Compte-Rendu TP CNPA

Unité de calcul FPGA et hardware trojan

DERLIQUE Florian
DUFOUR Alix

SE4SC 2021-2022

Sommaire

Introduction

Dans les circuits numériques, le calcul est une fonction primordiale. Cependant, les calculs intermédiaires doivent rester confidentiels afin de ne pas exposer les algorithmes. De plus, les circuits étant complexes, il est difficile de tester complètement l'intégrité des fonctions réalisées. Afin de détourner ce problème, il est possible de créer une porte dérobée.

Durant ces séances de TP, nous avons tout d'abord réalisé une unité de calcul FPGA avec une addition de deux vecteurs sur N bit, et la multiplication de deux vecteurs complexes sur N bit. Ils ont tous les deux été implémentés dans une version naïve pour commencer, puis dans une deuxième version optimisée et pipelinée. Ceci afin de pouvoir comparer leurs performances.

Nous avons par la suite réalisé une RAM "sécurisée" car elle possède une zone accessible sans limitation, et deux zones qui sans entrées particulières ne sont respectivement pas du tout accessibles, ou juste en lecture.

Pour finir nous avons réalisé une porte dérobée dans la mémoire "sécurisée", qui permet d'accéder aux données normalement protégées en lecture/écriture dans la RAM.

I - Principe de fonctionnement des composants

I.1) Unité de calcul - Additionneur N bits

Le premier élément de notre unité de calcul est un additionneur de vecteurs de N bits, nous avons créé 2 version de cette additionneur :

- Une version naïf (non-pipelinée)
- Une version pipelinée (utilisation de bascules D pour traiter les données en parallèle)

L'entité prend en compte deux entrées de données, la sortie de l'additionneur et une horloge

```
entity AddNbit is
    generic(N:natural := 4);
    Port ( clk : in STD_LOGIC;
           a_in : in STD_LOGIC_VECTOR(N-1 downto 0);
           b_in : in STD_LOGIC_VECTOR(N-1 downto 0);
           s_out : out STD_LOGIC_VECTOR(N-1 downto 0));
end AddNbit;
```

On commence par la version non-pipelinée, il suffit de réaliser une addition des deux entrées :

```
s <= std_logic_vector(unsigned(a_in) + unsigned(b_in));
```

Nous sommes obligés d'utiliser une horloge puis d'un ensemble de bascules D en entrée et en sortie, afin de capturer le temps d'exécution de ce module, dans ce cas l'addition s'effectue lorsqu'on détecte un front montant sur notre horloge :

```
if(clk'event and clk='1') then
    a<=a_in;
    b<=b_in;
    s <= std_logic_vector(unsigned(a) + unsigned(b));
end if;
```

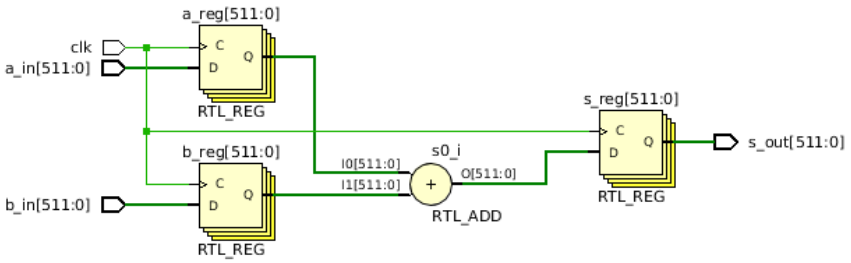


Schéma 1.1 : Additionneur N bits non-pipelinée (les bascules sont présentes pour les tests de temps)

Puis ensuite on réalise la version pipelinée de notre additionneur, le principe est de découper le calcul en deux. En d'autres termes, le module additionne les 4 premiers bits des données d'entrée et les 4 derniers bits des données d'entrée en parallèle. Puis on ajoute la carry sur le résultat de l'addition des 4 derniers bits.
On n'oublie pas de séparer les deux calculs par un ensemble de bascules D (e.g affectation de signaux dans un process en VHDL).
Nous sommes obligés de passer par des vecteurs de 0 à N/2 puisque nous devons capturer la carry (retenue) des sommes intermédiaires, pour pouvoir stocker les données de taille dans les bascules et de faire l'ajout de la carry, il faut absolument faire une concaténation pour additionner des vecteurs de même taille et le stocker dans un signal de même taille.

```
signal aLSB1,aMSB1,bLSB1,bMSB1,cLSB1,sMSB2,sMSB1,sLSB1 : unsigned(N/2 downto 0);
signal sLSB2 : unsigned(N/2 -1 downto 0);
begin
process(clk)
    begin
    if(rising_edge(clk)) then
        aLSB1<='0'&unsigned(a_in((N-1)/2 downto 0));
        aMSB1<='0'&unsigned(a_in(N-1 downto N/2));
        bLSB1<='0'&unsigned(b_in((N-1)/2 downto 0));
        bMSB1<='0'&unsigned(b_in(N-1 downto N/2));
        --somme des LSB et MSB
        sMSB1<=aMSB1+bMSB1;
        sLSB1<=aLSB1+bLSB1;
        --ajout de la carry de sLSB dans sMSB
        sLSB2<=sLSB1((N/2)-1 downto 0);
        sMSB2<= sMSB1 + ('0' & sLSB1(N/2));
    end if;
end process;
s_out(N-1 downto N/2)<=std_logic_vector(sMSB2((N/2)-1 downto 0));
s_out((N/2)-1 downto 0)<=std_logic_vector(sLSB2);
```

On obtient le schéma suivant :

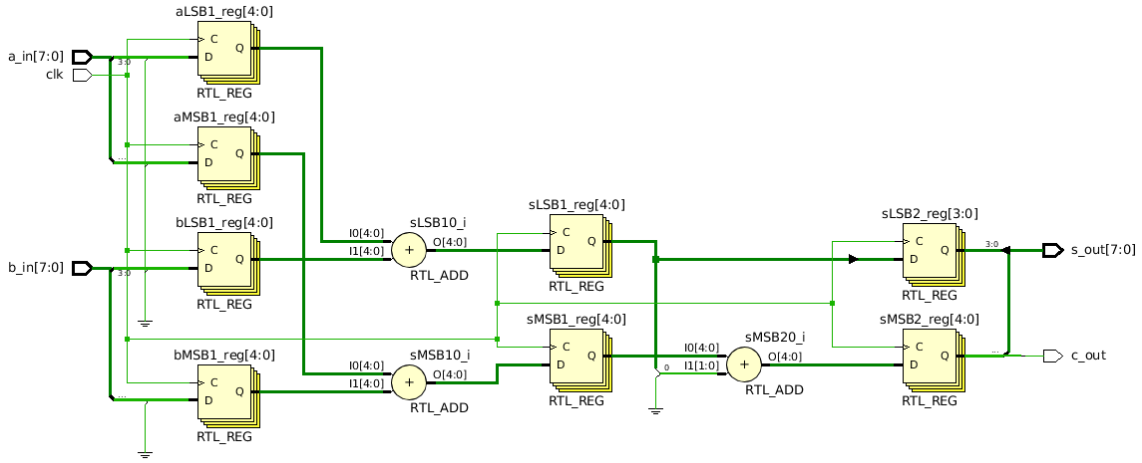


Schéma 1.2 : Additionneur N bits pipelinée

I.2) Unité de calcul - Multiplicateur complexe N bits

La dernière partie de notre unité de calcul consiste à faire un multiplicateur complexe de vecteurs N bits, nous devons aussi implémenter 2 versions : une non-pipelinée et l'autre pipelinée.
Comme pour l'additionneur, l'entité aura la même forme, il faut juste séparer les réels et imaginaires :

```
entity MulComplex_v1 is
    generic(N:natural:=128);
    Port ( ar_in : in STD_LOGIC_VECTOR (N-1 downto 0);
           ai_in : in STD_LOGIC_VECTOR (N-1 downto 0);
           br_in : in STD_LOGIC_VECTOR (N-1 downto 0);
           bi_in : in STD_LOGIC_VECTOR (N-1 downto 0);
           clk : in STD_LOGIC;
           r_out : out STD_LOGIC_VECTOR (2*N-1 downto 0);
           i_out : out STD_LOGIC_VECTOR (2*N-1 downto 0));
end MulComplex_v1;
```

On simplifie notre calcul : $(a + ib)(c + id) = ac - bd + i(ad + bc)$ et on met alors $ac - bd$ dans la sortie qui correspond à la partie réelle
$ad + bc$ dans la sortie qui correspond à la partie imaginaire.

```
r_out<=std_logic_vector(signed(ar_in)*signed(br_in)-signed(ai_in)*signed(bi_in));
i_out<=std_logic_vector(signed(ar_in)*signed(bi_in)+signed(ai_in)*signed(br_in));
```

On n'oublie pas les bascules D d'entrée et de sortie pour capturer le temps d'exécution, on fait pareil que pour l'additionneur et on obtient ceci :

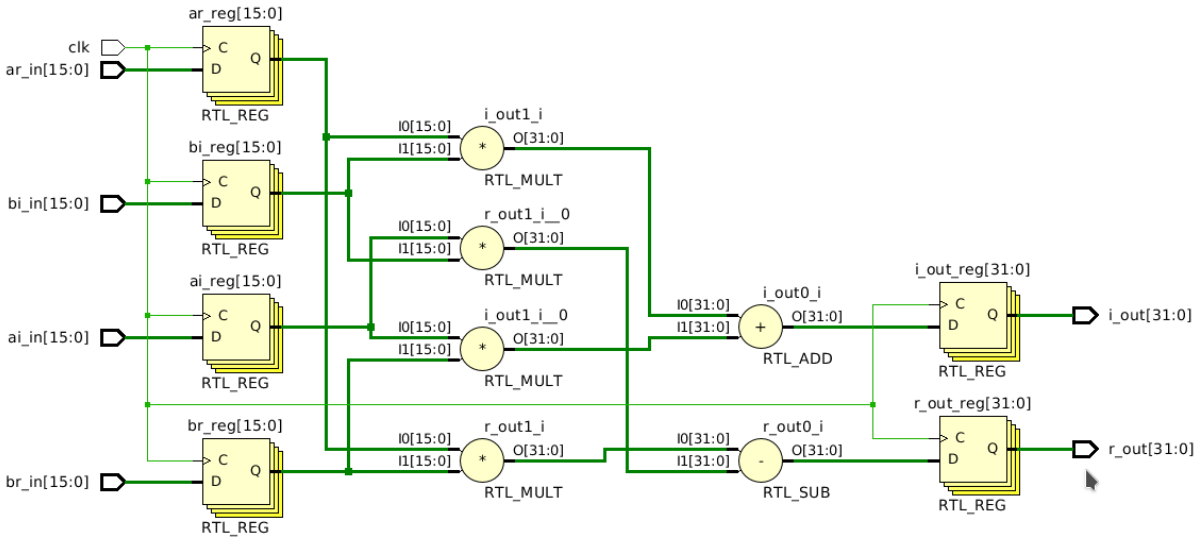


Schéma 1.3 : Multiplicateur complexe N bits pipelinée

Ensuite, nous réalisons une version pipelinée, l'astuce pour faire la version pipelinée est de trouver une solution pour ne faire que 3 multiplications au lieu de 4, nous décomposons alors plus le calcul, nous avons 3 étages avec 3 multiplications :

$r_{out} = ac - bd = prod1 + com$ et $i_{out} = ad + bc = prod2 + com$ où :

$$prod1 = a * pre2$$
$$prod2 = b * pre3$$
$$com = d * pre1$$
$$pre1 = a - b$$
$$pre2 = c - d$$
$$pre3 = c + d$$

```
signal ar,ai,br,bi : STD_LOGIC_VECTOR(N-1 downto 0);
signal pre1,pre2,pre3 : signed(N-1 downto 0);
signal prod1,prod2,com : signed(2*N-1 downto 0);
begin
process(clk)
begin
if(rising_edge(clk)) then
    ar <= ar_in;
    ai <= ai_in;
    br <= br_in;
    bi <= bi_in;
    pre1 <= signed(ar) - signed(ai);
    pre2 <= signed(br) - signed(bi);
    pre3 <= signed(br) + signed(bi);
    com <= signed(bi)*pre1;
    prod1 <= signed(ar)*pre2;
    prod2 <= signed(ai)*pre3;
    r_out <= std_logic_vector(com + prod1);
    i_out <= std_logic_vector(com + prod2);
    end if;
end process;
```

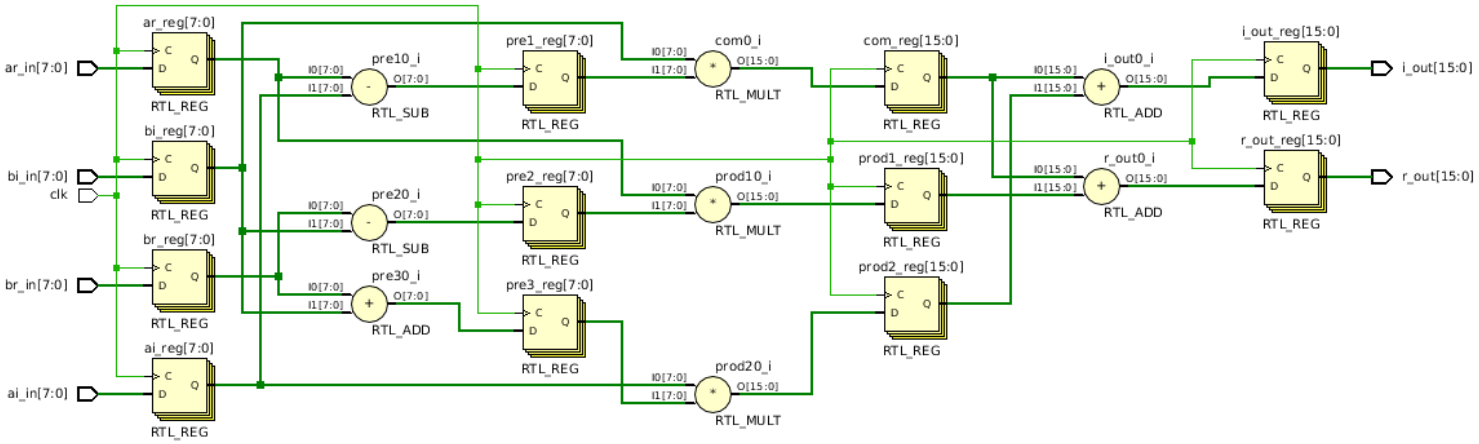


Schéma 1.4 : Multiplicateur complexe N bits pipelinée

I.3) RAM sécurisée

Cette partie consiste à réaliser une ram sécurisée, la RAM comporte 3 parties :

- Une non-protégée en écriture et en lecture
- Une protégée en écriture et non-protégée en lecture
- Une protégée en lecture et écriture

Il nous faut des entrées pour savoir si nous devons lire ou écrire dans la RAM, deux entrées de flags pour débloquer la protection, ainsi que d'autres entrées pour spécifier l'adresse de la RAM pour l'opération ainsi que pour spécifier quelle partie de la RAM nous devons utiliser. Et pour finir, les entrées de données d'entrées et de sorties.
L'entité en vhdl de ce composant, aura cette forme :

```
entity RAM is
    generic(    N:natural:=4;
                MemSize:natural:=255;
                AddrSize:natural:=7);
    port(clk : in STD_LOGIC;
           we_in : in STD_LOGIC;
           re_in : in STD_LOGIC;
           unlock_w_in : in STD_LOGIC;
           unlock_rw_in : in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR(N-1 downto 0);
           data_out : out STD_LOGIC_VECTOR(N-1 downto 0);
           addr_in : in STD_LOGIC_VECTOR(AddrSize downto 0);
           select_ram_in : in STD_LOGIC_VECTOR(1 downto 0));

end RAM;
```

En VHDL, pour créer une RAM, il suffit tout simplement de créer un nouveau type une liste (de taille donnée) de STD_LOGIC_VECTOR (ici aussi d'une taille donnée). Le type aura pour nom *memory.* Puis de créer 3 signaux, un signal par parties de RAM :

```
Type memory is array(MemSize downto 0) of STD_LOGIC_VECTOR(N-1 downto 0);
signal RAM1,RAM2,RAM3: memory;
```

Il est possible d'initialiser la RAM par des valeurs par défauts, ce que nous avons décidé de ne pas faire, vu que la RAM sera mise-à-jour au fur et mesure par les calculs de l'unité de calcul. Dans l'architecture il suffit de faire un case / when sur l'ensemble des parties de RAM disponible, et de tester si les flags sont bons pour pouvoir écrire ou lire sur la RAM, voici un exemple du code pour une des parties de la RAM protégée en entrée et en sortie

```
--"10" --> RAM3
        when "10" =>
            if(unlock_rw_in = '1' and we_in = '1') then
                RAM3(to_integer(unsigned(addr_in))) <= data_in;
            elsif(unlock_rw_in = '1' and re_in = '1') then
                data_out <= RAM3(to_integer(unsigned(addr_in)));
            end if;
```

I.4) Backdoor

Cette dernière partie consiste à réaliser une backdoor (porte dérobée) dans la RAM sécurisée.
Cette porte dérobée consiste en 2 principaux composants :

- Un automate de détection de séquence afin d'activation de la porte dérobée
- Un émetteur OOK pour transmettre les donnée de la RAM sécurisée en lecture et écriture

Nous avons décidé que notre automate de détection de séquences, détectera des séquences d'une taille de 5 bits, ce choix s'explique notamment dû à certains choix :

- Plus la détection est complexe, moins la porte dérobée est détectable
- Si la détection est sur beaucoup de bits, il faut plus de temps pour réaliser toute le détection

Il nous faut un nombre d'états ni trop élevé, ni trop bas, 5 bits nous semble bien. Si la détection est vraie, alors l'automate envoie un flag de sortie

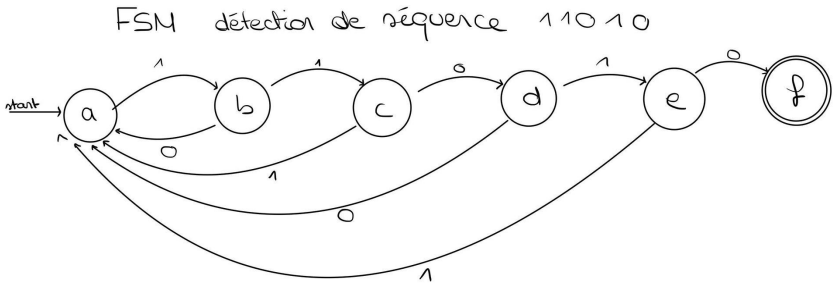


Schéma 1.5 : Exemple de FSM de détection de la séquence [11010]

Voici son entité :

```
entity FSM is
    Port ( data_in : in STD_LOGIC_VECTOR (4 downto 0);
           s_out : out STD_LOGIC;
           clk : in STD_LOGIC);
end FSM;
```

Le code est assez facile en soit, il suffit de faire un case / when, sauf qu'il faut définir les états comme ci :

```
type state_type is (s0,s1,s2,s3,s4,s5,s6);
signal state : state_type := s0 ;
```

Où s0 est l'état initial et s6 l'état final. L'état de détection ce code comme ci :

```
when s1 =>
            if(data_test(0) = '0') then
                state <= s2;
            else
                state <= s0;
                s_out<='0';
```

```
            end if;
```

Ensuite nous avons implémenté l'émetteur OOK, elle se met en action dès qu'une séquence est détectée. Le principe est assez simple et est résumé par ce schéma :

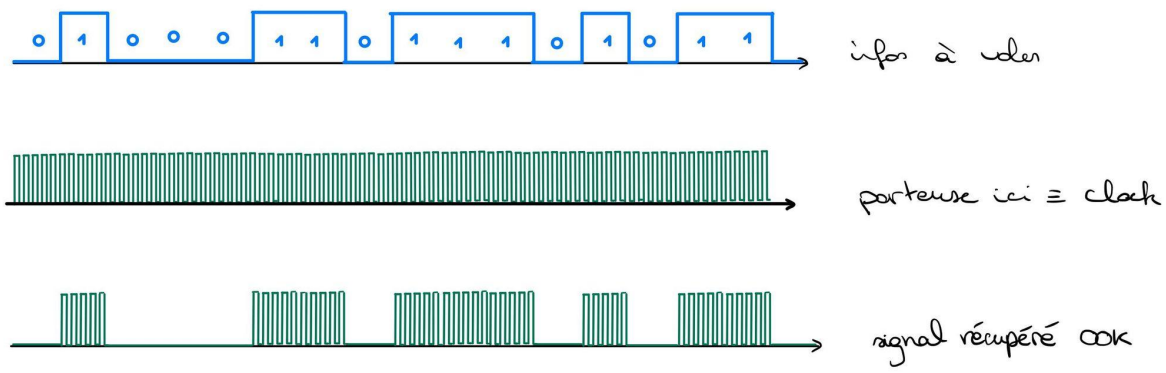


Schéma 1.6 : Principe d'un modulation On-Off Keying

Voici son entité :

```
entity OOK is
    generic(N:natural:=8);
    Port ( data_in : in STD_LOGIC_VECTOR (N-1 downto 0);
           s_out : out STD_LOGIC;
           clk : in STD_LOGIC;
           start_in : in STD_LOGIC);
end OOK;
```

Ce composant nous permet de comprendre plus l'intérêt des variables, en effet comme nous utilisons des compteurs, nous avons besoin de l'actualisation en temps réel de ces derniers, or c'est impossible en utilisant des signaux. Nous utilisons alors des variables que nous actualisons dans le process, puis des signaux pour stocker l'état des compteurs à chaque fin de process.
Dans notre code, dès qu'on détecte le début de l'émission, on met à '1' un signal qui permettra de savoir si l'émission a débuté ou non, un bit sera envoyé en 10 coups d'horloge, dès que la donnée est intégralement parcourue, on remet à '0' le signal de début d'émission

```
signal cmp,data_cmp : integer := 0;
signal start : STD_LOGIC := '0';
begin
OOK : process(clk,start_in)
variable tmp_cmp,tmp_data_cmp : integer := 0;
begin
if(start_in = '1' and start='0') then
    start <= '1';
end if;
if(start = '1') then
    if(rising_edge(clk)) then
        tmp_cmp := cmp;
        tmp_data_cmp := data_cmp;
        if(tmp_cmp < 10) then
```

```
            tmp_cmp := tmp_cmp + 1;
        else
            tmp_data_cmp := tmp_data_cmp + 1;
            tmp_cmp := 0;
        end if;
        if(tmp_data_cmp > N-1) then
            tmp_data_cmp := 0;
            start<='0';
        end if;
        data_cmp <= tmp_data_cmp;
        cmp<= tmp_cmp;
    end if;
    s_out <= data_in(tmp_data_cmp) and clk;
else
    s_out <= '0';
end if;
end process;
```

Pour finir nous ajoutons ces deux composants au fichier contenant la RAM, il sera plus aisé de récupérer les données dans la RAM en cas de détection de séquences. On fait ceci grâce à des port map, par exemple le port map pour la fsm :

```
--avant le begin de l'architecture
component FSM
    Port ( data_in : in STD_LOGIC_VECTOR (4 downto 0);
           s_out : out STD_LOGIC;
           clk : in STD_LOGIC);
end component;
--après le begin de l'architecture
u2: FSM port map(
    data_in=>data_in(4 downto 0),
    s_out=>FSM_out,
    clk=>clk
    );
```

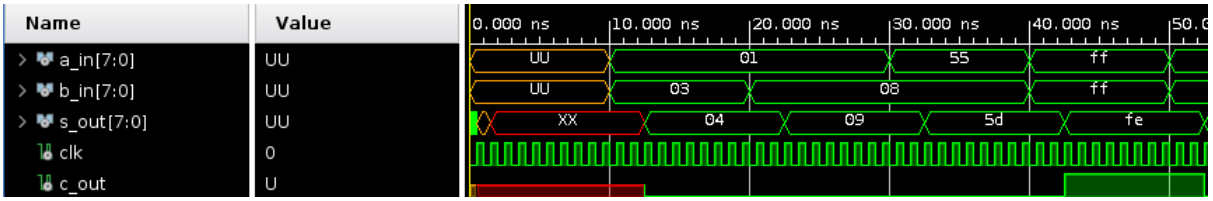
Puis on intègre ce code si il y a une détection, on change l'état du flag de l'OOK

```
backdoor: process(clk,FSM_out,start_in,backdoor_in)
begin
if(rising_edge(clk))then
    if(start_in = '0' and FSM_out = '1' and backdoor_in = '1') then
    data_OOK <= RAM3(to_integer(unsigned(addr_in)));
    start_in<='1';
    end if;
end if;
```

II - Tests de simulations et évaluations de performance

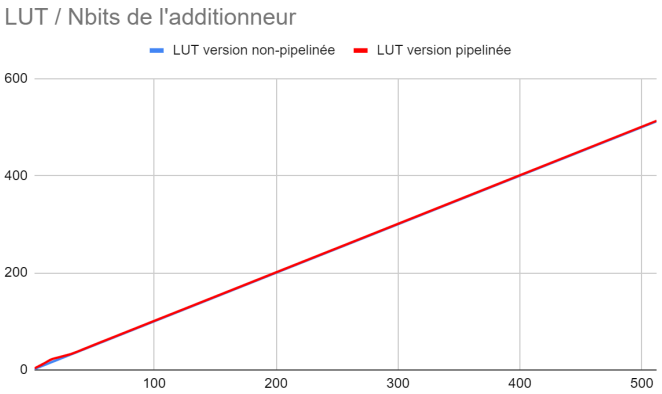
II.1) Unité de calcul - Additionneur N bits

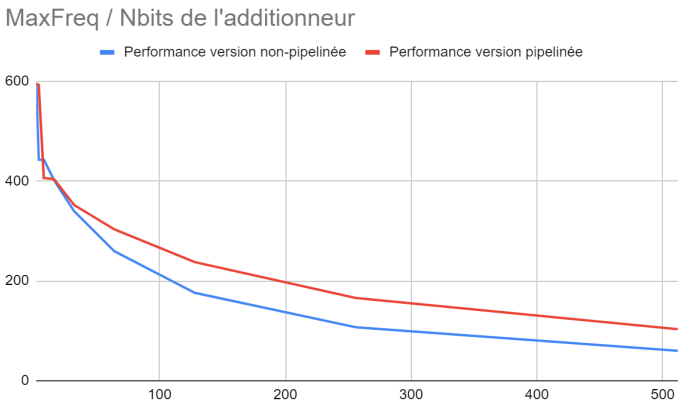
Nous réalisons un testbench, pour voir si notre version pipelinée marche, nous avons pas décidé de faire de testbench pour la version non-pipelinée, puisqu'il nous semble pas utile de tester ce dernier (le code consiste juste à une addition en une ligne) :

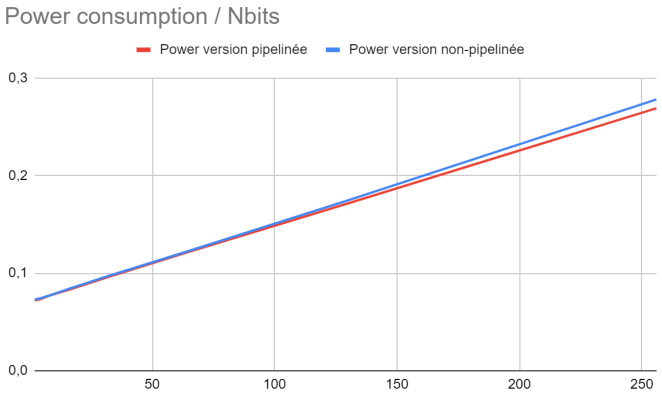


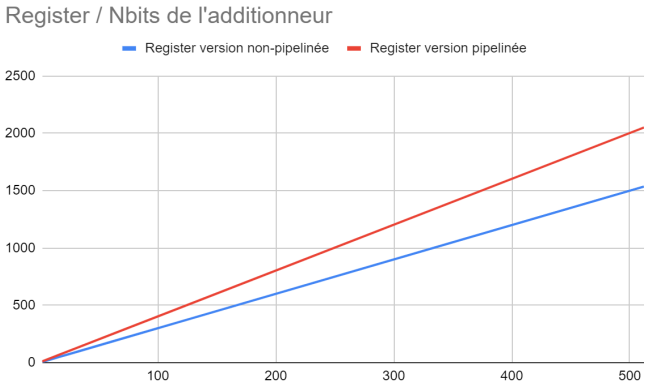
On remarque que notre version pipelinée marche (1+3=4) dans notre code détecte aussi la carry (ff + ff = 1fe), lorsqu'il n'y a pas de données d'entrée, la sortie à un état interdit puisque les bascules sont vides.

On procède au test de performance, pour synthétiser les données, nous réalisons 4 graphiques : la fréquence maximale, les LUTs, les registres et la consommation de puissance selon le nombre de bits (bleu : version non-pipelinée, rouge : version pipelinée)





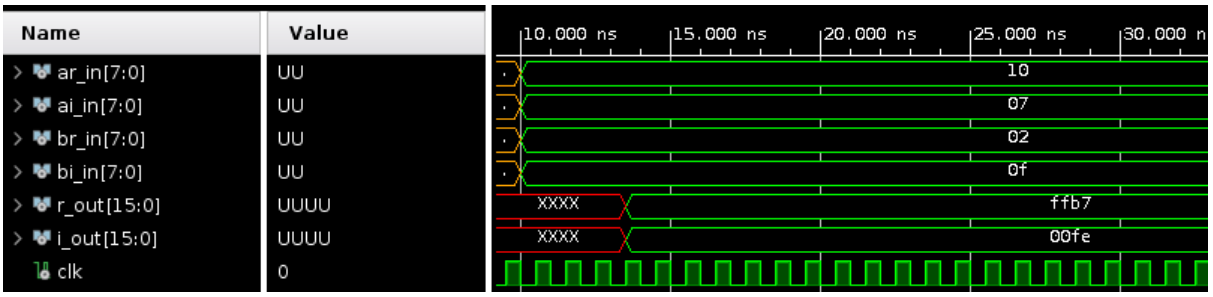




On remarque qu'il n'y a aucune différence de LUT et presque aucune différence de puissance, l'avantage de la version pipelinée est de pouvoir l'utiliser à une fréquence plus élevée uniquement rentable avec un nombre de bits élevé. La version pipelinée utilise beaucoup plus de registre, ce qui est un désavantage

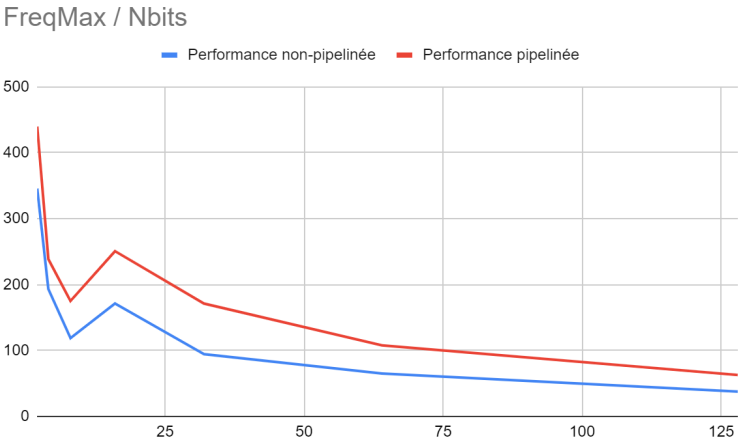
II.2) Unité de calcul - Multiplicateur complexe N bits

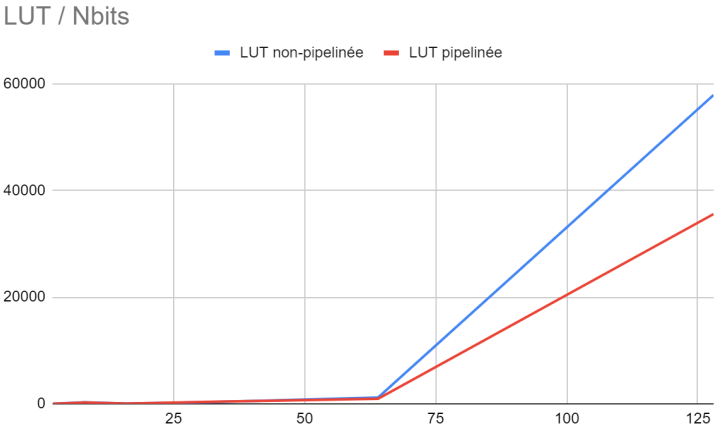
Nous réalisons un testbench, pour voir si notre version pipelinée, nous avons pas décidé de faire de testbench pour la version non-pipelinée, pour les même raisons que l'additionneur :

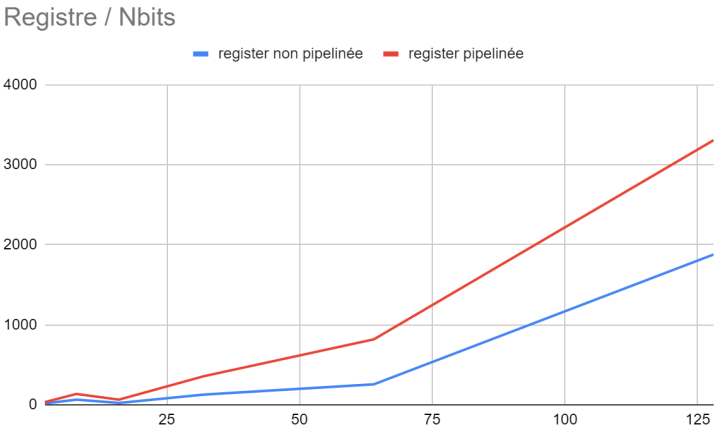


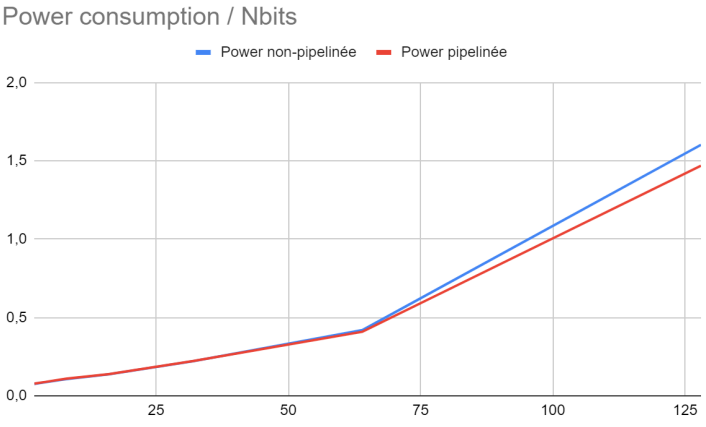
Notre multiplicateur complexe semble marcher. Pour les mêmes raisons que l'additionneur, un état interdit est détecté lorsque la bascule de sortie est vide. Notre multiplicateur semble fonctionner aussi avec les résultats négatifs, en effet ici : $(16+7 \cdot i) \cdot (2+15 \cdot i) = -73+254 \cdot i$ et -73 = 0xffb7 en signé et 254 = 0x00fe.

On réalise le test de performances, même façon que l'additionneur





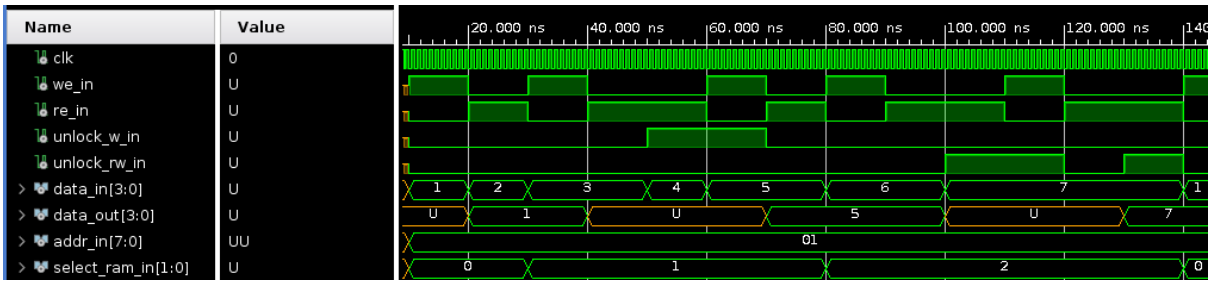




Ici, on peut tirer les mêmes conclusions que l'additionneur, cependant il semble que le multiplicateur soit optimisé pour une utilisation entre 16 et 64 bits : à 16 bits, le multiplicateur utilise moins de matériel et a une fréquence max plus élevée qu'à 8 bits, l'architecture à 16 bits et plus semble plus optimisée. Cependant le nombre de LUT, la puissance et le nombre de registre semble augmenter de façon significative après 64 bits. Mais en supprimant une multiplication, la version pipelinée semble moins consommer que sa version non-pipelinée.

II.3) RAM

Pour tester votre RAM, nous allons faire un testbench sur l'intégralité des possibilités des parties de la RAM :



On commence par la **partie 1** jusqu'à 30 ns :

- A 10 ns : On écrit la valeur "1" dans la RAM à l'adresse 0x0001
- A 20 ns : On lit la valeur de la RAM à l'adresse 0x0001, on a 1 ce qui logique, on en conclut que cette partie fonctionne

Ensuite par la **partie 2** (protection en écriture) jusqu'à 80 ns :

- A 30 ns : On essaye d'écrire la valeur "3" dans la RAM à l'adresse 0x0001 sauf que le blocage est actif
- A 40 ns : On lit la valeur de l'adresse 0x0001, il n'y a rien : ce qui logique puisque nous avons essayé d'écrire avec le blocage
- A 60 ns : On débloque l'accès et on essaye d'écrire la valeur "5"
- A 70 ns: On lit à la même adresse, et on lit bien la valeur "5" cette fois-ci : cette partie semble donc également fonctionner

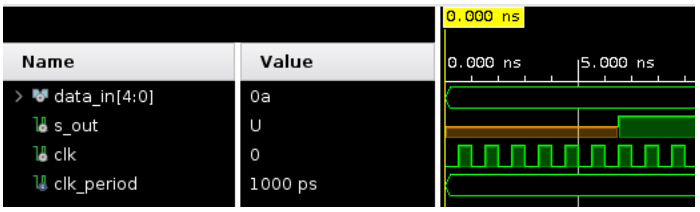
Pour finir, on teste la **partie 3** (protection écriture/lecture) :

- A 80 ns : On essaye d'écrire la valeur "6" dans la RAM à l'adresse 0x0001 sauf que le blocage est actif
- A 90 ns : On essaye de lire la valeur à l'adresse 0x0001 dans la partie 3, comme le blocage est actif, il ne lit rien, donc data_out garde son ancienne valeur ("5")
- A 100 ns : On débloque et on lit, comme nous avons écrit précédemment avec la protection, il n'y a rien dans cette partie de RAM à l'adresse 0x0001 comme prévu
- A 110 ns : On écrit la valeur "7" sans la protection
- A 120 ns ; On essaye de lire la valeur avec la protection ce qui marche pas
- A 130 ns : On lit la valeur sans protection ce qui marche.

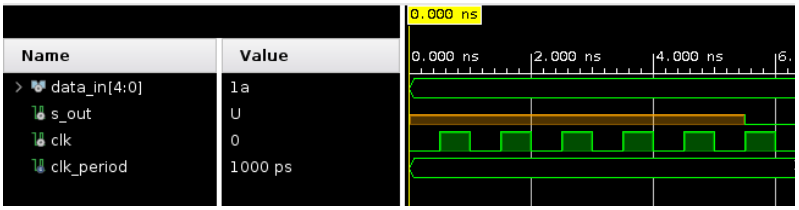
Notre RAM semble ainsi bien fonctionner dans son ensemble.

II.4) Backdoor

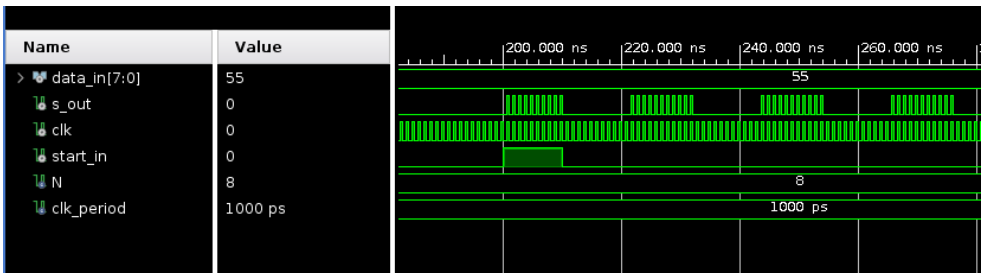
Nous allons dans un premier temps tester notre FSM et l'émetteur OOK séparément.
Pour la FSM, nous avons décider de détecter la séquence "01010" sur les 5 premiers bits des données arrivant à la RAM :



Notre FSM renvoie bien un “1” lorsqu'il détecte la bonne séquence, il prend 6 ticks d’horloges pour parcourir l'intégralité de la séquence. Nous pouvons aussi essayer avec une séquence fausse (voir ci-dessus), il renvoie bien un “0”.

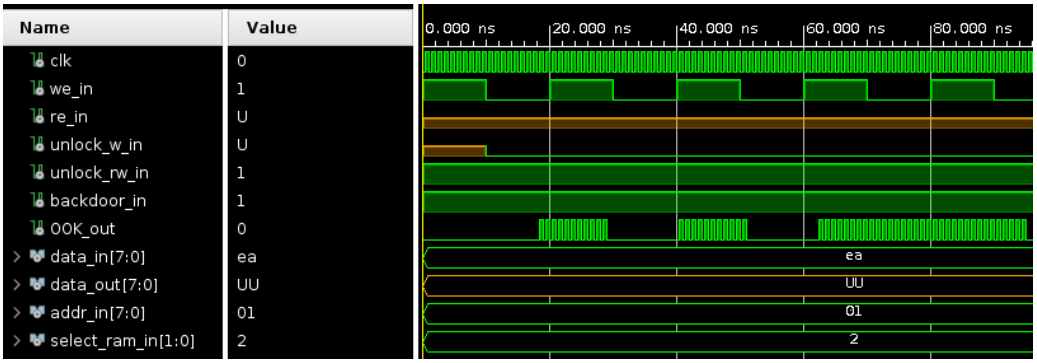


Ensuite on teste l'émetteur OOK, nous mettons en entrée une donnée “0b01010101”, on commence à émettre la LSB puis jusqu’a la MSB, on commence à émettre à t=200ns :

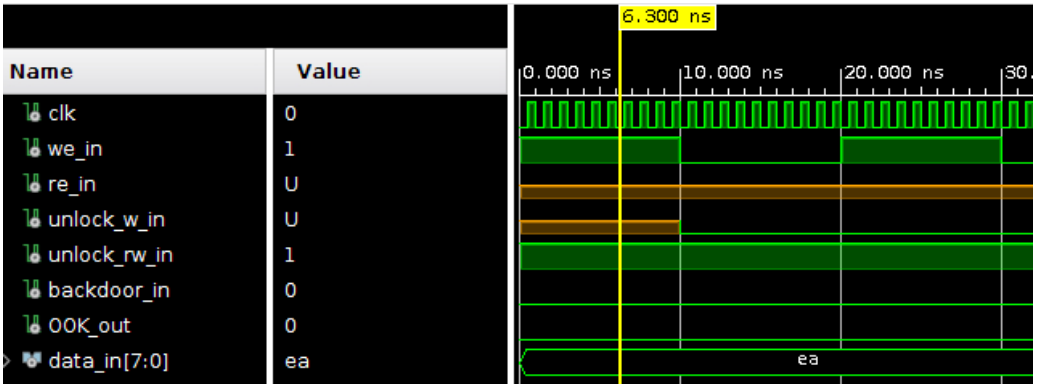


Il émet bien la valeur 10101010 soit : “0b01010101” puisque qu’on commence par la LSB.

Pour finir, on teste la backdoor entière intégrée dans le code de la RAM : On stocke la donnée “0b11101010” dans la RAM sécurisée, et la donnée va ensuite dans l’automate de détection. Nous avons pas mis le signal permettant de savoir si la détection fonctionne, mais l'émetteur commence à émettre après le 6ième tick d’horloge (fin de détection), l'émetteur renvoie bien la bonne valeur provenant de la RAM sécurisée.



Ainsi que lorsque "backdoor in" égale à “0”, l'émetteur n’envoie rien :



Conclusion

Ce TP nous a permis de comprendre en autonomie le fonctionnement avancé du code VHDL (différences entre les signaux et les variables, déclarations de types pour les automates et la mémoire sécurisée et le système de pipeline).

De plus, nous avons utilisé les fonctionnalités avancées de Vivado pour tester et simuler notre code dans des situations réelles (par exemple : le testbench des codes, le rapport de la consommation de puissance, de matériel utilisé et du temps d'exécution).

Nous avons codé l'ensemble des composants utiles pour le projet, cependant nous n'avons pas assemblé tous les composants dans un seul grand schéma VHDL. Ainsi que nous avons rencontré beaucoup de problèmes sur les rapports de tests, notamment celle de la consommation de puissance ainsi que celle du temps d'exécution.