-1} = 8$ nœuds par état sont nécessaires.

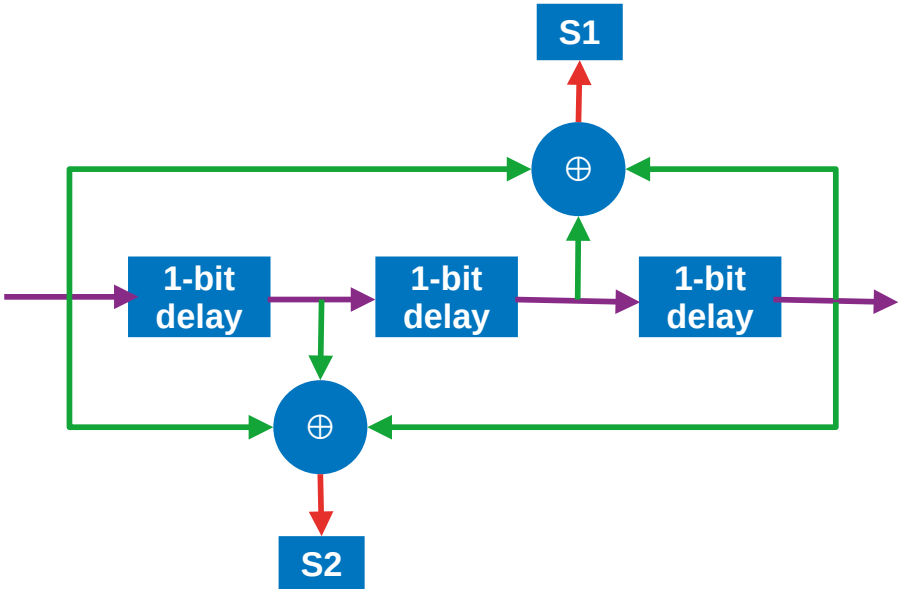

Figure 7 – Schéma bloc de l'encodeur Viterbi.

2.2 Décodage des données

Le script Python permettant le décodage des données fourni au début de ce projet comportait des erreurs. Il a donc été réécrit.

Le décodeur commence par calculer les métriques de branches pour chaque donnée reçue.

```
BM[3, :] =  r_in[0, 0:self.N:2] + r_in[0, 1:self.N:2] + 6
BM[2, :] =  r_in[0, 0:self.N:2] - r_in[0, 1:self.N:2] + 6
BM[1, :] = -r_in[0, 0:self.N:2] + r_in[0, 1:self.N:2] + 6
BM[0, :] = -r_in[0, 0:self.N:2] - r_in[0, 1:self.N:2] + 6
```

Ensuite, pour chaque état, et donc pour chaque caractéristique reçue, les métriques de nœuds sont calculées. La métrique de nœud correspondant au minimum de l'addition des métriques de nœuds des états précédents avec les métrique de branches correspondantes.

```
SM[0, i] = np.minimum(SM[0, i - 1] + BM[0, i - 1], SM[1, i - 1] + BM[3, i - 1])
SM[1, i] = np.minimum(SM[2, i - 1] + BM[2, i - 1], SM[3, i - 1] + BM[1, i - 1])
SM[2, i] = np.minimum(SM[4, i - 1] + BM[1, i - 1], SM[5, i - 1] + BM[2, i - 1])
SM[3, i] = np.minimum(SM[6, i - 1] + BM[3, i - 1], SM[7, i - 1] + BM[0, i - 1])
SM[4, i] = np.minimum(SM[0, i - 1] + BM[3, i - 1], SM[1, i - 1] + BM[0, i - 1])
SM[5, i] = np.minimum(SM[2, i - 1] + BM[1, i - 1], SM[3, i - 1] + BM[2, i - 1])
SM[6, i] = np.minimum(SM[4, i - 1] + BM[2, i - 1], SM[5, i - 1] + BM[1, i - 1])
SM[7, i] = np.minimum(SM[6, i - 1] + BM[0, i - 1], SM[7, i - 1] + BM[3, i - 1])
```

Par la suite, pour chaque nœud, le chemin pour y arriver est enregistré dans un tableau comportant le numéro de l'état précédent.

```
path[0, i] = np.where(SM[0, i - 1] + BM[0, i - 1] < SM[1, i - 1] + BM[3, i - 1], 0, 1)
path[1, i] = np.where(SM[2, i - 1] + BM[2, i - 1] < SM[3, i - 1] + BM[1, i - 1], 2, 3)
path[2, i] = np.where(SM[4, i - 1] + BM[1, i - 1] < SM[5, i - 1] + BM[2, i - 1], 4, 5)
path[3, i] = np.where(SM[6, i - 1] + BM[3, i - 1] < SM[7, i - 1] + BM[0, i - 1], 6, 7)
path[4, i] = np.where(SM[0, i - 1] + BM[3, i - 1] < SM[1, i - 1] + BM[0, i - 1], 0, 1)
path[5, i] = np.where(SM[2, i - 1] + BM[1, i - 1] < SM[3, i - 1] + BM[2, i - 1], 2, 3)
path[6, i] = np.where(SM[4, i - 1] + BM[2, i - 1] < SM[5, i - 1] + BM[1, i - 1], 4, 5)
path[7, i] = np.where(SM[6, i - 1] + BM[0, i - 1] < SM[7, i - 1] + BM[3, i - 1], 6, 7)
```

Enfin, le chemin survivant est évalué en partant de la métrique de nœud la plus faible à l'état final et en remontant jusqu'au début de la transmission.

```
tmp = SM[0, self.K]
current_state = 0

for i in range(1, 8):
    if tmp > SM[i, self.K - 1]:
        tmp = SM[i, self.K - 1]
        current_state = i

for i in range(self.K, 0, -1):
    if current_state < 4:
        r_out[0, i - 1] = 0
    else:
        r_out[0, i - 1] = 1
    current_state = path[current_state, i]
```

2.3 Forçage de la convergence

Comme expliqué dans la partie théorique de l'algorithme de Viterbi, les chemins convergent après un nombre de bits transmis suffisant, dépendant de la longueur de contrainte où la convergence est plus ou moins assurée après $L = 6 \times K$ où K est la longueur de contrainte. Ainsi, pour forcer la convergence sur les bits du message, des bits supplémentaires sont ajoutés à la fin du message à transmettre de façon à s'assurer que le dernier bit du message décodé aura bien convergé.

Après décodage, ces bits sont retirés afin de ne garder que le message.

2.4 Envoi du message à décoder sur la carte FPGA

Le message après démodulation est non seulement transmis au décodeur logiciel mais également au décodeur matériel par l'intermédiaire d'une liaison UART.

Les valeurs quantifiées des caractéristiques reçues sont de types `float32`, puisqu'elles ont une valeur entre -3 et 3 et sont toujours entières, ces valeurs doivent être réadaptées pour être envoyé au matériel. Chaque caractéristique quantifiée doit être envoyé en 8 bits, 4 bits pour chaque bit quantifié qui sont alors concaténés.

```
def send(self, r_in):
    ser = serial.Serial('/dev/ttyUSB1', 9600, bytesize=serial.EIGHTBITS)
    for i in range(0, self.N, 2):
```

```
        tmp1 = int(((3 - r_in[0, i])/2)*7/3)
        tmp2 = int(((3 - r_in[0, i + 1])/2)*7/3)
        tmp = tmp2 + (tmp1 << 3)
        print(f'{tmp:06b}')
        ser.write(tmp.to_bytes(1, byteorder='big', signed=False))
    return 0
```

2.5 Test de l'encodage et du décodage logiciel

Le fonctionnement de l'encodeur et du décodeur logiciel est vérifié grâce au module monitor d'*AFF3CT*. Celui-ci permet de mesurer le taux d'erreur binaire et le taux d'erreur de trame afin d'évaluer les performances.

Les performances sont évalués pour des rapports signal à bruit allant de 0 à 5dB. Compte tenu de la performance de l'ordinateur utilisé pour faire les mesures, il n'a pas été possible d'aller au delà de cette valeur afin de ne pas passer trop de temps à simuler.

Pour chaque valeur de rapport signal à bruit, le test s'arrête après 50 erreurs relevées, valeur aussi choisie pour réduire le temps de simulation. Chaque trame comporte 100 bits de message.

Eb/N0 (dB)	Frame number	BER	FER	Tpt (Mbps)
0.00	53	1.20e-01	9.43e-01	0.01
0.50	58	8.93e-02	8.62e-01	0.01
1.00	63	7.08e-02	7.94e-01	0.01
1.50	88	3.90e-02	5.68e-01	0.01
2.00	132	1.61e-02	3.79e-01	0.01
2.50	181	1.39e-02	2.76e-01	0.01
3.00	364	5.30e-03	1.37e-01	0.01
3.50	776	1.92e-03	6.44e-02	0.01
4.00	1626	8.73e-04	3.08e-02	0.01
4.50	3931	3.15e-04	1.27e-02	0.01
5.00	8771	1.29e-04	5.70e-03	0.01

On peut alors tracer la courbe montrant l'évolution du taux d'erreur binaire et de trame montrée en Figure 8. Il aurait été intéressant de comparer ces valeurs avec l'évolution de ces taux d'erreur sans encodage et décodage par Viterbi.

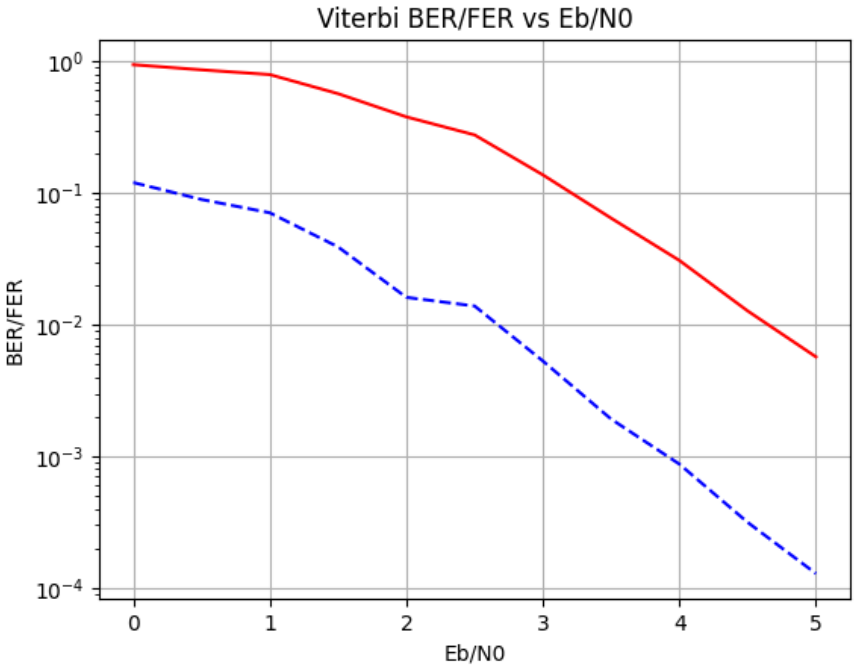

Figure 8 – Évolution du taux d'erreur binaire et de trame en fonction du rapport signal à bruit.

3 Partie matérielle

La partie matérielle nécessitait l'ajout de plusieurs fonctionnalités au projet :

1. Une interface de communication entre le carte et l'ordinateur ;
2. Un top level permettant d'inclure et interconnecter correctement cette interface et le décodeur ;
3. La distribution correcte des signaux d'activation des module et de validation des données.

3.1 Communication UART

Pour effectuer la communication entre le canal et le décodeur matériel, nous avons commencé par implémenter une liaison UART avec le FPGA. Nous avons essayé plusieurs IPs et nous sommes finalement arrêtés sur une solution développée par Yuya Kudo et mis à disposition sous la licence MIT (utilisable dans le cadre de ce projet).

Ces IPs sont interconnectés en System Verilog avec le décodeur.

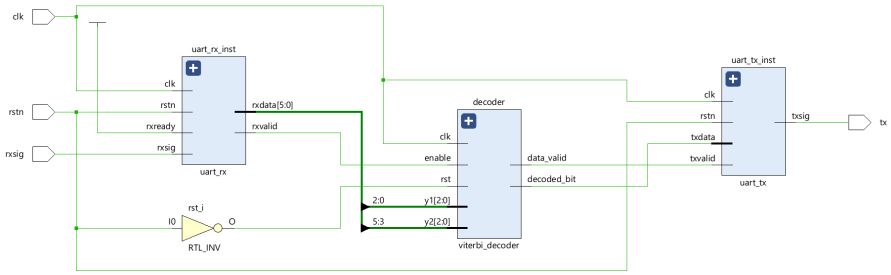

Figure 9 – Schéma bloc du décodeur et des modules UART

3.1.1 Taille des trames

Il est à noter que l'UART est utilisé la majorité du temps pour envoyer des caractères ASCII, soit des messages de 8 bits. Dans notre configuration, nous n'avons besoin d'envoyer pour chaque couple de signatures que 6 bits (deux signatures échantillonnées sur trois bits chacune) mais nous avons fait le choix d'envoyer tout de même 8 bits à chaque fois, en ajoutant deux bits de remplissage au début.

Cette décision est motivée par le fait que la bibliothèque Python permettant d'envoyer des données via une liaison série semble mal supporter les envois de « non standards » de seulement 6 bits et créait des erreurs. Ce choix ralentit légèrement le débit maximal théorique mais nous a permis de simplifier la résolution de ce problème pour le moment.

3.1.2 Vitesse de communication

La vitesse du décodeur est limité par la vitesse des transmissions UART. En effet, un seul cycle d'horloge suffit au décodeur pour traiter les données en entrée et se mettre en attente de la prochaine donnée. Par ailleurs, il faut plus de bits en entrée que le décodeur ne produit de bits en sortie, l'élément limitant dans la vitesse est donc la réception des données.

Il est ainsi nécessaire de les transmettre le plus vite possible. La vitesse de décodage est déterminée par la formule suivante :

$$V_{decodage} = \frac{B}{N_{input}}$$

où B est la vitesse de transmission en entrée et N_{input} est le nombre de bits en entrée (ici 8 bits de données, 1 bit de start et 1 bit de stop ou 10 bits au total).

On a donc, pour une baudrate en entrée de 115200 bauds, un débit de 11520 bit/s. Ce débit pourrait être augmenté en augmentant le baudrate mais cela nécessiterait de tester le système pour vérifier que rien ne dysfonctionne suite à cela.

En conséquence de cette limitation de vitesse en entrée, il a été nécessaire d'ajouter des signaux de validation des données nommés enable pour l'entrée et data_valid pour la sortie du décodeur.

3.2 Testbench

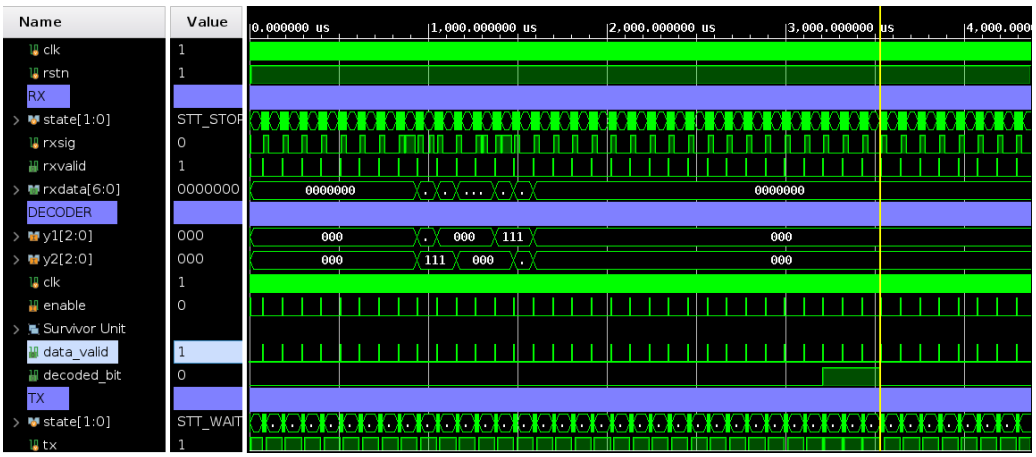

Figure 10 – Graphe du testbench réalisé

On calcule au préalable une suite de valeur possibles pour l'émetteur TX de l'ordinateur (connecté au RX de la carte) et on suit les valeurs de chaque signaux internes à l'architecture sur le graphe ainsi obtenu. Cela nous permet ainsi de vérifier que le signal en entrée est bien interprété comme des signatures, que le décodeur est bien activé après que ces signatures soient devenues disponibles, que le décodeur donne la bonne sortie et que l'émetteur TX de la carte renvoie les bonnes valeurs.

3.3 FIFO

L'illustration suivante montre la communication entre les deux. Deux mémoires de type FIFO (First In First Out) sont utilisés permettant d'envoyer des données dans leur ordre d'arrivée même si 2 périphériques fonctionnent à des vitesses différentes. Une mémoire FIFO agit donc comme un tampon, permettant de réguler le débit et d'éviter la perte de données.

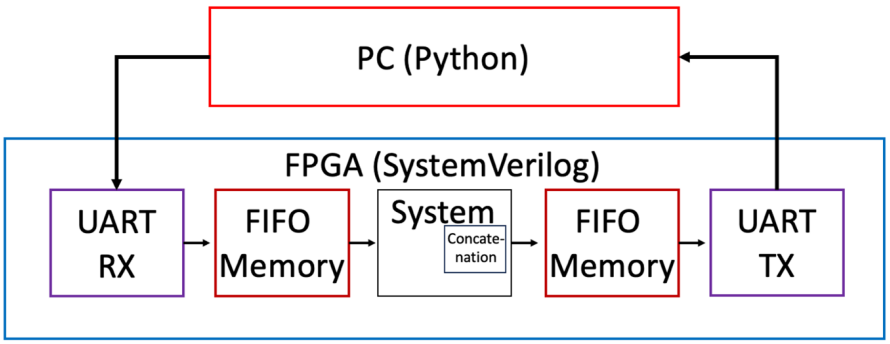

Figure 11 – Schéma des transferts de données

Les résultats du décodeur Viterbi sont sur 1 bit (nord ou sud) mais notre communication est sur 8 bits. Nous aurions pu envoyer 1 bit de résultat avec 7 « 0 » mais nous avons fait le choix de concaténer 8 résultats (8 bits) avant le stockage en mémoire FIFO divisant ainsi par 8 le nombre de bits envoyé du FPGA vers le PC, nous augmentons donc le rapport Débit utile/Débit brut et utilisons une transmission standard sur 8 bits.

Conclusion

Au terme de ce projet, nous avons réussi à débugger le décodeur logiciel, à créer une interface entre le décodeur matériel et le PC via une communication UART et valider leurs fonctionnements indépendamment.

Nous ne sommes malheureusement pas arrivés au terme du projet pour cause de manque de temps. Pour cela, nous aurions du, en plus :

- Implémenter la réception par le code Python des données renvoyées par le décodeur matériel ;
- Implémenter en Python un module de comparaison pour vérifier que les décodeurs logiciels et matériels ont bien les mêmes résultats ;
- Générifier l'encodeur et le décodeur afin de pouvoir changer le polynôme d'encodage.

Cependant, lors de ce projet, nous avons développé des compétences en SystemVerilog que nous n'avions pas avant (ayant seulement utilisé le VHDL jusque là), en Python et plus largement en communications numériques ainsi que sur la maîtrise de Git.

Références

1. Cassagne A. et al, Python to AFF3CT (2020), GitHub repository
2. Leroux C., Viterbi (2023), GitHub repository Leroux C., Module EN218, Tutorial
3. Tonnelier T., Module EN218, Tutorial
4. Kudo Y., SystemVerilog-UART (2019), GitHub repository