MANSOUR REMY / GALLET SAMUEL

Projet SLE
AES - contres mesures

Introduction :

Dans le cadre de notre formation et de notre futur emploi d'ingénieur, il est important de réaliser des projets types, semblables à ceux que nous réaliserons au sein d'une entreprise. Le but du projet en troisième année SLE est de concrétiser un travail à grande échelle où nous nous intéressons à tous les niveaux de conception d'un système embarqué. Ce projet nous a permis de mettre en pratique nos compétences et nos connaissances pour identifier, résoudre les problèmes, et obtenir des résultats espérés et cohérents. Cependant ce projet requiert plusieurs heures de travail et nous avons du faire preuve de « savoir être » pour s'impliquer, être efficaces et surtout être organisés.
Dans ce rapport nous explorerons tous le flow de conception de notre système en commençant par une présentation détaillée du cahier des charges. Nous verrons ensuite les différentes étapes de conceptions avant de soumettre les validations planifiées et réalisées. Nous finirons par expliquer le fonctionnement de notre système à travers le manuel utilisateur avant de faire un point sur la démonstration cliente et l'évolution de notre planning prévisionnel.

I. Cahier des charges

De nos jours, la sécurité numérique est un domaine clef dans l'innovation électronique et informatique. En effet, beaucoup de données sensibles, comme des informations personnelles ou encore de l'argent, transitent entre plusieurs systèmes et peuvent potentiellement être interceptées. Il faut, pour cela, protéger les données des personnes malveillantes qui tenteraient d'y avoir accès. Cependant les algorithmes de chiffrement sont lourds et coutes cher en performance. Ils nécessitent donc des accélérateurs matériels.
Toutefois, les algorithmes tels que l'AES ont beau être complexes et fiables, ils ne sont pas infaillibles face aux attaques par canaux cachés et aux attaques par fautes. Il est donc nécessaire de développer des contres mesures qui rendront la tâche plus difficile aux attaquants.
Dans le cadre de ce projet, nous nous intéresserons à une architecture d'AES relativement résistante aux attaques par fautes et aux attaques par canaux cachés. L'architecture est représentée figure 1 :

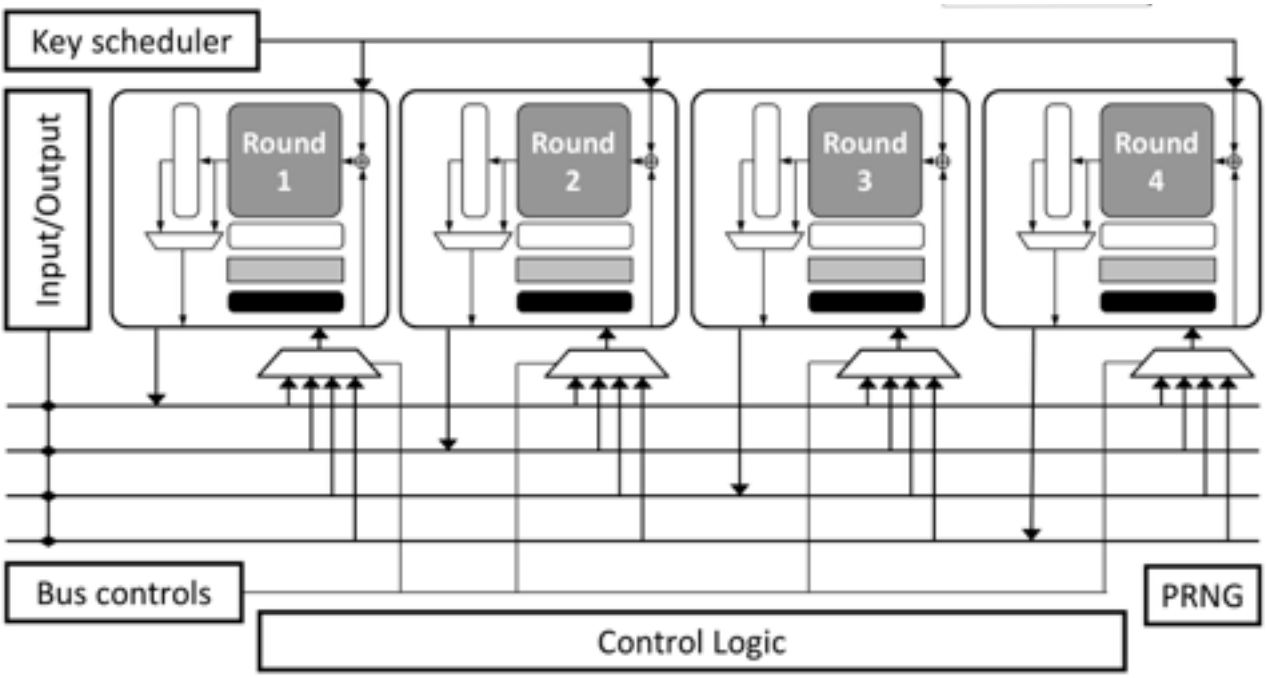

figure 1 - Représentation architecture AES

L'architecture est partagée en 4 blocs de calculs tous les 4 branchés sur un bus de communication. Le chiffrement d'une donnée (de 128 bits dans notre cas) en utilisant l'algorithme AES se fait en 10 étapes de calculs (voir http://fr.wikipedia.org/wiki/Advanced_Encryption_Standard pour plus de détail sur le calcul de l'AES).
Dans le cas de notre architecture, à chaque fin d'étape du chiffrement, les données sont envoyées sur le bus et déplacées vers un autre bloc de chiffrement de manière aléatoire. De plus, en parallèle, un des blocs non alloué pour le calcul de chiffrement va effectuer un calcul de vérification du chiffrement. A chaque fin d'étape, une comparaison est réalisée sur les données issues du chiffrement et celles issues de la vérification. Si une erreur a été injectée dans l'un des deux blocs, l'un des calculs sera fauté et la comparaison lèvera une alarme. Ceci confère au composant AES une forte robustesse contre les attaques par fautes.
Les deux blocs non alloués pour la vérification et le chiffrement effectuent un calcul en parallèle, utile ou non au chiffrement, afin de masquer la consommation des blocs utiles et être robuste aux attaques par canaux cachés.

Le problème dans cette architecture est que si l'attaquant injecte une faute sur l'instance de vérification, une « fausse » alarme sera levée. En effet, le système détectera la faute mais le chiffrement ne sera pas corrompu. Le but de ce projet est donc de minimiser le nombre de fausses alarmes en insérant un bloc détecteur qui analysera la sortie de l'instance de vérification et déterminera si l'instance de vérification est attaquée ou non. Ainsi, si le bloc de vérification est fauté, le code détecteur le remarquera et ne lèvera pas l'alarme car le chiffrement sera correct.
Voici le tableau récapitulatif des cas qu'il faut traiter au niveau des alarmes :

Vérification	Detecteur	sens
0	0	Aucune erreur, on sort le texte chiffré
1	0	Erreur dans instance de chiffrement ou erreur double dans parité, on cache le texte chiffré
0	1	Erreur de détection de parité (très peu probable), on sort le texte chiffré
1	1	Erreur dans l'instance de vérification, on sort le texte chiffré

Dans un premier temps, nous nous étions fixé l'objectif d'insérer un code détecteur qui vérifierait l'instance de chiffrement, et multiplexerait les données en sortie de chiffrement. Si l'instance de détection remarquait une anormalité, la sortie du bloc de vérification aurait été redirigée vers la sortie de l'AES. Cependant, après réflexion, nous aurions les mêmes résultats avec une logique combinatoire moins importante en vérifiant juste l'instance de vérification.
Dans le cadre de ce projet, nous allons implémenter un code détecteur de type « détecteur de parité ». Cependant il faut que le code soit le plus générique possible pour que l'on puisse changer le type de détecteur facilement.

Après avoir ajouté le détecteur de parité au crypto-processeur mis à notre disposition, nous devrons l'implémenter sur carte FPGA autour d'une architecture composé d'un bus PLB, d'un processeur microblaze et d'autres composants type « boutons poussoirs » , « dip-switch » , « leds » en fonction du besoin. Le microblaze devra embarqué une application qui utilisera l'IP développé.

II. Conception et Implémentation du système

La première étape dans la conception de notre système, avant l'intégration sur carte, fut le développement du bloc détecteur qui vérifierait l'instance de vérification du chiffrement.

A. Développement du détecteur de parité

1. Architecture

Comme expliqué précédemment, nous devons développer un détecteur de type parité mais qui reste le plus générique possible pour pouvoir changer à volonté, et le plus facilement possible, le type du détecteur. Pour cela, nous avons choisi l'architecture représentée figure 2:

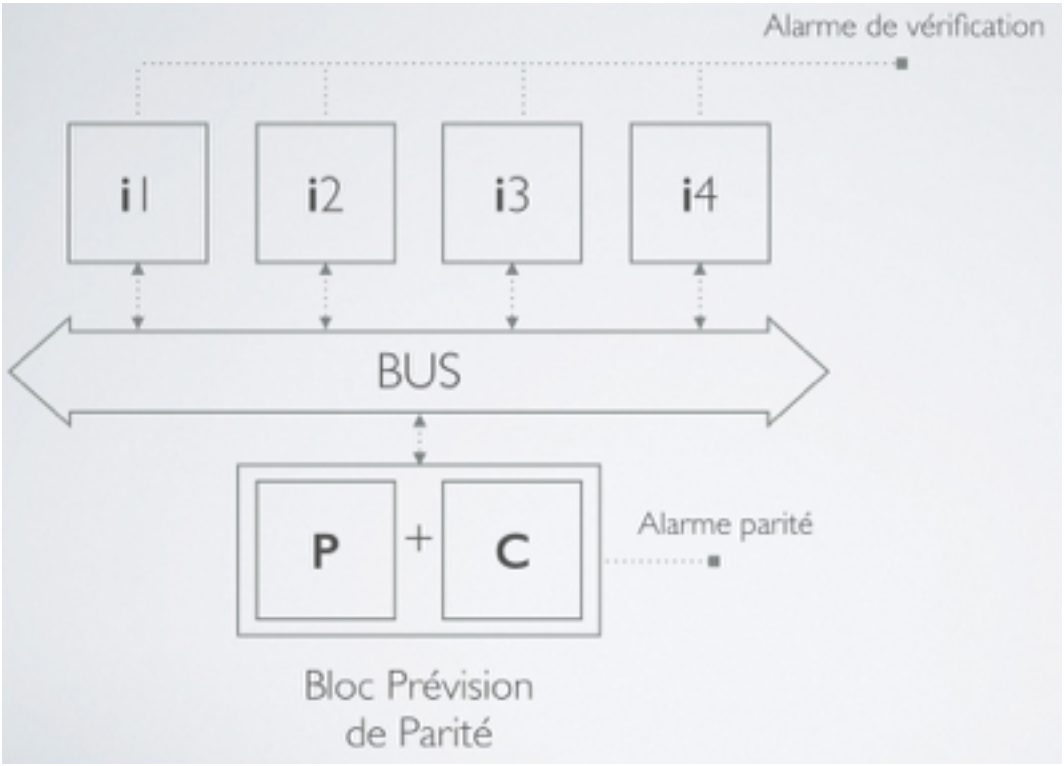

figure 2 - Architecture réalisée

Nous avons raccordé sur le bus, où sont connectées les 4 blocs de chiffrement, un bloc composé d'un sous bloc, qui effectuera la prévision de parité, et d'un comparateur. Dans le cas où nous souhaitons changer de détecteur, il suffit de remplacer les sources VHDL du bloc « prédicteur » moyennant le changement des entrées/sorties nécessaires au composant « predicteur ».
Dans le cas où le bloc « prévision de parité » détecte une anomalie sur l'instance de vérification, celui-ci lève une alarme appelée « alarme parité » qui va masquer l'alarme de vérification car celle-ci sera une fausse alarme.

2. Implémentation

L'architecture choisie pour la prévision de la parité est représentée figure 3:

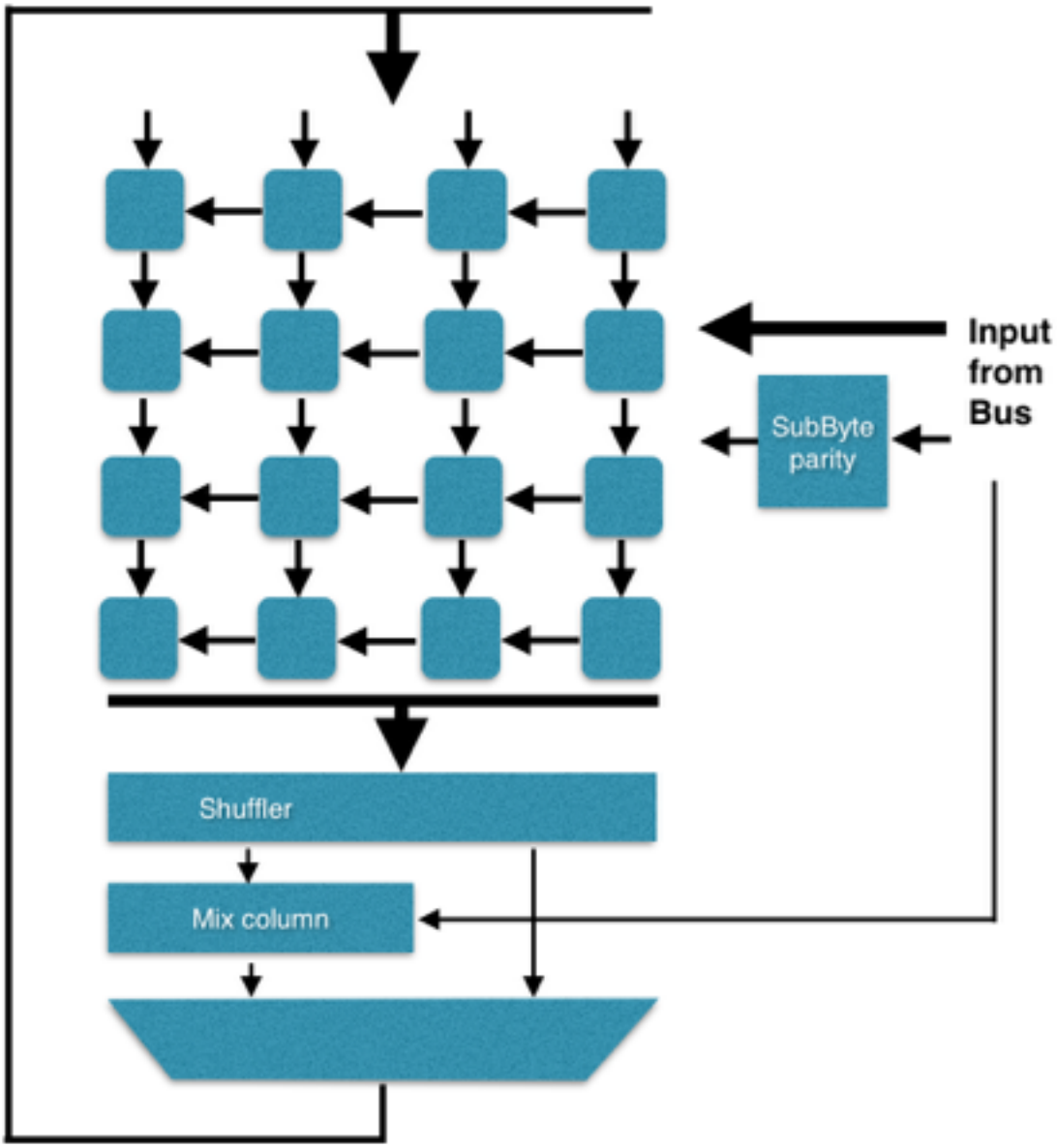

figure 3 - architecture calcul de parité

L'implémentation de la prévision de la parité se base sur le principe décrit ci-après. L'entrée subit une addition avec la clef correspondante à la première ronde puis on démasque l'entrée. En effet, les données circulant sur le bus sont toujours masquées pour des raisons de sécurité. Ensuite, pour chaque octet de l'entrée sur 128 bits, nous calculons la parité associée à l'octet en question et la parité associée au calcul « SubByte ». Nous chargeons ensuite la matrice d'état de la prévision de parité avec un xor entre les 2 parités précédemment calculées.
Une fois la matrice d'état chargé, à chaque cycle d'horloge, la dernière ligne de la matrice subit le un mélange de lignes puis un mélange de colonnes semblable à ceux utilisés lors du calcul du chiffrement. Ensuite, les valeurs sont renvoyées dans la matrice d'état pour la prochaine ronde. Pour les prochaines étapes, nous ne calculons que la parité pour l'opération « SubByte » que nous

associons à l'ancienne parité calculée. Ainsi, si une erreur est injectée, la parité sera propagée entre deux rondes de chiffrement et sera donc plus sensible pour la détection d'erreurs.
Pour effectuer les mêmes opérations lors du « mix column », nous récupérons sur le bus les données en sortie du « mix column » issues de l'instance de vérification. La matrice d'état est composée de registres 1 bit qui contiennent la parité associée à chaque octet de la matrice d'état de l'instance de vérification. De plus entre chaque opération (Subbytes, mix column et shuffler) nous avons conservé le pipeline présent dans les instances de chiffrement pour conserver une synchronisation entre le calcul de la parité et le chiffrement.

La difficulté dans cette synchronisation des données avec l'instance de vérification est de savoir quelle bloc est entrain de faire la vérification. En effet, plusieurs données doivent être partagées entre l'instance de vérification et notre bloc de détection. Il faut récupérer les mêmes données en entrée, obtenir les mêmes vecteurs de rotation des lignes et colonnes, et le même vecteur de réarrangement des données. Pour réaliser une telle synchronisation, nous nous sommes servis des vecteurs de bits utilisés pour l'allocation de l'utilité de chaque bloc. Nous avons donc rajouter un multiplexeur qui redirige la sortie de chaque bloc de calcul vers notre bloc de détection en fonction de ces vecteurs de bits. Cependant certaines données utiles comme la sortie de l'opération « mix column » ne pouvait pas être fournies au moment souhaiter pour cause d'indisponibilité du bus de communication. De plus, quand ces données étaient disponibles, le vecteur d'allocation des fonctions de chaque instance avait changé entre temps. La solution fut de stocker les données dans des registres pour les transférer à un moment opportun et de conserver une variable qui indique quelle instance effectue la vérification à chaque ronde de chiffrement. Toutefois augmentait la complexité de la synchronisation.
Les données en entrée de chaque instance possèdent, de plus, des informations primordiales, comme les masques utilisés pour protéger les données et les vecteurs de rotations affectés pour chaque instance de calcul. Si la synchronisation entre l'instance de chiffrement et notre bloc détecteur était incorrect, nous pouvions obtenir les bonnes données à chiffrer mais avec le mauvais vecteur de rotation et ainsi fausser notre calcul.

A chaque fin de ronde de chiffrement, le comparateur interne au bloc détecteur récupère un mot de 16 bits issues de la matrice de parité et le compare avec la parité de la donnée en sortie de l'instance de vérification. Si ces 2 parités ne sont pas équivalentes, le comparateur lève une alarme qui se propagera en sortie du crypto-processeur et annulera l'alarme de comparaison entre l'instance de chiffrement et de vérification, théoriquement levée.

Pour tester l'injection de fautes nous avons ajouté une entrée dans notre composant AES. Tant que cette entrée est à un, nous opérons un bit flip à l'entrée de l'opération « shuffler » dans une des instances choisies au hasard. Pour simuler une attaque par faute il suffit de forcer cette entrée à « 1 » pendant un cycle d'horloge.
Concernant le choix au hasard de l'instance fautée, nous avons utilisé un générateur congruentiel linéaire de la forme X(n+1) = a * X(n) + b mod c avec a = 1140671485, b = 12820163 et c = 2^24 (générateur utilisé sous Microsoft visual basic 6). Ce système génère un nombre aléatoire modulo 4 qui nous indique quelle instance doit être attaquée. Nous avons choisi celui-ci car c'est le seul que nous avons trouvé pouvant être utilisé avec des entiers signés sur 32 bits. En effet, nous étions contraint d'utiliser un nombre limité de bibliothèques VHDL.

Durant une ronde de chiffrement (équivalent à 8 cycles d'horloge), les données sensibles au chiffrement ne sont présentes à l'entrée de l'opération « shuffler » que pendant les 4 premiers cycles. Si nous voulons que la faute injectée ne soit pas silencieuse, il faut que celle-ci soit effective pendant les 4 premiers cycles d'une ronde de chiffrement.

2. Implantation et fonctionnement global

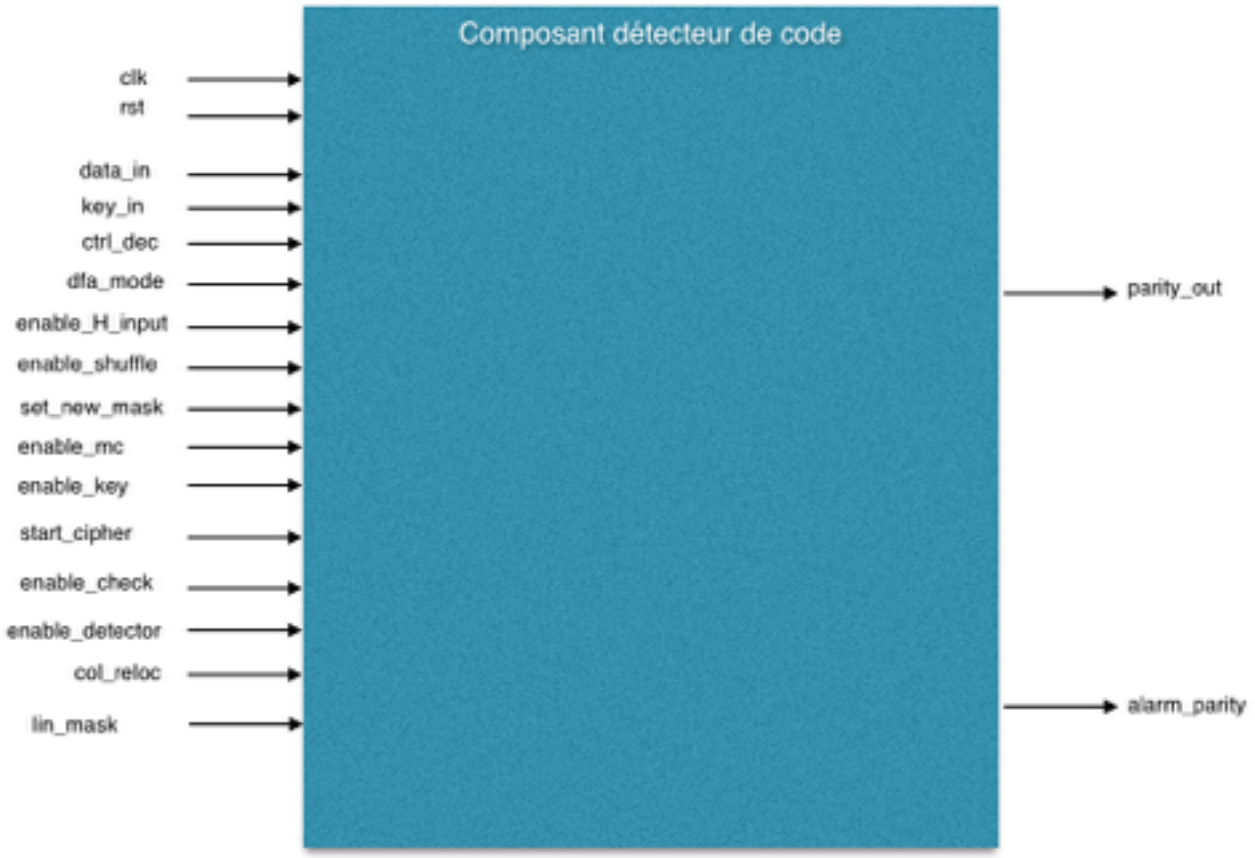

figure 4 - Implantation du composant

Les signaux « clk » et « rst » sont connectés directement à ceux de l'AES. Les autres signaux sont directement connectés au bus de communication et aux signaux de contrôles générés par le contrôleur :

- « data_in » (resp « key_in ») récupèrent les données sur le bus correspondantes à celles en entrée de l'instance de vérification
- « ctrl_dec » indique si l'on chiffre ou l'on déchiffre la donnée
- « dfa_mode » indique si l'on est en mode redondance partielle ou redondance totale. Dans ce projet nous nous intéressons qu'au mode redondance totale.
- « enable_H_inpute » indique le début d'une ronde de chiffrement et qu'il faut prendre en compte les données en entrée
- « enable_shuffle » autorise l'opération « shuffler » tandis que « enable_mc » autorise l'opération « mix_column ».
- « enable_key » indique que la clef va être chargé et « enable_detector » autorise la vérification à l'aide de notre composant
- « lin_mask » est le masque qui va être utilisé pour masquer les données.
- « enable_check » autorise la comparaison qui lèvera l'alarme ou pas.
- « parity_out » et « alarm_parity » sont les 2 sorties du composants qui renvoient le vecteur de bits de parité calculé et une alarme correspondante à la comparaison entre ce vecteur de bits et la parité des données en entrée.

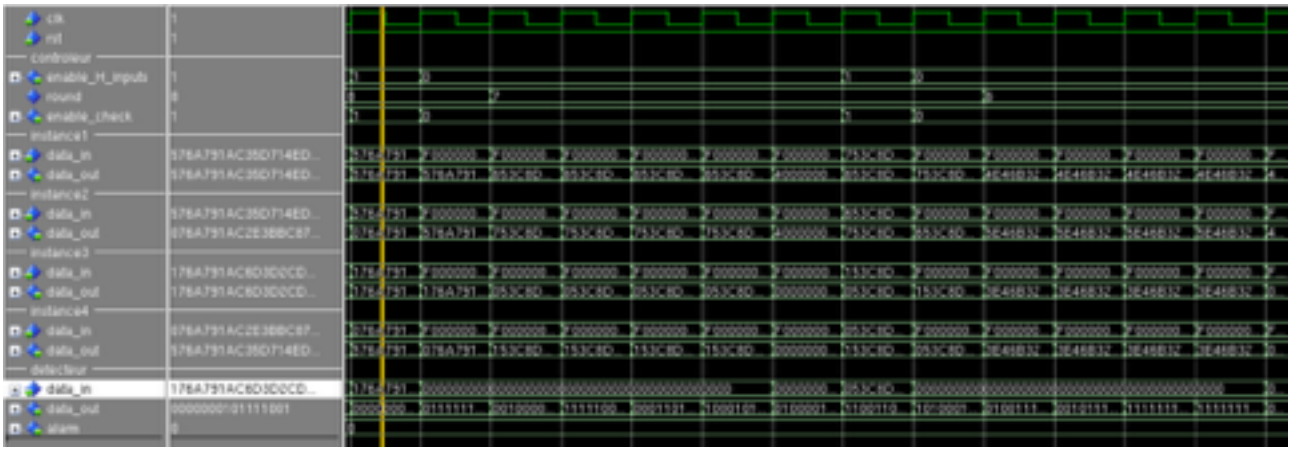

figure 5 - Fonctionnement globale AES

La simulation présente figure 5 nous montre le fonctionnement globale de l'AES. Ici nous avons pris l'exemple d'une ronde de chiffrement (la ronde 7, cf simulation). Nous remarquons que lorsque que l'entrée « enable_H_input » passe à 1 au début de la ronde, le détecteur reçoit en entrée les même donnée que l'instance 3 qui est l'instance de vérification à ce moment là. Ensuite les calculs « subByte », « shuffler » s'effectuent, puis le cycle avant la fin de la ronde (cf quand « enable_H_input » repassera à 1), le détecteur reçoit en entrée les même données en sortie de l'instance 3. Ce sont les sorties de mix_column nécessaire pour la calcul de parité. Nous observons ensuite que le signal « enable_check » passe à 1 et que l'alarme de parité ne s'est pas levée. Il n'y avait donc pas d'erreur. Dans la partie validation, nous verrons un cas où l'alarme du détecteur est levée.

B. Implémentation du système sur carte

Une fois la conception de notre comparateur effectuée nous devions connecter notre crypto-processeur à un microblaze à l'aide d'un bus type PLB.
La première étape fut de faire la synthèse de notre composant en portes logiques. Cette synthèse a été réalisée à l'aide du logiciel ISE qui transforme les sources VHDL hiérarchisée en composant décrit à l'aide de portes logiques. Le résultat de la synthèse pour notre bloc de prévision de parité est le suivant :

Matériel ajouté

Matériel	dénombrement
Compteur	1
Bascule D	47
Additionneur	2
Comparateur	1

La prévision de parité ne rajoute pas énormément de matériel comparé à une ronde qui ne décompte pas moins de 100 bascules D, 20 multiplexeurs …

Logic ajoutée

Logic utililisée	Aes originale	Aes modifiée	pourcentage supplémentaire
Slice Registers	2280	2655	16,45
Slice LUTs	10257	13325	29,91
Slice LUTs FF	1687	1813	7,47

Cependant la logic ajoutée lors de la synthèse de notre composant par rapport à la synthèse originale n'est pas négligeable. En effet nous obtenons 16% de « slice Registers » en plus et 30 % de « slice LUTs ». Si nous ne considérons que les 4 blocs qui gèrent le calcul (ceux qui consomme le pus de logique). Ces instances doivent contenir environ 25% de la logic synthétisée. Si nous nous référons aux résultats obtenus lors de notre synthèse, nous pouvons mettre en doute la « qualité » du code généré. En effet, nous verrons dans la partie « Validation » que notre code respecte les fonctionnalités voulus. Cependant il doit pouvoir être optimisé aux vues du tableau « Logic ajoutée ».
Dans le rapport de synthèse, nous avons un chemin critique de 8,996 ns ce qui équivaut à une fréquence maximale d'utilisation de 111Mhz. Cependant nous verrons plus tard que nous sommes contraints d'utiliser une fréquence de 25Mhz. En effet, le routage est tel que le chemin critique augmente fortement après cette étape.
Ensuite nous avons utilisé le logiciel EDK pour créer notre architecture centrée autour d'un microblaze et d'un bus. Nous avons choisi d'utiliser une mémoire interne au microblaze de 64Ko pour ne pas avoir de contrainte au niveau du développement de l'application. Nous avons par la suite, décidé de connecter sur le bus des « dip-switch », des « boutons poussoirs », et un « uart » pour le debug et la démonstration que nous verrons par la suite.
Une fois l'architecture mise en place nous avons importé notre composant à l'aide de la netlist générée après la synthèse. Cependant, notre composant n'est pas adapté au protocole de communication du bus. Nous avons donc choisi d'insérer 14 registres connectés au bus de communication pour gérer les entrées de notre composant. La logique, entre le bus et le composant, qui affecte les entrées du composant en fonction des demandes du microblaze est décrite dans le fichier « user_logic.vhd ». Les 14 registres (registres 32 bits) remplissent les fonctions suivantes :

- registres 6, 7, 8, 9 sont les registres où sont stockés les entrées du composant AES. En effet, dans notre cas, nous ne considérons que des textes de 128 bits. Ces 128 bits sont envoyés dans les 4 registres décrits précédemment.
- registres 10, 11, 12, 13 contiennent le texte chiffré de 128 bits en sorti du composant AES.
- le registre 1 est le registre de configuration:
  - Le bit 0 active l'envoie de faute si il est à 1 et le désactive sinon.
  - Le bit 1 active le mode redondance totale si il est à 1 et le désactive sinon. Dans le cadre de ce projet nous nous placerons toujours dans ce mode là
  - Le bit 2 active le mode redondance partiel. Comme nous nous plaçons toujours dans le cas de la redondance totale nous ne nous occuperons pas de ce mode.
  - Le bit 3 active le détecteur implémenté s'il est à 1 et le désactive sinon.
- les registres 2, 3, 4 sont des registres connectés à des sorties de l'AES
  - le registre 2 contient « 1 » quand le chiffrement est fini et que les données en sortie sont prêtes
  - le registre 3 contient « 1 » quand le composant AES est prêt à effectuer un chiffrement
  - le registre 4 est connecté au sorties des erreurs. Le bit 0 est à 1 si le chiffrement est fauté et à 0 sinon. Tandis que le bit 1 est à 1 si une fausse alarme a été détectée.
- le registre 0 permet de démarrer un chiffrement. Une fois les données chargées, il suffit d'insérer un 1 dans ce registre pour que le chiffrement commence.

Pour simuler l'injection de fautes au sein de notre système, nous avons de nouveau utiliser un générateur congruentiel pour nous fournir un nombre aléatoire modulo 80 (nombre de cycles d'horloge pour qu'un chiffrement soit effectif) qui indique quand l'entrée du composant activant la faute doit passer à 1. Nous obtenons grâce à ce système, une faute aléatoire au niveau temporel et aléatoire au niveau spatial à l'aide de la sélection aléatoire de l'instance fautée.
L'application embarquée par le microblaze nous permet d'utiliser notre crypto processeur implémenté. L'algorithme utilisé est décrit ci après:

```
Initialisation_uart;
//activation detecteur - mode redondance totale - activation des fautes
ecriture_registre(1, 0x00000008);
//attente que l'AES soit prêt à chiffrer
ready = lire_registre (3);
while (ready != 1)
    ready = lire_registre(3);
end while;

while (true)
    //attente que l'AES soit prêt à chiffrer
    ready = lire_registre (3);
    while (ready != 1)
    ready = lire_registre(3);
    end while;
    //chiffrement de 0x3243f6a8885a308d313198a2e0370734
    ecriture_registre(6, 0xe0370734);
    ecriture_registre(7, 0x313198a2);
    ecriture_registre(8, 0x885a308d);
    ecriture_registre(9, 0x3243f6a8);
    //attente fin de chiffrement
    data_ready = lire_registre(2);
    while (data_ready != 1)
    data_ready = lire_registre(2);
    end while;
    //reception des données de fin de chiffrement
    result1 = lire_registre(10);
    result2 = lire_registre(11);
    result3 = lire_registre(12);
    result4 = lire_registre(13);
    errors = lire_registre(4);
    if (errors == 1)
        print("chiffrement fauté");
        cpt_cipher_faulted++;
    else if (errors == 2)
        print("fausse alarme");
        cpt_fausse_alarme ++;
    else if (results != resultat_attendu)
        print("problème détection alarme");
    end if;

    cpt_cipher++;
    if ((cpt_cipher % 1000000) == 0 && cpt_cipher != 0) {
            print("nb_chiffrement : %d  \r\n", cpt_cipher);
            print("chiffrement fauté : %d ", cpt_cipher_faulted * 100 / cpt_cipher);print("%\r\n");
            print("fausse alarme : %d ", cpt_fausse_alarme * 100 / cpt_cipher);print("%\r\n");
    }

end while;
```

Algorithme application embarquée

Nous faisons des chiffrements de manière continue et nous observons sur un terminal si un chiffrement est fauté ou si une fausse alarme est détecté. Tous les 100 000 chiffrement, nous affichons le nombre de chiffrement, le pourcentage de chiffrement fauté et le pourcentage de fausses alarmes détectées.
Une fois la logique, qui gère les entrée et sorties de notre composant, implantée nous avons généré le bitstream pour programmer le FPGA. Cependant nous avions des problèmes de violation du chemin critique. En effet, l'horloge de notre architecture était branchée à celle du bus qui est de 125 MHz. Cependant, le chemin critique était d'environ 39ns . Nous avons donc généré une horloge spécifique à notre composant de 25 MHz et obtenu le bitstream pour programmer la carte FPGA.

III. Validation

Afin de mener à bien un tel projet nous nous devions d'avancer de manière incrémentale et de tester le fonctionnement du système à chaque étape du projet.

A. Validation de la conception

Après l'étude du fonctionnement de la prévision de parité, nous nous sommes rendus compte que l'étape critique était l'implantation du calcul de parité de l'opération « Mix column ». Nous avons donc développer et tester cette opération en amont du développement de la prévision de parité.

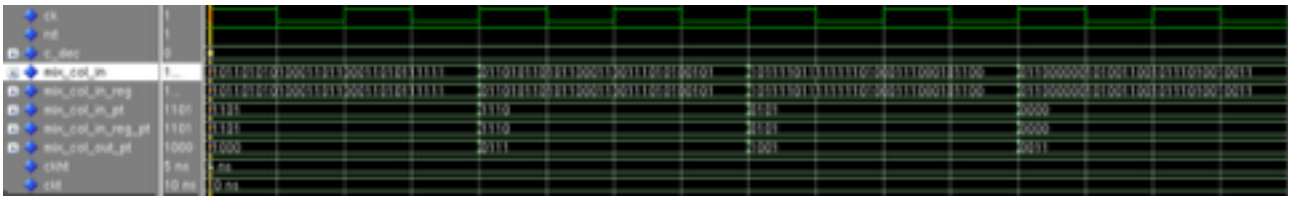

figure 6 - Simulation test du calcul de parité de « mix

Dans la simulation figure 6 nous remarquons que pour les 4 entrées de 4 octets, nous obtenons le bon calcul de parité pour l'opération « mix column » attendue en sortie. En effet, en comparant la parité de la sortie de « mix column » dans l'aes originale, nous avons pu valider nos parités calculées.
Une fois la fonctionnalité testée, nous l'avons incorporé dans notre bloc détecteur pour tester le fonctionnement global. En premier lieu, le but était de vérifier que la parité calculée était la même que la parité des données en sortie de l'instance de vérification. A l'aide des simulations réalisées avec l'outil modelsim, nous pouvions remonter jusqu'aux différents problèmes d'implémentation qui empêchait le bon fonction de notre système. Une fois que la fonctionnalité semblait marcher pour un chiffrement nous l'avons testé sur plusieurs chiffrement sans insérer de faute.
Dès que le fonctionnement du calcul de parité fut validé, nous devions valider le fonctionnement de détection de fautes. Nous avons donc injecté un bit flip à l'entrée de l'opération « shuffler » dans la ronde de vérification à plusieurs reprises et dans plusieurs chiffrement et à des instants différents.

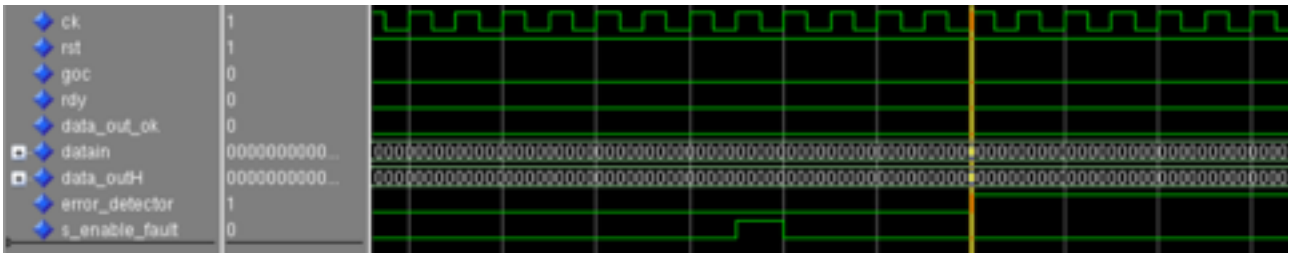

figure 7 - Simulation test détection d'erreur de parité

Dans la simulation représentée par la figure 7, nous observons l'activation de l'alarme de détection de la parité après l'injection d'une faute.
Nous nous sommes rendus compte que certaines fautes étaient silencieuses et que certaines n'étaient pas détectées par la parité, alors que l'alarme de comparaison entre l'instance de chiffrement et vérification était levée. Les fautes silencieuses étaient dues au fait que nous

injections la faute à l'entrée de l'opération « shuffler » qui contient, comme expliqué précédemment, des données sensibles au chiffrement pendant 4 cycles d'horloge sur 8 cycles au total. Les fautes non détectées étaient celles qui provoquaient une erreur double au niveau de la parité.
Dès lors, nous avons estimé le nombre de chiffrement fauté que nous aurions sur un grand nombre de calcul. Nous avons 50% de chance de fauter l'instance de vérification ou l'instance de chiffrement et donc de lever l'alarme de comparaison. Ensuite nous avons 50% de chance de créer une faute non silencieuse. Ce qui fait 25% de chance de lever l'alarme de comparaison. Nous avons environ 12,5% de chance d'injecter une faute non silencieuse dans l'instance de chiffrement.
Nous avons donc développé un banc de test effectuant 10 000 chiffrements à la suite ou dans chaque chiffrement nous injections une faute. Ce test écrit dans un fichier .txt le nombre de chiffrement, le nombre de chiffrement fautés, et le nombre de fausses alarmes. A l'aide d'un script nous avons analysé les résultats de ce fichier .txt et nous obtenons le résultat suivant :

Nombre de chiffrement	10 000
Pourcentage d'alarmes levées	22 %
Pourcentage de fausses alarmes détectées	5 %

En l'absence du détecteur de parité, nous obtenons 27 % de détection de fautes. Ce qui est normal d'après les estimations et le fait que nous utilisons un générateur « pseudo aléatoire » de nombre pour choisir l'instant et le lieu d'injection des fautes.
Nous obtenons, cependant, 5 % de détection de faute dans l'instance de chiffrement. Cette donnée est plutôt faible et démontre que la vérification à l'aide de la parité n'est pas très fiable. Cependant 5 % de détection de fausses alarmes est un pourcentage non négligeable et démontre le fonctionnement de notre détecteur.
Une fois que le détecteur fut validé et synthétisé nous avons du repasser ce test qui était notamment plus long une fois la synthèse effectuée. Cependant nous obtenions les mêmes résultats et nous pouvions passer à l'implémentation de l'architecture sur carte programmable.

Avant l'intégration sur carte, nous avons effectué des simulations dites structurelles pour valider le bon fonctionnement de la connexion entre le bus de communication et de notre composant.

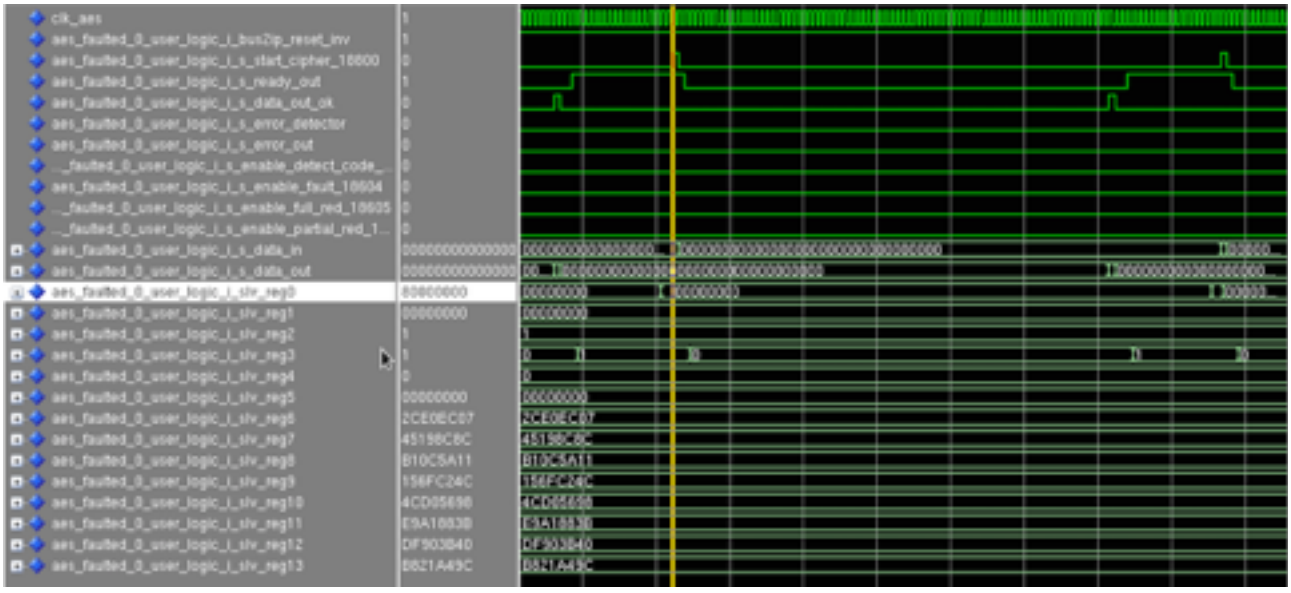

figure 8 - Simulation du user_logic

En effectuant différentes simulations et en injectant différentes valeurs dans les registres (slv_regi) nous avons validé le fonctionnement du « user_logic » en observant le bon fonctionnement du chiffrement au sein de notre AES.
Ensuite nous avons téléchargé le bitstream généré à l'aide de EDK sur la carte FPGA où nous avons connecté l'UART à un pc dans l'optique d'afficher les résultats sur le terminal d'un ordinateur. Nous nous sommes basés sur le test que nous avions effectué en amont du flow de conception, à savoir, de tester sur 10 000 chiffrement le pourcentage d'alarmes levées et le pourcentage de fausses alarmes détectées. Nous obtenons les mêmes chiffres que cités précédemment. Ceci est normal car nous utilisions le mêmes générateur de nombre aléatoire.

IV. Manuel utilisateur et démonstration

A. Archive du projet

L'archive du projet est partagée en 2 dossiers. Le dossier « AES_Moprh_pt » contient l'architecture développée sans l'injection de fautes. Dans ce dossier est présent 3 sous dossiers :

- « vhd » où les sources vhdl du composant sont présentes
- « sim » où les sources vhdl du banc de test sont présentes. Ce banc de test effectue 10 000 chiffrements
- « libs » où un script de compilation est présent. Pour exécuter ce script il faut se placer dans cd répertoire. Il créera les libraries du composant de du banc de test.

Le deuxième dossier « AES_Morph_faulted » contient l'architecture décrites tout au long de ce rapport, à savoir, la même que celle décrites dans le dossier « Aes_morph_pt » sauf que l'on a rajouté l'injection de faute. Les sous répertoire composant ce dossier son les suivants :

- « vhd » où les sources vhdl du composant sont présentes
- « sim » où les sources vhdl du banc de test est présents. Ce banc de test effectue 10 000 chiffrements
- « libs » où un script de compilation est présent. Pour exécuter ce script il faut se placer dans cd répertoire. Il créera les libraries du composant de du banc de test.
- « edk » où est présent l'application embarquée dans le microblaze
- « synth » où sont présent tous les projet de la génération de netlist avec ise et du bitstream avec edk

Les libraries, la netlist et le bitstream ont déjà été générés. la netlist est présente dans « ./ Aes_morph_faulted/synth/Aes_faulted/aes_core.ngc ». Le bitstream quant à lui se trouve dans « ./ Aes_morph_faulted/synth/Aes_faulted/edk/implementation/download.bit » . Il suffit de brancher une carte FPGA à un pc, d'ouvrir le logiciel « ise_impact » et télécharger le logiciel sur la carte.
NB : La synthèse et le bitsream ont été conçus pour une carte FPGA virtex5 ML507.

Si l'on souhaite modifier l'architecture autour du bus PLB avec xps, la sauvegarde du projet est présente dans « ./Aes_morph_faulted/synth/Aes_faulted/edk/system.xps »

B. Démonstration cliente

Concernant la démonstration de notre produit fini, nous avons connecté les « boutons poussoirs » et « dip-switch » de la carte FPGA afin de pouvoir injecter des fautes et configurer de manière manuelle notre composant AES. En effet à l'aide des « dip-switch », nous choisissons d'activer ou non le détecteur parité. De plus nous pouvons nous placer dans 2 modes différents :

- un mode où chaque chiffrement est fauté de manière aléatoire. Tous les 100 000 chiffrements nous affichons le pourcentage de chiffrements corrompus et le pourcentage de fausses alarmes détectées.
- un mode où nous injectons des fautes à l'aide des boutons poussoirs. Lors d'un appui sur un bouton poussoir, une faute est injectée sur 1 cycle d'horloge à l'entrée de l'opération « shuffler » dans une instance choisie aléatoirement. Nous affichons alors, à l'aide de l'uart, si le chiffrement est fauté ou si une fausse alarme est détectée.

C. Evolution du planning et répartition des taches

Le planning prévisionnel défini au début du projet est annexé en page 15 tandis que le planning effectué est annexé page 16.
Nous remarquons que nous avions relativement bien défini les tâches à faire au début du projet. Cependant nous avions mal jaugé les heures de travail à effectuer pour certaines tâches, notamment lors de la conception du détecteur de parité. En effet, la compréhension du fonctionnement de l'architecture originale fut bloquante. Il fallait comprendre la synchronisation du bus de communication, de l'allocation des instances de chiffrement et de vérification etc. Ceci fut plus compliqué que prévu. Il en est de même pour le calcul de parité, notamment pour la prévision de l'opération « mix column ». Forte heureusement, nous avions prévu une semaine de flottement pour éventuellement prendre de l'avance ou rattraper notre retard. De plus, le fait d'avoir utilisé les logiciels « edk » et « ise » lors des TPs de « sécurité des systèmes embarqués » nous a fait gagné du temps lors de l'intégration sur carte.

Conclusion :

Ce projet fut très formateur autant sur le plan technique que sur le plan organisationnel. En effet, nous nous avons perfectionné nos connaissances en VHDL et nous avons utilisé des technologies que nous n'avions jamais approfondi, notamment au niveau de la programmation de carte FPGA. De plus il fut très intéressant de devoir ce projeter sur un projet conséquent et de s'organiser pour le mener à bien. Nous pensions qu'il n'aurait pas été possible, pour notre binôme, de finir dans les temps si nous n'avions pas pris le temps de discuter des tâches à faire.

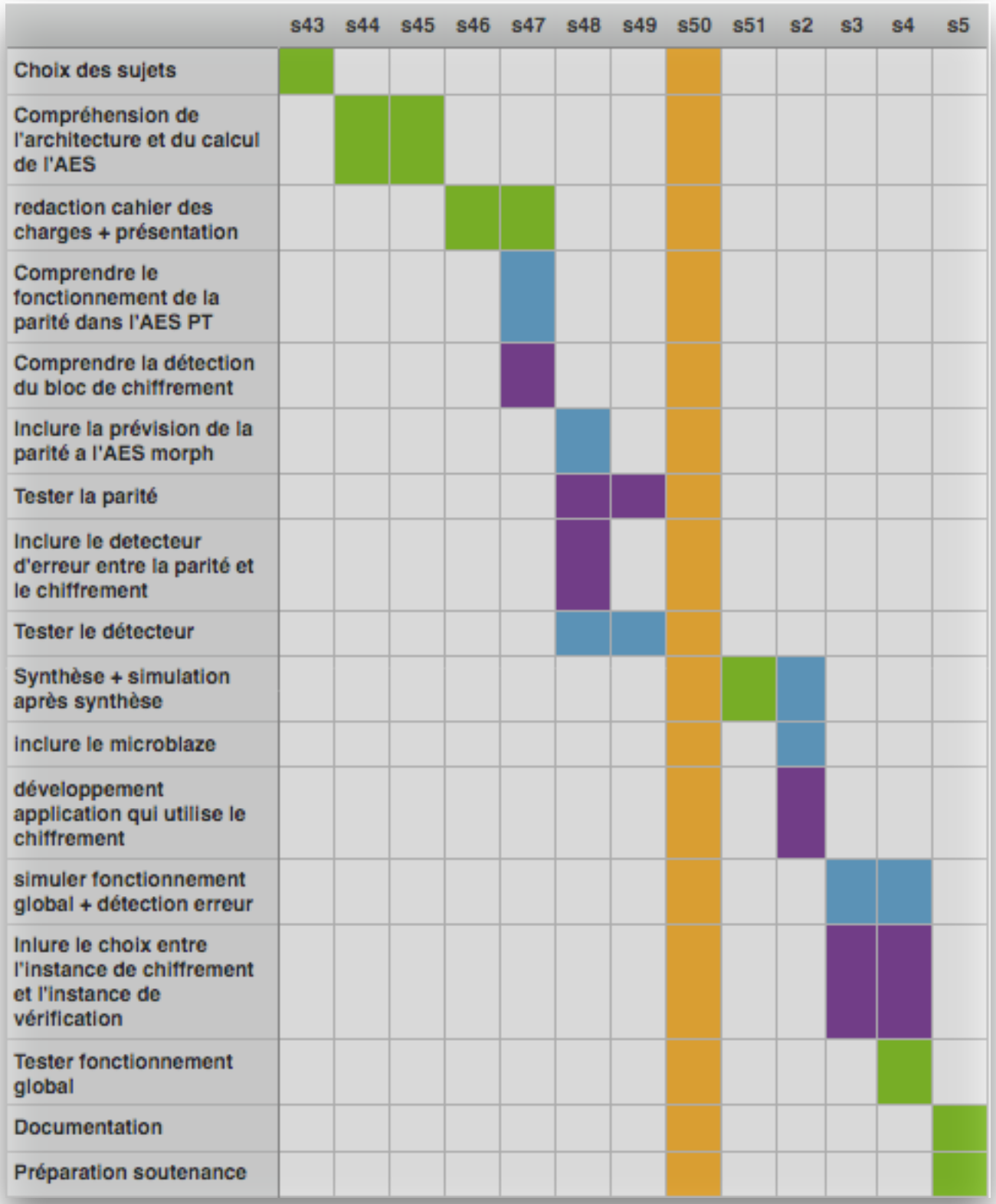

figure 10 - Planning prévisionnel

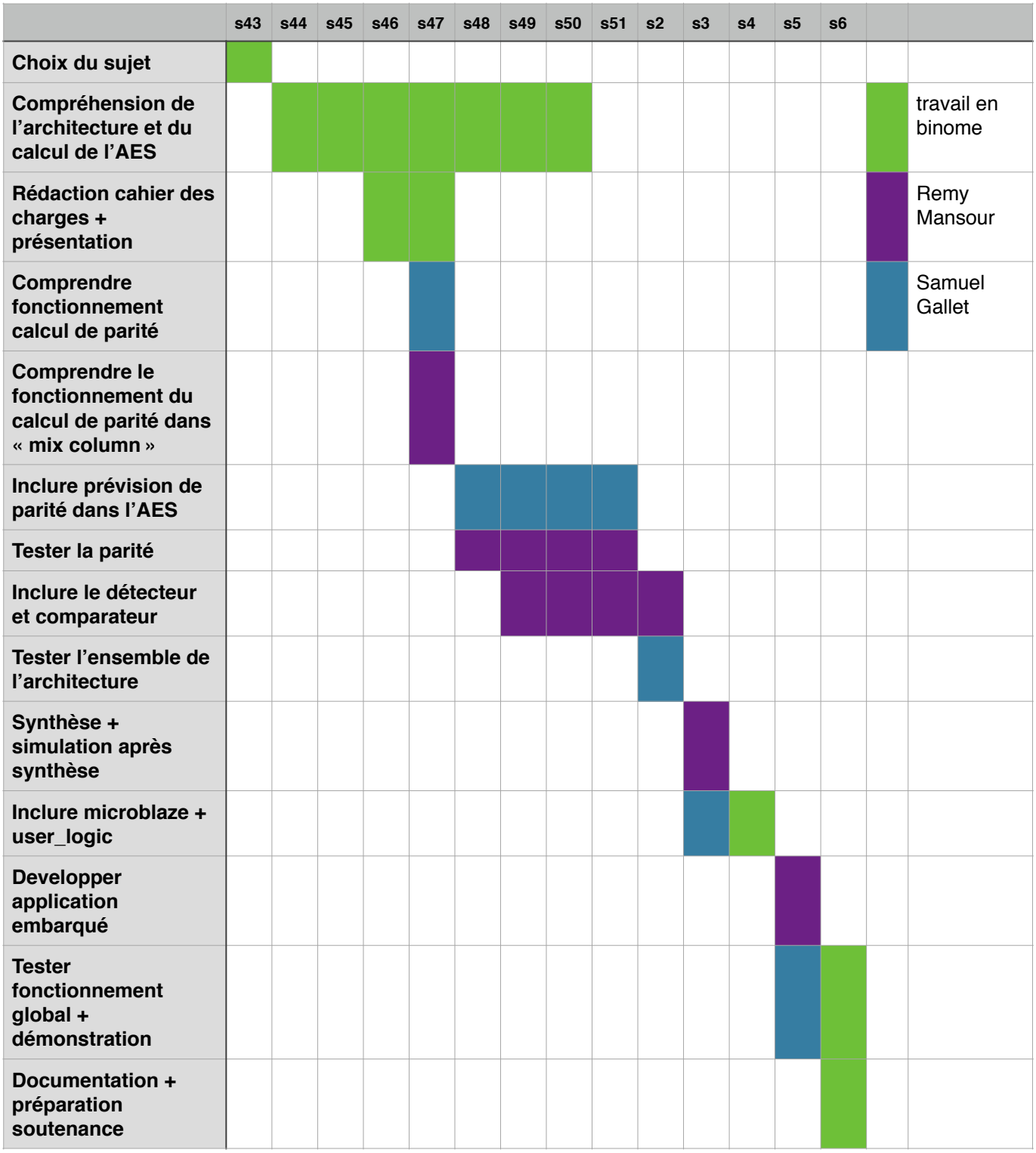

figure 9 - Planning réalisé