PJ
Protocole de communication

Jérémy Cheynet
Yann Sionneau

Année 2010

Table des matières

I. Théorie 4

1. Objectifs 5

2. Spécifictions 6
2.1. Matériel . . . 6
2.1.1. Microcontrôleur . . . 6
2.1.2. FPGA . . . 7
2.2. Le BUS . . . 7
2.3. Structure des couches d'abstractions . . . 7
2.3.1. La couche physique . . . 7
2.3.1.1. Etat logique par défaut . . . 7
2.3.1.2. Définition des bits . . . 7
2.3.1.3. Trame de données . . . 8
2.3.2. La couche de transport . . . 9
2.3.2.1. L'adressage . . . 9
2.3.2.2. La vérification . . . 9
2.3.2.3. La forme d'un paquet . . . 9
2.3.3. La couche applicative . . . 9

II. Implémentation 10

3. Microcontrôleur 11
3.1. La couche physique : phy.c et phy.h . . . 11
3.1.1. Réception d'un octet . . . 11
3.1.1.1. Interruption sur front montant de la pin PD5 . . . 11
3.1.1.2. Interruption du timer1 . . . 11
3.1.2. Emission d'un octet . . . 12
3.1.2.1. Envoi d'un bit logique "haut" . . . 12
3.1.2.2. Envoi d'un bit logique "bas" . . . 12
3.1.3. Calcul de la parité . . . 12
3.2. La couche d'adressage : mac.c et mac.h . . . 13
3.2.1. Réception d'un paquet . . . 13
3.2.1.1. Réception d'un octet de la couche physique . . . 13
3.2.1.2. Le buffer circulaire . . . 14
3.2.1.3. Lecture d'un paquet depuis la couche applicative . . . 14
3.2.2. Emission d'un paquet . . . 15
3.2.3. Fonctions globales . . . 15
3.2.3.1. Calcul du checksum . . . 15
3.2.3.2. Gestion des erreurs . . . 15

3.3. Le débogage . . . 15
3.3.1. La liaison série . . . 15
3.3.1.1. uart_init . . . 16
3.3.1.2. uart_send_char . . . 16
3.3.1.3. puts . . . 16
3.3.1.4. uart_recv_char . . . 16
3.3.2. Les programmes de tests . . . 16
3.3.2.1. Sender . . . 16
3.3.2.2. Receiver . . . 16

4. FPGA **18**

III. Résultat, problème et analyse **19**

5. Les problèmes qui sont apparus **20**
5.1. Le compilateur . . . 20
5.2. La désynchronisation . . . 21
5.2.1. Problème de lecture d'un octet . . . 21
5.2.2. Problème de lecture d'un paquet . . . 21
5.3. Le datasheet et la liaison série . . . 21

6. Les résultats **22**
6.1. Emission et réception simultanées . . . 22
6.2. La couche haute . . . 22

IV. Nos impressions **23**

7. Jérémy Cheynet **24**
7.1. Le langage C version microcontrôleur . . . 24
7.2. Git . . . 25
7.3. Résolution des problèmes . . . 25
7.4. Conclusion . . . 25

8. Yann Sionneau **26**
8.1. Le C pour microcontrôleur . . . 26
8.2. Git . . . 26
8.3. FPGA . . . 27
8.4. Conclusion, mon sentiment . . . 27

Première partie .

Théorie

1. Objectifs

L'objectif de notre projet est de créer un système qui permet de faire communiquer toutes sortes d'électronique embarquée entre elles. Une des conditions que nous nous sommes fixée, est de faire communiquer les appareils sur un seul et unique fil pour pouvoir, dans un second temps, faire communiquer ces appareils avec une liaison sans fils.

Les systèmes embarqués que nous comptons utiliser sont des microcontrôleurs et des FPGA [1].

Afin de faciliter la communication entre les appareils et l'architecture de notre système, nous avons choisi d'utiliser une structure sous forme de couches d'abstractions (en s'inspirant du model OSI).

1. *FPGA* (field-programmable gate array, réseau matriciel de portes logiques programmables).

2. Spécifictions

2.1. Matériel

Notre choix est de faire fonctionner notre protocole sur plusieurs systèmes et différentes architectures. Pour cela, nous avons décidé d'implémenter notre protocole sur un microctontrôleur et sur un FPGA. Nous avons donc choisi, parmi la grande gamme de produits disponibles dans ces deux catégories, une architecture de microcontrôleur et un type de FPGA.

2.1.1. Microcontrôleur

Nous avons choisi d'implémenter notre protocole sur microcontrôleur car ces puces électroniques sont très répandues, très utilisées et faciles d'utilisation. En effet, utiliser un microprocesseur oblige de rajouter de la mémoire, et des modules externe, tandis qu'un microcontrôleur est autonome.

Le microcontrôleur que nous utilisons fait partie la gamme ATMega (architecture AVR) de chez ATMEL. Ce type de microcontrôleur est facilement programmable en C/C++, et se trouve pour un prix correct.

Voici les 3 types de microcontrôleurs que nous allons utiliser :

La carte arduino : Il s'agit d'une carte microcontrôleur toute prête, programmable en C/C++ avec un logiciel fourni gratuitement. Tous les programmes et tous les shields [1] sont open-source. Le microcontrôleur qui se trouve sur la carte est un ATMEGA328p de chez ATMEL.

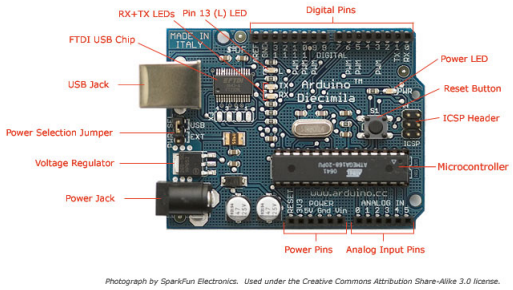


FIGURE 2.1.: Carte Arduino

L'ATMEGA324 : Il s'agit d'un microcontrôleur 40 broches de 32 entrées/sorties de chez ATMEL. La carte que nous utilisons pour l'utiliser est une carte faite maison, et programmée grâce au logiciel libre avrdude à l'aide d'un ISP programmer (système libre pour programmer la série avr de chez ATMEL). Le programme s'effectue toujours en C/C++ et se compile grâce à la toolchain GNU pour l'architecture AVR (avr-gcc, avr-objdump, avr-ld, avr-as etc ...).

L'ATTINY13 : Microcontrôleur à prix réduit de chez ATMEL. Il s'agit d'un petit microcontrôleur de 8 broches, programmable comme l'ATMEGA324. Là aussi, nous allons designer une carte pour pouvoir faire des tests de notre système.

1. Petite carte électronique que l'on peut facilement brancher sur la carte arduino

2.1.2. FPGA

Nous avons choisi d'utiliser pour nos tests un Spartan-3A 400k portes (XC3S400A) de chez Xilinx. Cette puce est présente sur la plaque de développement "AVnet Spartan-3A Evaluation Kit" en notre possession. Un port de 40 pins d'entrées/sorties (GPIOs) est disponible sur cette plaque que nous pourront utiliser pour tester le protocole, en la reliant par exemple à un Arduino. Le bitstream est synthétisé en utilisant les outils Xilinx ISE Webpack (Xst), le code source est en langage Verilog et versionné sur github. Une simulation est faite en utilisant le logiciel icarus verilog, les résultats de cette simulation sont analysés en utilisant gtkwave qui génère les chronogrammes des signaux importants.

FIGURE 2.2.: Plaque de développement "AVnet Spartan-3A Evaluation Kit"

2.2. Le BUS

Notre système fonctionne sur 1 seul et unique fil. Les appareils communiquent entre-eux en envoyant des signaux logiques sur ce fil.

Afin de ne pas avoir de conflits entre les appareils qui pourront communiquer sur ce fil, nous avons choisi :

Topologie du bus : 1 maitre, plusieurs esclaves.

Protocole de communication : Questions / Réponses

2.3. Structure des couches d'abstractions

2.3.1. La couche physique

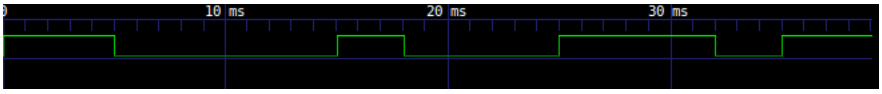
2.3.1.1. Etat logique par défaut

Par défaut, la ligne sera à un état logique haut (5V). Lorsqu'un des appareils voudra communiquer avec un autre, il devra créer un front montant pour commencer la communication (donc, passer par un état bas avant de repasser par un état haut). Ainsi, avant de communiquer, tout appareil n'ayant pas reçu d'interruption pendant un certain temps, verifiera que la "ligne" est à un état logique haut. Si ce n'est pas le cas, il attendra avant d'émettre son signal.

2.3.1.2. Définition des bits

Un bit est un quantum de temps de 10ms (que nous pourrons diminuer plus tard). Il existe 4 types de bit :

- Le bit de start
- Le bit d'un état logique bas
- Le bit d'un état logique haut
- Le bit de stop



De gauche à droite : Le bit de start (10ms à l'état bas), suivi du bit représentant l'état logique bas, puis celui de l'état logique haut, et enfin, le bit de stop (front montant restant à l'état haut).

FIGURE 2.3.: Les 4 types de bits possible

Un des objectifs est de ne pas avoir de désynchronisation entres les 2 appareils qui communiquent entre eux.

En effet, supposons que tous les bits fassent 10ms. Une trame normal fait 11 bits (1octet + 1 bit de parité + 1 bit de start + 1 bit de stop). Le récepteur va, à partir du premier bit reçu déclencher un timer, et regarder régulière le signal pour voir à quel état se trouve celui-ci. Cependant, si les 2 systèmes n'ont pas exactement la même fréquence, il y aura une désynchronisation des 2 appareils et une perte de bit.

Afin d'éviter cela, nous avons décidé de resynchroniser nos appareils à chaque bit. Tous les bits (état logique haut ou bas) commencent par un front montant. A partir de ce moment, le récepteur lance un timer. Au bout de 5ms (50% du temps du bit), il va regarder l'état du signal (haut ou bas) et en déduire l'état du bit. Mais au lieu de faire continuer son timer, il va le stopper, et le relancer lors de l'arriver du prochain bit par un front montant.

Pour cela, un état logique haut sera représenté par un état haut de 2/3 du temps total (soit 7ms) puis d'un état bas pendant 1/3 du temps (3ms). Il en va de même pour l'état logique bas, représenté par un état haut de 1/3 du temps et les 2/3 restant par un état bas. Ainsi, la durée du bit est de 10ms, tout les bits commencent par un front montant et finissent pas un état bas (permettant de faire un nouveau front montant).

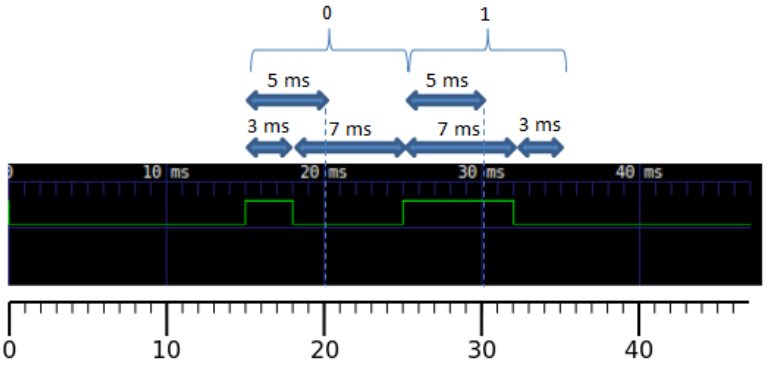


FIGURE 2.4.: Représentation des état haut et bas

2.3.1.3. Trame de données

La trame de données est constituée par une trame de 8 bits contenant un octet de données, suivi par 1 bit de parité. Cela permet d'avoir une première vérification du signal sur la couche physique. En cas d'erreur, celle-ci est transmise à la couche supérieur, qui traitera l'erreur.

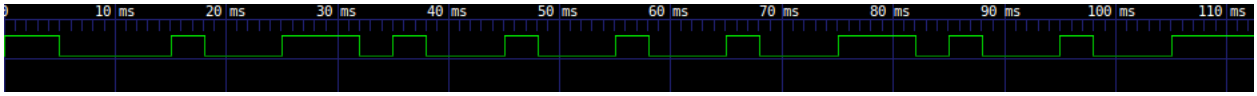


FIGURE 2.5.: Exemple d'une trame d'un octet (0x42)

2.3.2. La couche de transport

La couche de transport nous permet d'avoir un système d'adresse et de vérification du signal.

2.3.2.1. L'adressage

L'adressage se fait sur un octet. Lors de la communication entre 2 appareils, le premier transferts correspondra à l'adresse de l'appareil source. Ensuite, nous transmettrons l'adresse de destination.

Cela permet de pouvoir s'adresser à un seul et unique appareil.

2.3.2.2. La vérification

Nous avons un octet qui est réservé pour pouvoir transférer des données de vérification ainsi que la taille du paquet.

Les 4 premiers bits de cet octet servent à définir la taille de notre paquet, pour pouvoir dire au récepteur combien d'octets de données utiles sont transférés.

Les 4 derniers bits servent de checksum, pour s'assurer qu'il n'y a pas d'erreur dans le transfert des données dans le paquet.

2.3.2.3. La forme d'un paquet

Après ces 3 octets, nous transférons un certain nombre de paquets de données utiles, qui dépendra de ce que l'émetteur veut envoyer.

Voici la structure d'un paquet :

1. Un octet contenant l'adresse source
2. Un octet contenant l'adresse de destination
3. Un octet de vérification :
   - 4 bits indiquant le nombre d'octet de donnée utile à transferer
   - 4 bits de checksum sur le paquet total pour éviter d'avoir des erreurs.
4. Les octets contenant les données utiles (payload).

Adresse Source	Adresse Destination	Taille du paquet	Checksum	Octet(s) de donnée(s)

FIGURE 2.6.: Structure d'un paquet

2.3.3. La couche applicative

La couche supérieur est la couche applicative. C'est celle qui appellera la fonction d'envoie pour l'émetteur, et qui sera appelée lorsqu'une interruption arrive, signifiant l'arrivée d'un paquet, déjà traité par la couche 2.

Deuxième partie .

Implémentation

3. Microcontrôleur

3.1. La couche physique : phy.c et phy.h

Nous allons commencer par essayer de comprendre comment fonctionne la couche la plus bas niveau : la couche physique.

Anfin de savoir comment fonction cette couche, nous avons l'algorithme 3.1 qui nous indique la liste des fonctions disponibles dans ces fichiers. De plus, nous allons étudier le fonctionnement des fonctions les plus importantes. Pour de plus amples détails, il est conseillé de se reporter au code source disponible en annexe qui est commenté.

Algorithm 3.1 Prototypes des fonctions de phy.c

```
void relancerTimer(uint16_t valeur);

void initTimer(void);

void emissionOctet( uint8_t octet);

void envoieHaut( void );

void envoieBas( void );

void pause(void);

void depart(void);

uint8_t xor( uint8_t octet );

uint8_t sample(void);

void stop_timer();
```

3.1.1. Réception d'un octet

Afin de recevoir un octet, sans avoir de système bloquant, nous utilisons les interruptions du microcontrôleur. La première interruption est l'interruption de front montant qui signale l'arrivé d'un bit transmis. La seconde interruption est celle du timer 1 qui permet de mesure le temps pour pouvoir analyser le bit reçu au bon moment.

3.1.1.1. Interruption sur front montant de la pin PD5

Le vecteur d'interruption de front montant sur PCINT0 *ISR(INT0_vect)* permet de détecter l'arrivée d'un bit, même si le programme principal ne nous donne pas la main. Lors de la gestion de cette interruption, nous allons lancer le timer pour que celui-ci provoque une interruption 1ms après l'arrivée du bit.

3.1.1.2. Interruption du timer1

En cas d'arrivée d'une interruption du au timer 1 (lancé par l'arrivé d'un bit), nous allons vérifier que le temps nécessaire c'est bien écoulé, et enregistrer chaque bit reçu de manière à pouvoir envoyer un octet entier à la couche supérieur.

Réception d'un bit de start : Le première chose que l'on va faire lors de cette routine d'interruption, et de vérifier que nous ayons bien reçu un bit de start. Si nous avons bien reçu un bit de start, nous incrémentons un compteur qui nous indique le numéro du prochain bit que l'on va recevoir et nous prenons le mutex de ligne afin de ne pas émettre tant que nous n'avons pas reçu un octet en entier. Puis nous quittons la routine d'interruption. Dans le cas contraire, nous quittons la routine d'interruption en continuant d'attendre un bit de start.

Réception du bit de stop : Cela peut paraitre bizarre, mais après avoir reçu le bit de start, on va vérifier que nous ne recevons pas un bit de stop. Si nous avons un bit de stop, on libère le mutex de ligne, et on quitte l'interruption après avoir réinitialisé les variables locales. Le fait de vérifier le bit de stop avant les bits de données permet de gagner du temps dans les routines d'interruptions (puisque nous les quittons plus tôt) et donc laisser le temps processeur au programme principal.

Réception des bits de donnée : Enfin, nous allons enregistrer les neufs bits suivants[1]. Cette opération se fait en 2 temps.

Réception des données : Tout d'abord, nous enregistrons les 8 premiers bits reçu puisqu'il s'agit de l'octet que l'on veut recevoir.

Réception du bit de parité : Puis nous enregistrons le neuvième bit qui correspond au bit de parité. C'est à ce moment là que l'on appel la couche supérieur avec soit, l'envoi de l'octet reçu si la parité reçue et calculée sont identique, ou l'appel à une fonction de gestion des erreurs dans le cas contraire.

3.1.2. Emission d'un octet

Pour envoyer 1 octet, nous avons une fonction bloquante (*void emissionOctet(uint8_ t octet)*)qui prend un octet en paramètre, et qui s'occupe de faire appel aux fonctions très bas niveau qui envoie un bit logique "haut" ou un bit logique "bas". Cette fonction fait un appel à la fonction de calcul de parité pour envoyer la parité après les données.
Cette fonction va d'abord vérifier que la ligne est libre (à l'aide du mutex de ligne), puis envoyer un bit de start suivi des données, de la parité et enfin le bit de stop qui remet la ligne à l'état (idle) logique haut (5V).

3.1.2.1. Envoi d'un bit logique "haut"

La fonction *void envoieHaut(void)* se contente d'envoyer un bit logique "haut" en envoyant un état logique haut (5V) pendant 7ms puis un état logique bas (0V) pendant 3ms.

3.1.2.2. Envoi d'un bit logique "bas"

La fonction *void envoieBas(void)* fait la même chose que la fonction précédente, sauf qu'elle envoie un bit logique "bas".

3.1.3. Calcul de la parité

Un fonction de calcul de parité *uint8_ t xor(uint8_ t octet)* prend en paramètre un octet, calcul la valeur de la parité qui lui est associée, et renvoie un octet[2] qui vaut soit 0 soit 1.

1. 8 bits de données et 1 bit de parité
2. il n'est pas possible de définir un bit en langage C avr

3.2. La couche d'adressage : mac.c et mac.h

Maintenant que nous savons comment fonctionne la couche physique, nous allonc voir comment la couche 2 (la couche de transport) traite les données reçues par la couche physique, et prépare les paquets et les stocke avant une utilisation de ces données par la couche applicative.

Comme pour la couche physique, nous avons une liste des prototypes de fonctions existante, mais nous n'étudierons que le système global avec les fonctions les plus importantes. Pour de plus amples informations, se reporter au code source.

Algorithm 3.2 Prototype des fonctions de mac.c

```
void init_mac(void);

void push_byte(unsigned char);

void clear_ring_buffer_overflow(void);

void copy_packet_to_rx_ring(void);

void send( uint8_t address_dest, uint8_t data[16], uint8_t taille);

int8_t recv (uint8_t *src, uint8_t *taille, unsigned char *datas);

uint8_t calcul_checksum( uint8_t data[16], uint8_t taille);

unsigned char rx_buffer_overflow(void);

uint8_t can_write(void);

uint8_t can_read(void);

void inc_write_pointer(void);

void inc_read_pointer(void);

inline void clear_ring_buffer_overflow(void);

void detection_erreur( uint8_t erreur);
```

3.2.1. Réception d'un paquet

Pour que l'application puisse recevoir les données d'un paquet, il faut créer ce paquet et traiter toutes les données reçues. Cela se fait en 3 grandes étapes que nous allons voir.

3.2.1.1. Réception d'un octet de la couche physique

La première étape correspond à la fonction *void push_ byte(unsigned char b)* qui est appelé par la couche physique lorsque celle-ci a reçu un octet. C'est cette fonction qui va constituer le paquet. Pour ce faire, nous utilisons un compteur qui permet de connaitre le numéro de l'octet reçu. C'est à partir de ce compteur que tout se fait, comme suit :

Est-ce le dernier octet : On va d'abord vérifier que le numéro de l'octet reçu vaut soit la taille du paquet que l'on doit recevoir, soit la taille maximale du paquet. Si c'est le cas, on va stocker le paquet local[3] que l'on a créé dans le buffer circulaire partagé entre la couche 2 et la couche applicative. Enfin, on va réinitialiser toutes les variables locales et le compteur.

Octet de data : Si l'octet que l'on reçoit n'est pas le dernier octet, on va constituer le début de notre paquet.

3. Nous avons définit une structure paquet

Avons nous un octet de start : On va d'abord vérifier que l'on a un octet de start [4] qui est aussi l'octet correspondant à l'adresse source. Si ce n'est pas le cas, on retourne et on attend un octet de start. Si c'est bien le bon octet, on le stocke dans notre structure paquet, dans la partie source, puis on incrémente notre compteur pour signaler que l'on attend un octet correspondant à l'adresse de destination.

Adresse de destination : L'octet suivant est enregistré directement sans traitement. Il s'agit de l'adresse de destination.

Checksum et taille : L'octet suivant est le checksum du paquet et la taille des données. Le checksum est enregistré, tandis que la taille sera utilisée pour savoir combien d'octets de données nous allons enregistrer pour finir de créer notre paquet.

Data : La suite correspond aux données utiles (payload). On va utiliser la taille du paquet reçu précédemment afin de savoir combien d'octets enregistrer.

3.2.1.2. Le buffer circulaire

Lorsque un paquet est entièrement reçu par la fonction que nous avons vu précédement, le paquet est envoyé dans le buffer circulaire grâce à la fonction *void copy_ packet_ to_ rx_ ring(void).*

Cette fonction va, avant de mettre le paquet dans le buffer, vérifier quelques paramètres :

- Vérifier que ce paquet nous est bien destiné en comparant l'adresse de destination et l'adresse du programme.
- Vérifier que le paquet n'est pas erroné en calculant le checksum du paquet, et en le comparant avec celui contenu dans le paquet.
- Vérifier que le buffer n'est pas plein pour ne pas effacer des paquets que l'application n'aurait pas encore lu.

Si une de ces conditions n'est pas vérifiée, le paquet est supprimé et la fonction de gestion des erreurs est appelées pour signaler à l'utilisateur qu'une erreur s'est produite et qu'un paquet a été supprimé.

En revanche, si tout s'est bien passé, on stocke le paquet dans le buffer.

Afin d'éviter de supprimer des paquets que l'application n'aurait pas lu, et de savoir où lire et où écrire, nous avons des pointeurs de lecture et d'écriture sur ce ring buffer. Une comparaison de ces 2 pointeurs, en fonction de l'overflow (qui indique si nous avons fait un tour complet ou non du buffer), nous permet de savoir où écrire et où lire.

3.2.1.3. Lecture d'un paquet depuis la couche applicative

La fonction *uint8_ t recv (uint8_ t *src, uint8_ t *taille, unsigned char *datas)* peut-être appelée par la couche applicative pour lire un paquet reçu depuis le buffer circulaire de réception.

Cette fonction est une fonction bloquante [5]. En revanche, en cas d'erreur détectée (checksum, parité, ...), on débloque la fonction.

Cette fonction renvoie un code retour positif si tout s'est bien passé, et un code retour négatif en cas d'erreur de détection. Ce code retour permet d'éviter de croire que tout s'est bien déroulé, mais avoir retourné des pointeurs vides, ce qui peut être très problématique pour la suite du programme.

4. L'octet de start permet d'éviter une désynchronisation de paquet

5. Nous avons fait le choix de faire une fonction bloquante pour éviter de se retrouver dans la situation suivante : Je lis un paquet avec une fonction non bloquante, et je passe à la suite du programme. J'utilise mon paquet dans la suite du programme. Cependant, ma fonction non bloquante n'a pas eu le temps de récupérer le paquet. Donc le reste du programme s'exécutera avec de fausse valeur.

3.2.2. Emission d'un paquet

Lorsque la couche applicative veut envoyer un paquet, il lui suffit d'appeler la fonction *void send(uint8_ t address_ dest, uint8_ t data[16], uint8_ t taille)* en lui indiquant à qui envoyer le paquet, sa taille et les données.

Cette fonction rempliera automatiquement l'adresse source et le checksum.

3.2.3. Fonctions globales

3.2.3.1. Calcul du checksum

La fonction *uint8_ t calcul_ checksum(uint8_ t data[16], uint8_ t taille)* calcul le checksum et renvoie la valeur du checksum. Elle est utilisée lors de l'envoie et lors de la réception d'un paquet.

A l'heure actuelle, cette fonction n'est pas optimisée pour détecter ou corriger les erreurs. En effet, il est possible que 2 erreurs se compensent. Cependant, elle fonctionne très bien pour détecter une erreur sur plusieurs bits en fonction de leurs positions dans les données.

3.2.3.2. Gestion des erreurs

Une fonction de gestion des erreurs appelée *void detection_ erreur(uint8_ t erreur)* a été implémentée. Elle est appelée en cas d'erreur quelconque lors de la réception de données. Les erreurs peuvent être une erreur dans la parité, dans le checksum ou dans la taille du buffer qui peut être rempli. Dans ce cas, en fonction du type d'erreur, elle affichera un message différent, et débloquera la fonction *recv()* avec un code retour d'erreur.

3.3. Le débogage

Afin de déboguer notre programme, nous avons mis au point des petits programmes de tests qui utilisent la liaison série. Ces programmes envoient ou reçoivent des octets, et les renvoient sur la liaison série pour pouvoir les afficher sur un ordinateur.

3.3.1. La liaison série

En plus du port série du microcontrôleur ATmega328p, la carte arduino (que nous avons utilisé pour le développement de notre programme) possède un FTDI. Celui-ci permet de faire la conversion du signal série TTL en signal série RS232 puis de l'envoyer ensuite sur de l'USB en étant reconnu par GNU/Linux et Windows comme une console série (over USB).

Nous avons créé les fonctions qui nous permettent d'émettre sur le port série, afin de voir la progression de notre programme.

Les fonctions que nous avons faites (dont les codes sources se trouvent en annexe) permettent d'initialiser la liaison série, de recevoir ou émettre un caractère, et d'envoyer une chaine de caractères.

Algorithm 3.3 prototype des fonctions utilisant la liaison série

```
void uart_init(void);

void puts(const char *);

unsigned char uart_recv_char(void);

void uart_send_char(unsigned char);
```

3.3.1.1. uart_init

La fonction *void uart_ init(void)* permet d'initialiser la liaison série à la bonne fréquence [6] en fonction de la fréquence du quartz [7]. Le mode de fonctionnement que nous utilisons est le mode polling (pour ne pas utiliser de timer).

3.3.1.2. uart_send_char

La fonction *void uart_ send_ char(unsigned char)* prend en paramètre un octet, et l'envoie par la liaison série.

3.3.1.3. puts

La fonction *void puts(const char *)* prend en paramètre un pointeur vers une chaine de caractère à zéro terminal et va utiliser la fontion uart_send_char pour envoyer la chaine caractère par caractère.

3.3.1.4. uart_recv_char

La fonction *unsigned char uart_ recv_ char(void)* permet de recevoir un octet de la liaison série.

3.3.2. Les programmes de tests

3.3.2.1. Sender

Afin de tester les fonctions de base, nous avons fait un programme qui émet en boucle un caractère [8].

Algorithm 3.4 sender.c

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include "usart.h"
4  #include "global.h"
5  #include "mac.h"
6
7  int main(void)
8  {
9          while (1)
10         {
11                 emissionOctet('@');
12         }
13         return 0;
14 }
```

3.3.2.2. Receiver

Toujours dans le cadre du débogage, nous avons fait une fonction qui reçoit un caractère par l'intermédiaire de notre protocole, et renvoie ce caractère par la liaison série. Ainsi, nous pouvons vérifier que nous recevons bien le caractère voulu.

6. La liaison série s'effectue à une fréquence de 9600 bauds.
7. Le quartz de la carte arduino est de 16 MHz.
8. Nous émettons le signal "@".

Algorithm 3.5 receiver.c

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #include "usart.h"
4  #include "global.h"
5  #include "mac.h"
6  #include "phy.h"
7
8  int main(void)
9  {
10         uart_init();
11         puts("initialisation...\n\r");
12         init_mac();
13         puts("MAC layer initialized !\n\r");
14         while (1)
15         {
16                 puts("Received : ");
17                 uart_send_char(reception_buffer.src);
18                 puts("\n\r");
19         }
20         puts("=FIN=\n\r");
21         return 0;
22 }
```

4. FPGA

Nous n'avons malheureusement pas eu le temps d'implémenter notre protocole sur FPGA, la tâche s'est avérée être plus ardue qu'espéré. En effet il n'est pas évident de transposer l'algorithme en code Verilog pour décrire le fonctionnement interne du FPGA car nous travaillons sur des signaux, et un signal ne peut être piloté que par un seul processus, ce qui rend la structure du code difficile et demande pas mal de réflexion. Ceci-dit une brique de base a été implémentée, il s'agit d'une pile FIFO qui se base sur de la BlockRAM (de la SRAM) interne aux FPGA Xilinx pour mémoriser des octets de données, puis les restituer dans l'ordre. Il s'agit en fait d'un ring buffer en mode FIFO, avec deux pointeurs (lecture / écriture), c'est une sur-couche au module classique de BlockRAM, on peut en effet envoyer à notre FIFO des ordres de lecture et des ordres d'écriture, le module va indiquer quand il est occupé, acquitter les ordres, et effectuer les lectures et les écritures et gérer en interne ses pointeurs de lecture-écriture. On n'a donc pas à s'occuper de l'adresse mémoire où l'ont lis/écrit dans la BlockRAM car ce travail est fait en interne dans le module et n'est pas visible à l'extérieur !

Ce module a été écrit en Vérilog, un module de test (test-bench) a aussi été écrit en Verilog afin de simuler le fonctionnement de notre pile FIFO et de tester la conformité de sa réaction à nos attentes. Nous avons utilisé le simulateur libre Icarus Verilog (iverilog), ainsi que l'outil libre GTKWave pour analyser les données de sortie du simulateur en affichant des chronogrammes des signaux importants afin de valider le fonctionnement du module.

Le module de test maintient tout d'abord la FIFO en état RESET pendant 1 cycle d'horloge (10 ns @ 100 MHz).

Ensuite le RESET est relaché pendant 1 cycle d'horloge, puis on demande une écriture de la valeur décimale 42 dans la FIFO (signal "do_write" mis à 1, et valeur décimale 42 mise dans le registre d'entré "di"), on attend ensuite 2 cycles d'horloge (20 ns @ 100 MHz) car c'est le temps nécessaire à la FIFO pour sauvegarder un octet et faire son travail interne sur les pointeurs de lecture/écriture. Puis de la même manière nous écrivons les valeurs décimales 25 et 32 dans la FIFO. Ensuite nous allons effectuer 3 lectures depuis la FIFO pour récupérer nos 3 octets si tout se passe bien (et dans l'ordre dans lequel nous les avons écrit).

Nous mettons alors le signal "do_write" à 0 et le signal "do_read" à 1 pour indiquer à la FIFO que nous souhaitons lire, nous attendons 2 cycles d'horloge, ce qui a pour effet de mettre sur le signal de sortie "do" la valeur lue depuis la FIFO, puis nous attendons encore 4 cycles supplémentaires afin de lire encore 2 octets.

L'affichage du chronogramme de ce test dans GTKWave permet de bien valider le fonctionnement de la FIFO, nous lisons bel et bien les octets écrit précedemment !

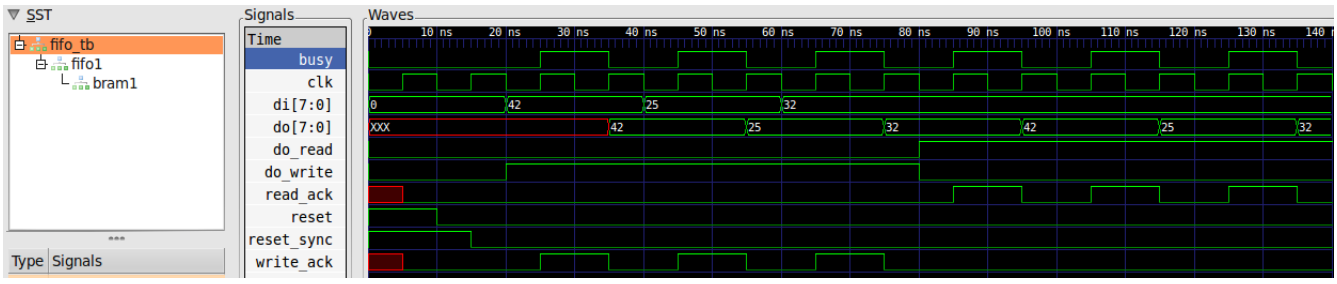


FIGURE 4.1.: chronogramme de la simulation du test-bench du module FIFO

Troisième partie .

Résultat, problème et analyse

5. Les problèmes qui sont apparus

5.1. Le compilateur

Ci dessus, je vais écrire deux bouts de code qui font presque la même chose. D'après vous quelle est la différence entre ces 2 bouts de code ?

Algorithm 5.1 Erreur compilateur 1

```
1  while(1)
2  {
3          uart_send_char('"');
4          if (has_been_received)
5          {
6                  uart_send_char(received_byte);
7                  has_been_received = 0;
8          }
9          else
10                 emissionOctet('@');
11 }
```

et

Algorithm 5.2 Erreur compilateur 2

```
1  while(1)
2  {
3          if (has_been_received)
4          {
5                  uart_send_char(received_byte);
6                  has_been_received = 0;
7          }
8          else
9                  emissionOctet('@');
10 }
```

Comme vous pouvez le voir, ces 2 bouts de codes, qui sont la boucle principale du programme main font pratiquement la même chose.

Le premier va envoyer un caractère sur la liaison série [1]. Ensuite, il va vérifier la valeur de la variable globale *has_been_received*. Si celle-ci vaut 1 [2], on remet la variable à 0, et on affiche l'octet que nous avons reçu. Sinon, on envoie le caractère "@".

Dans le second, on fait la même chose, si ce n'est que l'on n'affiche pas le caractère ".

Et bien la différence entre ces 2 programmes, c'est que le premier fonctionne très bien, et affiche les caractères reçus, tandis que le second n'affiche absolument pas le caractère reçu, alors que les interruptions nous montrent bien que l'on a reçu le caractère en question.

L'erreur viens de la compilation. Compiler le programme avec l'option d'optimisation d'avr-gcc (-O2) fait que le programme 2 ne fonctionne pas. Sans cette option (avec simplement -O0 ou rien du tout), le

1. Il va envoyer le caractère ".
2. Cette variable est remise à 1 lors par interruption lors de la réception d'un octet.

second programme fonctionne très bien.

Malheureusement pour nous, nous avions rajouté l'option d'optimisation de la compilation pour éviter d'avoir des *warnings* lors de la compilation de la librairie *delay*. Nous avons ainsi perdu beaucoup de temps à chercher d'où provenait cette erreur assez incompréhensible. On n'arrivait pas à savoir d'où elle venait.

Ce n'est que plus tard, au restaurant avec un ancien élève du club de robotique de l'école (David Coutherut), en discutant de ce problème que nous avons appris le pourquoi du comment. Avec l'optimisation lors de la compilation, le test sera effectué lors du premier passage dans le if. Pour les suivants, il va utiliser le résultat qu'il a déjà, et donc, ne va pas aller vérifier si la variable a changé ou non. Pour éviter d'avoir ce problème, il aurait fallu que l'on rajoute l'option *volatile* devant la définition de notre variable. Il s'agit en fait de forcer à lire et à écrire directement en mémoire RAM en by-passant la mémoire cache du System-on-Chip AVR, seulement nous pensions que l'AVR n'avait pas de mémoire cache, erreur !

5.2. La désynchronisation

5.2.1. Problème de lecture d'un octet

Lors des tests que nous avons effectué, nous nous sommes rendu compte qu'il y avait un problème lors de la réception. En effet, nous envoyions le caractère"@", et nous recevions un caractère totalement différent. Le plus troublant, c'est que parfois, notre programme foncionnait très bien.

Après des nombreuses recherches pour savoir pourquoi, nous nous sommes rendu compte qu'il y avait une désynchronisation entre les bits envoyés et ceux que nous recevions. En effet, si j'envoyais '010000100' [3], nous recevions parfois '100001000'. Par contre, si on lançait les programmes au bon moment, les données passaient très bien.

Pour corriger ce problème, nous avons décidé de rajouter, par rapport à ce que nous avions prévu en théorie, un bit de start, qui est unique, et qui ne peut pas être confondu avec un bit normal. Dès que nous avons rajouté ce bit de start, les octets étaient bien reçus, quelque soit la façon dont on démarrait les microcontrôleurs.

5.2.2. Problème de lecture d'un paquet

Le même problème est apparu lorsque la couche 2 [4] lisait les octets pour former les paquets. Elle considérait le premier octet reçu comme le premier octet de la trame, donc cela pouvait créer un décalage. Pour corriger cette erreur, nous avons décidé de créer un octet de start, qui est aussi l'adresse source du paquet. En fait, nous avons décidé que toutes les adresses commenceraient par 101.

Donc, comme le paquet commence par l'adresse source, forcément, le premier octet doit être de la forme 101xxxxx.

5.3. Le datasheet et la liaison série

Pour le déboguage, nous avons décidé d'utiliser la liaison série, car celle-ci est pratique, et facile à mettre en place. Mais, lorsque nous l'avons implémentée, nous nous sommes rendu compte que celle-ci ne fonctionnait pas. Après quelques recherche pour trouver l'origine, qui aurait pu être dans le max232, ou le convertiseur série-USB, nous l'avons trouvée, et elle venait du code. En effet, dans le datasheet, il était indiqué d'utiliser la formule suivante pour configurer le débit de la liaison série : $UBRR = \frac{f_{osc}}{2*BAUD} - 1$. Après analyse de la librairie serial de la carte arduino, nous nous sommes rendu compte que la formule qui fallait utiliser était la suivante : $UBRR = \frac{\frac{f_{osc}}{8*BAUD}-1}{2}$. Il nous a suffit de changer la formule pour que la liaison série fonctionne correctement.

3. Cela correspond à 0x42 plus le bit de parité.
4. La couche MAC.

6. Les résultats

6.1. Emission et réception simultanées

Pour tester le fonctionnement de notre système, nous avons fait un premier test. Le principe de celui-ci est d'envoyer un caractère en continue, et en cas de réception d'un octet, afficher ce que nous avons reçu.

Algorithm 6.1 Test d'émission et réception en simultanée

```
1  while(1)
2  {
3          if (has_been_received)
4          {
5                  uart_send_char(received_byte);
6                  has_been_received = 0;
7          }
8          else
9                  emissionOctet('@');
10 }
```

Ensuite, nous avons relié entre eux 2 microcontrôleurs flashés avec ce même programme pour observer le foncionnement de notre programme.

Le résultat est positif étant donné que les caractères sont envoyés un coup par l'un, un coup par l'autre, de manière aléatoire, sans qu'il y ait de caractère erroné, ou de chevauchement de caractère. Lorsque un microcontrôleur est en émission, l'autre ne peut absolument pas passer en émission.

6.2. La couche haute

Pour utiliser notre programme, il suffit d'inclure notre librairie avec la commande suivante : *#include "mac.h"*. Ensuite, il suffit d'envoyer des octets en appelant la fonction *void send(uint8_ t address_ dest, uint8_ t data[16], uint8_ t taille)*. Cette fonction envoie un caractère dès que la ligne est libre.

Pour la réception, il suffit d'appeler la fonction *int8_ t recv (uint8_ t *src, uint8_ t *taille, unsigned char *datas)*. Cette fonction retourne un entier négatif en cas d'erreur de réception. Si l'entier est positif, c'est que le paquet a bien été reçu.

Quatrième partie .

Nos impressions

7. Jérémy Cheynet

7.1. Le langage C version microcontrôleur

Ma première surprise lorsque nous avons commencé ce PJ fut le langage C. J'avais l'habitude de programmer des microcontrôleurs en basic, ou en langage C arduino [1]. Il y a une différence de difficulté entre les 2 versions : une version est très bas niveau avec écriture dans les registres et programmation avec le datasheet à côté, tandis que l'autre version est beaucoup plus haut niveau, sans trop comprendre le principe de fonctionnement du système.

Algorithm 7.1 Example de code de type arduino : faire clignoter une LED

```
1  int ledPin =  13;     // LED connected to digital pin 13
2
3  void setup()
4  {
5          pinMode(ledPin, OUTPUT);
6  }
7
8  void loop()
9  {
10         digitalWrite(ledPin, HIGH);
11         delay(1000);
12         digitalWrite(ledPin, LOW);
13         delay(1000);
14 }
```

Le code ci-dessus configure le port C, bit 5 en sorti (renommé pin 13 sur arduino) et le fait clignoter en le changeant d'état toute les secondes. Il est évident à comprendre, mais on ne sait pas trop ce qu'il se passe lorsque l'on fait le *pinMode*, par exemple. Maintenant, voyons ce que cela donne en langage C avr :

Algorithm 7.2 Example de code de type langage C avr : faire clignoter une LED

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  int main (void)
5  {
6          DDRC |= ( 1 << DDC5);
7
8          while(1)
9          {
10                 PORTC |= ( 1 << PORTC5 );
11                 _delay_ms(1000);
12                 PORTC &= ~( 1 << PORTC5 );
13                 _delay_ms(1000);
14         }
15 }
```

On voit bien dans cette seconde version du même programme que comment fonctionne le système, où l'on écrit les 1 et les 0, les registres de configuration, etc ...

1. Langage C simplifié que l'on utilise avec la plateforme de développement arduino

Personnellement, je préfère la seconde méthode pour programmer, car l'on comprend beaucoup mieux ce qu'il se passe, et aujourd'hui, je pense être capable de programmer beaucoup plus facilement qu'avant.

7.2. Git

Pour que l'on puisse travailler en parallèle, Yann a mis en place un répository git. J'ai donc du apprendre le fonctionnement de git, car jusque là, j'utilisais seulement des dépots svn. Cet outil est très pratique, mais un peu plus complexe à utiliser que svn. Peut-être que par la suite, j'utiliserai git, mais pour l'instant, je préfère quand même svn. Seul le futur me dira ce que j'utiliserai ...

7.3. Résolution des problèmes

Losqu'un programme fonctionne du premier coup, ce n'est pas normal. Heureusement pour nous, nous sommes resté normaux, et nos programmes ont eu quelques bogues. C'est donc ainsi que nous avons appris à trouver les problèmes, et surtout les résoudre.

Pour la détection de problème, nous avons utilisé la liaison série. Nous avons donc appris à faire afficher des caractères ou des entiers utiles, et au bon endroit. La méthode que nous avons appliqué est de cibler fonction par fonction pour voir où se situe le problème. Une fois que nous l'avons trouvé, il ne restait plus qu'à le déboguer.

En général, le débugage est facile. On s'apperçoie vite de ce qu'il manque au programme. Mais parfois, on ne le sait pas. Ce fut le cas avec l'erreur du compilateur. Le programme fonctionnait très bien avec une compilation sans optimisation. Mais dès que l'on compilait avec une optimisation, le programme ne fonctionnait plus. Malheureusement pour nous, il était conseillé de compiler avec l'option d'optimisation pour éviter des *warnings* lors de la compilation d'une librairie. Et nous avons perdu un temps fou à chercher l'erreur dans notre programme, alors qu'il fallait aller voir les options de compilation. Maintenant, je serai au courant, ce qui m'évitera de perdre beaucoup de temps sur une chose comme ça.

7.4. Conclusion

Je suis très content d'avoir fait ce programme, étant donné qu'il nous a permis de voir beaucoup de choses, de corriger des problèmes. Il m'a aussi montré que dans la théorie, nous oublions souvent des choses que nous gène dans la pratique car des problèmes que nous n'imaginons pas peuvent apparaitre.

8. Yann Sionneau

8.1. Le C pour microcontrôleur

Bien qu'ayant déjà fait du C pour microcontrôleur lors d'un stage (sur AVR AtTiny24), l'implémentation du protocole a été parsemée d'embuches et loin d'être triviale, je pensais sincèrement que ça nous poserait moins de problèmes et que nous pourrions donc nous consacrer à ce qui me semblait être le plus complexe : l'implémentation sur FPGA.

Ceci-dit c'était mon premier projet en utilisant la plaque de développement Arduino, qui je l'avoue simplifie pas mal le développement pour plusieurs raisons :

1. C'est pas cher
2. Le programmateur est intégré, il suffit de brancher en USB et on peut programme avec avrdude
3. C'est alimenté en 5V par l'USB
4. Il y a une liaison Série over USB, pas besoin de MAX232 et de FTDI externe, tout est sur la plaque !
5. Pas besoin de se faire son propre PCB, de le router, percer, souder : on gagne du temps pour la partie software

L'Arduino a aussi l'avantage de proposer un utilitaire (en java) pour programmer et flasher le microcontroleur, tout en fournissant une librairie d'abstraction assez haut niveau pour pouvoir faire du microcontrôleur (ou plutôt de "l'Arduino" dans ce cas là) sans forcément être très calé en C, en software ou en hardware.

En effet l'arduino était à la base fait pour des artistes, donc pas forcément calés en informatique !

Il y a donc des fonctions simples pour rajouter des routines d'interruptions, changer le mode d'une pate, changer l'état d'une pate, gérer un timer, une liaison série etc...

Tout ce qui est bas niveau, registres et autres est caché par la librairie Arduino.

J'étais en effet d'accord pour utiliser la plaque de développement Arduino pour les points 1 à 5, mais je n'étais pas partant pour faire du "C Arduino" car je trouvais bien plus intéressant de lire la datasheet du composant et de toucher directement aux registres afin de configurer le device comme il faut. Cela fait en effet perdre pas mal de temps, car au début ça ne marche pas, on réinvente la roue, mais c'est très formateur et c'était bien le but ici : apprendre.

Que dire de plus dans cette partie, si ce n'est que j'ai beaucoup appris encore sur l'architecture AVR à force de lire la datasheet et de jouer avec les registres de ce joli composant qui ne demanderait plus qu'à être ouvert (au niveau des sources vérilog/vhdl de la puce) pour être un produit séduisant !

8.2. Git

Je dois dire que c'est moi qui ai insisté pour qu'on utilise git en lieu et place de svn, je savais déjà que Jérémy avait bien l'habitude de svn, et je voulais lui montrer qu'il existait autre chose et je pense clairement que git est bien plus puissant et mieux fait que git. En effet git a été développé à l'origine par Linus Torvalds pour versionner les sources du noyau en remplacement de BitKeeper tombé sous le coup d'un copyright.

Après avoir écouté quelques conférences à propos de git (dont une de Linus lui même) il me semble que c'est vraiment un outil très puissant et pratique, mais cependant pas forcément évident à appréhender.

Je l'avais déjà utilisé, mais uniquement les fonctions basiques et nous nous sommes donc mis à l'utiliser intensément (via un repository sur la forge github.com)

Cependant etant debutant avec git nous avons commis quelques erreurs, surtout au niveau des branches qui ont par la suite compliqué son utilisation et aussi le développement à plusieurs. Pourtant l'outil est censé faciliter le travail à plusieurs, mais je suis convaincu que c'est à cause de notre mauvaise utilisation de cet outil que notre productivité à baissé à un moment. L'outil en lui même est bon mais nous l'avons mal utilisé. Nous avons cependant appris de nos erreurs et je pense que par la suite je vais continuer de l'utiliser, en essayant de ne pas refaire les même erreurs, même si je marche quand même encore sur des oeufs en l'utilisant.

8.3. FPGA

Les FPGAs sont un peu ma passion du moment, j'ai fais découvrir ces puces aux gens de MiNET et d'INTech et leur ai fais quelques petites démonstrations simples que j'ai pu faire avec mon niveau de débutant, puis j'ai par la suite mené à bien avec Guillaume Rose (1A) un petit projet de bruteforcer de hash MD5 sur FPGA qui fonctionne plutôt pas mal !

Je dois dire que je suis très déçu de ne pas avoir eu le temps d'implémenter le protocole sur FPGA, car c'était la partie qui m'intéressait le plus !

Seulement devant l'ampleur de la tâche, et le fait que nous avions pas mal de bugs sur la partie microcontrôleurs, nous avons préféré nous consacrer à la correction des bugs afin d'avoir au moins une version fonctionnelle sur microcontrôleur plutôt que du code non achevé/non fonctionnel sur FPGA et microcontrôleur.

8.4. Conclusion, mon sentiment

Je suis frustré de ne pas avoir mené la mission à bien à 100%, la partie FPGA n'est qu'un brouillon de réflexion et qu'un début d'implémentation.

Mais je suis bien content que celà fonctionne sur microcontrôleur vu la quantité de problèmes rencontrée.

J'ai clairement le sentiment d'avoir appris de la rigueur et d'avoir acquis encore de l'expérience dans le monde de l'embarqué qui m'intéresse tant, même si j'aurai préféré gagner en compétences en FPGA plutôt qu'en microcontrôleurs AVR.

Je garderai donc un bon souvenir de cette expérience de projet en binôme, un sentiment de satisfaction à propos de l'implémentation fonctionnelle sur microcontrôleurs et un arrière goût amer pour la partie FPGA inachevée.

Je conclurai sur le fait que bien que ce projet puisse paraitre trivial (Quoi encore une implémentation d'un protocole série ? il en existe déjà plein ! I2C, SPI, RS-232, 1 wire etc ...) il arrive en fait une quantité non négligeable de problèmes dès qu'on veut mettre en pratique la théorie. Il faut donc réellement faire les choses afin de vraiment jauger leur degré de difficulté, et ne pas juger à priori. Il suffit de comparer le nombre de pages de ce rapport consacrées à la partie "Théorie" et à la partie "Pratique" pour se rendre compte de la différence des deux aspects !

Je conclue donc sur la fameuse expression idiomatique (en anglais pour Jérémy ;)) : "Don't judge a book by its cover."

Table des figures

2.1. Carte Arduino 6
2.2. Plaque de développement “AVnet Spartan-3A Evaluation Kit” 7
2.3. Les 4 types de bits possible 8
2.4. Représentation des état haut et bas 8
2.5. Exemple d’une trame d’un octet (0x42) 8
2.6. Structure d’un paquet 9

4.1. chronogramme de la simulation du test-bench du module FIFO 18

Liste des algorithmes

3.1. Prototypes des fonctions de phy.c 11
3.2. Prototype des fonctions de mac.c 13
3.3. prototype des fonctions utilisant la liaison série 15
3.4. sender.c 16
3.5. receiver.c 17

5.1. Erreur compilateur 1 20
5.2. Erreur compilateur 2 20

6.1. Test d'émission et réception en simultanée 22

7.1. Example de code de type arduino : faire clignoter une LED 24
7.2. Example de code de type langage C avr : faire clignoter une LED 24