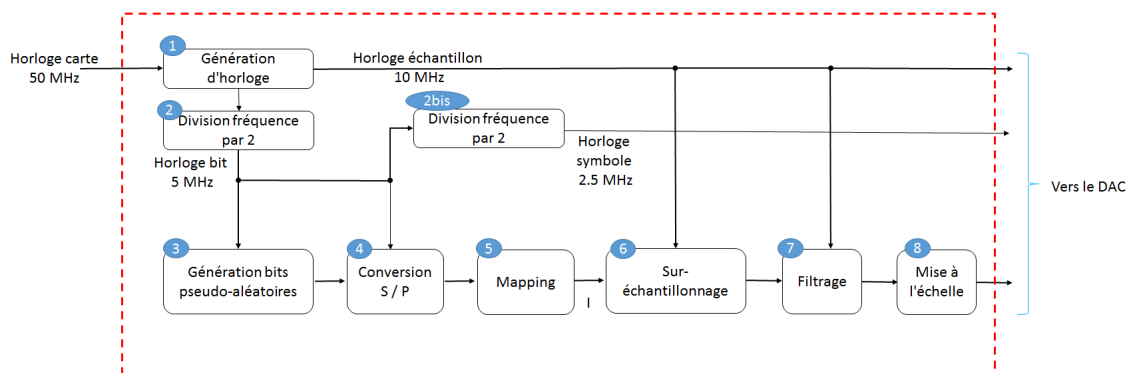


# TP6 : Description VHDL d'un émetteur QPSK bande de base

Le TP se veut un complément au cours et contient beaucoup d'explications. Les choses que vous avez à faire sont repérées par une étoile (☆).

## 1 Découpage fonctionnel et feuille de route

La première étape de la conception consiste à découper le projet en *modules* ; chaque module doit réaliser une tâche et une seule. Le schéma bloc que nous allons utiliser est le suivant :



Lors de la conception d'un circuit, la phase de test est primordiale. En pratique, on teste chaque module séparément à l'aide d'un *testbench*. Vous apprendrez à écrire un testbench basique au cours du TP.

Certains des modules ci-dessus vous seront fournis, et vous n'aurez pas besoin de les tester (on suppose qu'ils ont déjà été validés). D'autres seront écrits et testés par vous.

Le tableau donné en Annexe 2 est une feuille de route que vous pouvez détacher et garder à côté de vous pendant le TP. Elle récapitule les tâches à traiter, dans l'ordre, et vous permet de vous auto-évaluer (la note n'est pas prise en compte, c'est purement pédagogique).

Bien suivre les noms et types de signaux spécifiés dans l'énoncé, pour que les fichiers que vous écrivez et les fichiers fournis soient cohérents.

## 2 Le générateur binaire pseudo-aléatoire

### 2.1 Description VHDL du générateur

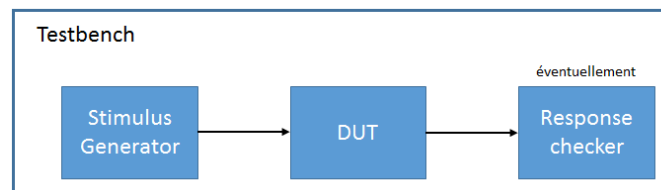
- ☆ Copier sous Moodle le répertoire TP6 et l'extraire dans C:/Users/votrelogin. Lancer Quartus et ouvrir le projet filtre.qpf (File > Open Project). Créer un nouveau fichier VHDL (File > New..., VHDL File), et y copier le code que vous avez préparé. Faire CTRL+S, l'appeler `gen_binaire`, avec l'extension `.vhd`. (Ne pas oublier que le nom du fichier doit être celui de l'entité). Dire que votre fichier est l'entité de plus haut niveau (Project Navigator, onglet Files, clic droit, Set as Top-Level entity). Enfin, lancer l'analyse et synthèse (CTRL+K).

S'il n'y a pas d'erreur de syntaxe, passer à la suite.

## 2.2 Validation par testbench

Le test d'un module se fait à l'aide d'un *testbench*. Il s'agit, pour nous, d'un fichier VHDL qui :

- instancie l'entité à tester ; celle-ci est très souvent appelée *Device Under Test* (DUT) (ou parfois Device Under Verification, DUV, ou encore Unit Under test, UUT) ;
- applique des signaux d'entrée (stimuli) à cette entité ;
- éventuellement vérifie les sorties, et produit une réponse (PASS ou FAIL) ; dans ce cas on parle de self-checking testbench.



- ☆ Créer un nouveau fichier VHDL, et l'enregistrer sous le nom `gen_binaire_tb` dans le répertoire `simulation/modelsim`. Par convention, on utilise le même nom que celui de l'entité à tester, avec le suffixe `_tb` (extension : `.vhd`).

La validation ne sera pas automatisée et se fera par inspection du chronogramme de sortie (waveform) – autrement dit, si l'on reprend le schéma ci-dessus, il n'y aura pas ici de "Response checker". C'est moins efficace, mais d'une part il y a une vertu pédagogique à observer des chronogrammes, et d'autre part, ceci simplifie le code. Saisir ce dernier, lancer l'analyse et synthèse (CTRL+K) (pas besoin de mettre le testbench au top-level) et, s'il n'y a pas d'erreurs, lire les explications qui figurent juste en dessous.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity gen_binaire_tb is
5  end;
6
7  architecture testbench of gen_binaire_tb is
8      signal clock : std_logic := '0';
9      signal register_output : std_logic;
10     constant BIT_CLOCK_PERIOD : time := 200 ns;
11
12  begin
13     dut : entity work.gen_binaire port map(clock => clock, register_output =>
14         register_output);
15
16     clock <= not(clock) after BIT_CLOCK_PERIOD/2;
17  end;
  
```

Les points importants sont :

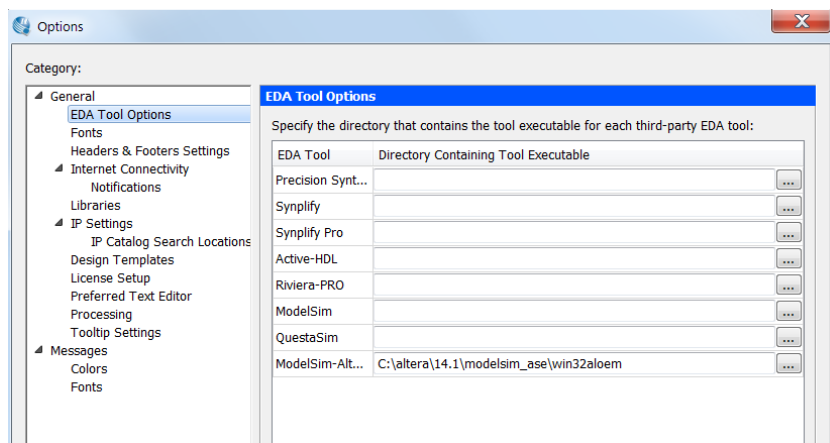
- un testbench est une entité ne possédant ni entrée, ni sortie (lignes 4-5) ;
- le générateur binaire est notre Device Under Test ; il s'agit donc de l'instancier. Mais l'entrée et la sortie de ce dernier n'étant pas des signaux d'entrée/sortie du testbench (puisque ce dernier n'en possède pas), on déclare chacune comme `signal` dans la zone déclarative de l'architecture (lignes 8-9).;

- l'instanciation a lieu à la ligne 13 ; elle connecte les entrées/sorties de `gen_binaire` avec les signaux créés ;
- Le générateur de stimulus doit ici simplement créer un signal d'horloge. Ceci se fait à l'aide d'une *assignation avec retard* : une assignation simple suivie du mot **after** et d'un temps. On ne peut utiliser les assignations avec retard qu'en simulation. Pour que cela génère une horloge, il faut que la valeur initiale de `clock` soit 0 ou 1, d'où l'initialisation de la ligne 8. Ici l'horloge est la seule entrée ; dans le cas général, si le `dut` possède d'autres signaux d'entrée, on les génère dans un certain ordre en utilisant un process, comme nous le verrons un peu plus loin.

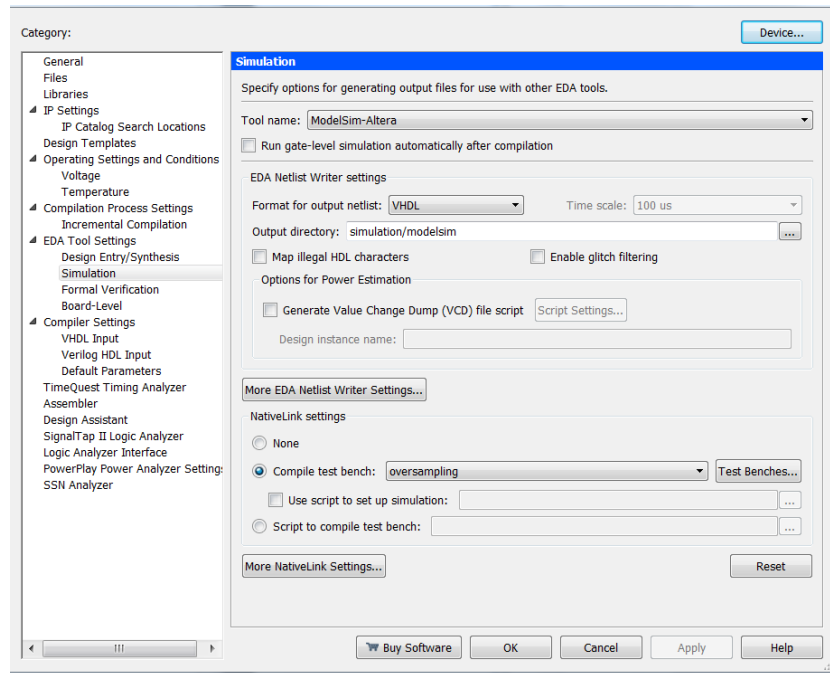
Les testbenches ne sont pas supportés par Quartus ; on utilise pour cela un autre logiciel, **Modelsim-Altera**. Il est toutefois possible de lancer directement ce dernier depuis Quartus ; un certain nombres de tâches sont alors automatisées (création d'un projet Modelsim-Altera, ajout de fichiers, compilation, etc).

☆ Pour cela réaliser les 3 opérations décrites ci-dessous :

1. dans **Tools > Options...**, choisir dans **Category** (barre de gauche) **EDA Tool Options**. Puis, pour Modelsim-Altera, ajouter le chemin de l'exécutable comme dans la capture ci-dessous ;



2. dans **Assignments > Settings...**, choisir dans **Category** (barre de gauche) **EDA Tool Settings** et **Simulation**. Dans **Tool Name**, vérifier que Modelsim-Altera est sélectionné et que le **Format for output netlist** est VHDL ; dans **Output Directory**, il doit y avoir `simulation/modelsim`.



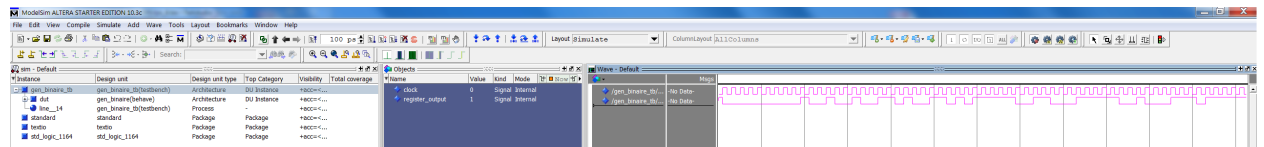
### 3. dans Native Link Settings :

- cocher **Compile test bench** ;
- cliquer sur **Test benches...** La fenêtre de dialogue **Test Benches** apparaît ;
- cliquer sur **New...** La fenêtre de dialogue **New Test Bench Settings** apparaît ;
- le **Test bench name** peut être le nom de votre choix (ce nom permettra de choisir rapidement le testbench à effectuer), par exemple **gen\_binaire\_tb** ;
- le **Top level module in test bench** doit être le nom de l'entité de votre testbench : à priori **gen\_binaire\_tb** ;
- cocher **Use test bench to perform VHDL timing simulation** et comme **Design instance name in test bench**, entrer le nom utilisé lors de l'instanciation de votre module (ci-dessus : **dut**, ligne 13) ;
- dans **Simulation period**, cocher **End simulation at** et donner par exemple **10  $\mu$ s** ;
- enfin, dans **Test bench and simulation files**, sélectionner votre fichier testbench. Cliquer sur **Add**, sur **OK**, et dans la fenêtre **Test Benches**, sélectionner le testbench que vous venez de créer. Cliquer sur **OK** deux fois.

La troisième étape est à refaire pour chaque nouveau testbench (rappelez-vous qu'on crée un testbench pour chaque module du projet).

☆ Une fois cela fait, la simulation sous ModelSim-Altera se lance directement à partir du menu **Tools > Run Simulation Tool > RTL Simulation**, ou en cliquant sur l'icône correspondante dans la barre d'outils :

Le programme doit s'ouvrir. Si tout va bien, à droite, dans la fenêtre **Wave**, vous aurez quelque chose : les chronogrammes des entrées et sorties de votre **dut**. En cliquant dans la fenêtre, vous pouvez zoomer (**Shift+I**) ou dézoomer (**Shift+O**). En dézoomant, vous obtiendrez :

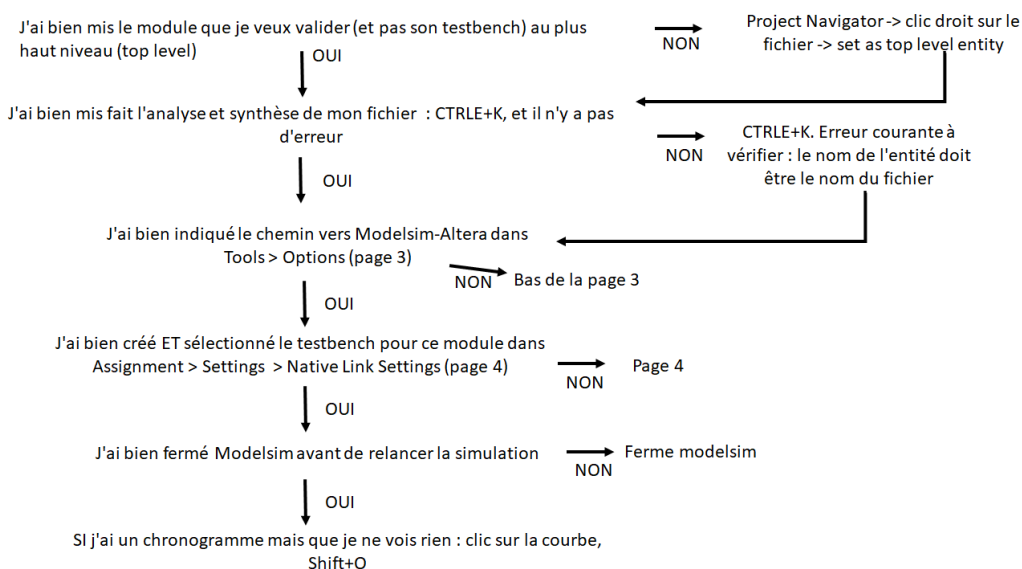


Normalement les premiers bits sont :



1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1. Le vérifier. Si c'est bien ce que vous obtenez, vous pouvez considérer que votre module 3 est validé et passer à la suite.

### Guide de débannage : à lire avant d'appeler l'enseignant !

Si vous n'avez pas de chronogrammes, c'est qu'un message d'erreur s'est affiché en bas dans le log (**Transcript**) : agrandissez la fenêtre du bas pour le voir. Ce peut être un problème de VHDL si vous n'avez pas fait le **CTRL+K** avec Quartus ; mais, le plus souvent, c'est plutôt un souci de configuration du testbench ou du lien entre Quartus et Modelsim. Si vous ne parvenez pas à résoudre le souci avec le message fourni par Modelsim, suivre le guide de débannage ci-dessous :



Si vous devez modifier votre fichier `gen_binaire` :

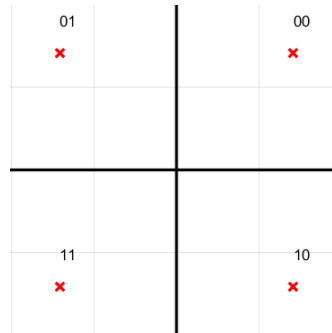
- penser à relancer l'analyse et synthèse (**CTRL+K**) ;
- fermer Altera-Modelsim avant de cliquer sur  sinon vous aurez un message d'erreur ; ou, encore mieux, ne pas fermer Altera-Modelsim et ne pas cliquer sur  ; à la place, sous Altera-Modelsim, taper la commande : `do filtre_run_msim_rtl_vhdl.do`. Ceci permet de relancer la série d'opérations qu'avait faite modelsim à son ouverture<sup>1</sup>

Dans tous les cas, fermer Altera-Modelsim une fois le module validé.

1. le fichier `filtre_run_msim_rtl_vhdl.do` a été créé par Quartus et se situe dans `simulation/modelsim` ; vous pouvez observer son contenu avec un éditeur de texte.

### 3 Le mapping

Le bloc de mapping possède une entrée qui est le couple de bits ; et normalement deux sorties, I et Q. Dans ce TP, nous ne nous intéresserons qu'à I, donc nous n'aurons qu'une seule sortie. Nous pouvons utiliser la même constellation que Matlab/simulink, en donnant comme valeur  $\pm 1$  à I plutôt que  $\pm\sqrt{2}/2$  :



- ☆ Créer un nouveau fichier VHDL appelé `mapping.vhd`, le définir comme `Top-level entity`, et saisir le code permettant de décrire son comportement (indice : c'est un simple circuit combinatoire  $\rightarrow$  revoir le décodeur binaire - 7 segments du cours). Utiliser une entrée que vous appellerez `bit_couple` de type `unsigned` sur 2 bits, et une sortie appelée I de type `integer`, en restreignant les valeurs à l'intervalle  $[-1, +1]$  (indice : regarder dans le cours comment déclarer des signaux utilisant ces types et quel package est nécessaire à leur utilisation). Faire l'analyse et synthèse (`CTRL+K`). S'il n'y a pas d'erreurs, créer un second fichier pour le testbench, l'enregistrer sous `simulation/modelsim` et l'appeler `mapping_tb.vhd`.

- ☆ Saisir le code suivant pour le testbench :

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity mapping_tb is
6  end;
7
8  architecture testbench of mapping_tb is
9      constant DELTA_TIME : time := 20 ns;
10     signal bit_couple : unsigned(1 downto 0);
11     signal I : integer range -1 to 1;
12
13 begin
14     dut : entity work.mapping port map(bit_couple => bit_couple, I => I);
15
16     process
17     begin
18
19
20         wait for DELTA_TIME;
21
22         bit_couple <= "00";
23
24         wait for DELTA_TIME;
25
26         bit_couple <= "01";
27

```

```


28     wait for DELTA_TIME;
29
30     bit_couple <= "10";
31
32     wait for DELTA_TIME;
33
34     bit_couple <= "11";
35
36     wait;
37
38 end process;
39 end;

```

Le début du testbench est analogue au précédent : l'entité est déclarée sans entrée/sortie (lignes 5-6), des signaux `bit_couple` et `I` sont déclarés (ligne 10-11) pour être connectés aux entrées/sorties du dut que l'on instancie (instanciation à la ligne 14).

On vérifie alors successivement toutes les entrées possibles du circuit (ici c'est très rapide car il n'y en a que quatre). *La seule façon d'appliquer ces entrées en séquence est d'utiliser un process*<sup>2</sup>. Avant de changer l'entrée, on attend `DELTA_TIME` pour que la sortie ait le temps de changer en conséquence (en pratique, ce temps est de l'ordre de quelques ns à quelques dizaines de ns selon le circuit combinatoire).

Lorsqu'on arrive à la dernière instruction d'un process, l'exécution reprend au début de celui-ci. Si l'on ne fait rien, les mêmes entrées seront appliquées de nouveau au dut. La dernière instruction `wait` (ligne 36) permet de suspendre le process indéfiniment pour que chaque entrée ne soit testée qu'une fois.

- ☆ Lancer l'analyse et synthèse (CTRL+K). Il faut ensuite retourner dans **Assignment > Settings** pour y créer un nouveau testbench (étape 3 en bas de la page 4). Fermer Altera-Modelsim. Le relancer ( **Tools > Run Simulation Tool > RTL Simulation**, ou  )

Vérifier les sorties.

## 4 Le filtre

L'implantation d'un filtre en cosinus surélevé nécessite de suréchantillonner le signal, puisqu'on passe de 1 échantillon par temps symbole à `OSR` échantillons (`OSR` : Over Sampling Ratio). En pratique, on a deux étapes :

- le suréchantillonnage, qui consiste à insérer `OSR-1` échantillons '0' après chaque échantillon d'entrée ;
- le filtrage proprement dit, c'est-à-dire, pour notre filtre FIR, l'implantation de l'opération de convolution.

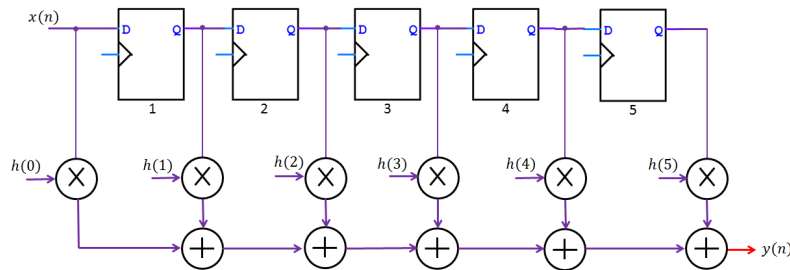
Le fichier VHDL décrivant le suréchantillonnage est fourni ; vous avez à compléter le filtre. On rappelle que l'échantillon de sortie est calculé de la façon suivante :

$$y(n) = \sum_{k=0}^{M-1} h(k) x(n-k)$$

2. On rappelle que les instructions séquentielles qu'on trouve dans un process sont exécutées, précisément, de manière séquentielle – les unes à la suite des autres – à la différence des instructions concurrentes qui s'exécutent toutes en même temps.

où les  $h(k)$  sont les coefficients du filtre.

La *forme directe* permettant d'implémenter le filtre est représentée ci-dessous, pour  $M = 5$  (on rappelle que pour nous il y a 33 coefficients, donc  $M = 32$ ).

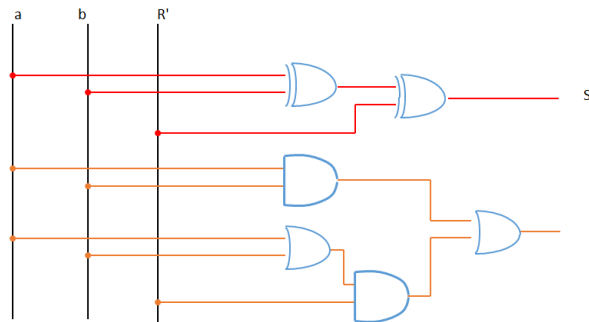


On n'a pas représenté l'horloge pour alléger le schéma (il s'agit bien sûr de l'horloge rythmant l'arrivée des échantillons, pour nous à 10 MHz).

Il y a une certaine ressemblance avec les schémas du registre à décalage et du LFSR ; le code VHDL de l'ensemble des bascules pourra être similaire à celui du registre à décalage, en changeant de type de signaux.<sup>3</sup>

Dans ce schéma, l'ensemble des multiplieurs et additionneurs est implanté sous forme combinatoire.

Cependant, cela pose un petit problème. Pour le comprendre, considérons un bloc combinatoire formé de portes logiques : pour faire simple, un additionneur 1 bit<sup>4</sup>. Un tel circuit possède 3 entrées : les deux bits à additionner  $a$  et  $b$  et une retenue entrante  $R'$  ; et deux sorties : la somme  $S$  et la retenue  $R$ . L'implantation se fait à l'aide de 6 portes logiques :



Quand une entrée change, la sortie change *presque immédiatement*, au temps de propagation des portes près. Ce temps de propagation est, pour une porte logique, de l'ordre de la nanoseconde. Mettons qu'il soit de 1 ns, la sortie  $S$  va mettre 2 ns à se stabiliser après évolution d'une entrée, et la sortie  $R$ , 3 ns, puisque pour chacune, le plus long chemin compte respectivement 2 et 3 portes. Le plus long chemin entre une entrée et une sortie d'un circuit combinatoire est appelé *chemin critique* : ici, il s'agit du chemin entre  $a$  ou  $b$  et  $R$ .

Revenons à présent à notre filtre. Au front d'horloge, tous les échantillons  $x(k)$  évoluent en même temps. A cause du grand nombre d'additionneurs et de multiplieurs, la sortie  $y$  va mettre un temps non négligeable (et difficile à évaluer a priori) à se stabiliser. Pour l'étage suivant, ce peut être

3. Toutefois, il faut avoir en tête que  $x(n)$  et les sorties des bascules représentées ici sont en fait les échantillons  $x(k)$ , codés sur plusieurs bits ; on aura donc, dans l'implantation physique, plusieurs fils parallèles et plusieurs bascules.

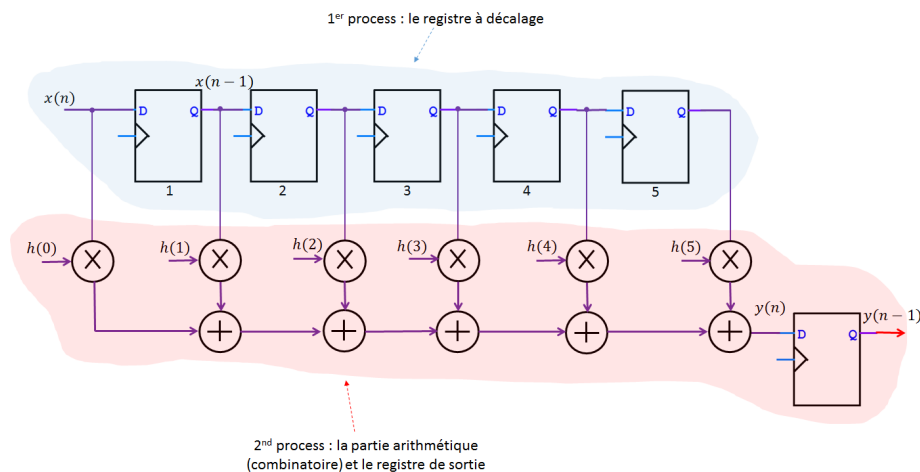
4. Chacun des additionneurs du filtre est implanté à l'aide de plusieurs additionneurs 1 bit comme celui que nous allons décrire.



gênant (en l'occurrence, pour nous, c'est gênant). C'est pourquoi on préfère synchroniser la sortie, c'est-à-dire la faire passer par un registre. On parle de *sortie synchronisée* ou *registered output*.

Noter que le signal de sortie doit s'être stabilisé lorsqu'intervient le prochain front d'horloge, sans quoi le signal écrit dans le registre de sortie sera erroné. Cet aspect est fondamental en pratique : *le chemin critique détermine la fréquence maximale de fonctionnement*.

Pour indiquer que la sortie doit évoluer sur front d'horloge, en VHDL, on utilise un process, avec uniquement l'horloge dans la liste de sensibilité (voir préparation). La bonne pratique consistant à donner une seule tâche à un process, on en comptera deux dans l'architecture :



☆ Ouvrir le fichier `filtrage.vhd`, l'ajouter au projet (ceci se fait dans **Assignment > Settings**, puis choisir **Files** dans **Category**) et le déclarer comme **Top-level entity**. Vous devez obtenir le code VHDL incomplet ci-dessous. Commencer par lire la déclaration des entrées/sorties.

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity filtrage is
5      port(
6          clock : in std_logic;
7          x0 : in integer range -1 to 1;
8          y : out integer range -8192 to 8191 := 0
9      );
10 end;
11
12 architecture behave of filtrage is
13
14     constant NUMBER_OF_COEFFFS : natural := 33;
15
16     type sample_type is array (0 to NUMBER_OF_COEFFFS-1) of integer range -1 to 1;
17     type coeff_type is array (0 to NUMBER_OF_COEFFFS-1) of integer range -16384 to 16383;
18
19
20     constant H : coeff_type := (
21         0, -1, -21, -36, 0, 101, 213, 213, -1, -396, -749, -710, 0, 1383, 3077,
22         4477, 5019, 4477, 3077, 1383, 0, -710, -749, -396, -1, 213, 213, 101,
23         0, -36, -21, -1, 0 );
24
25     signal x : sample_type := (others => 0);


```

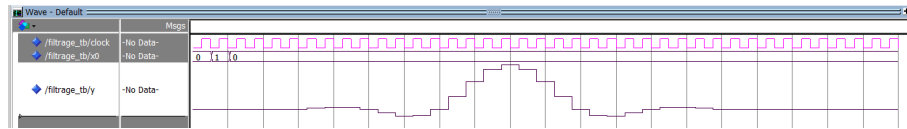
```

24
25 begin
26
27     x(0) <= x0;
28
29     shift : process(clock)
30     begin
31         if falling_edge(clock) then
32             x(1 to NUMBER_OF_COEFFS-1) <= x (0 to NUMBER_OF_COEFFS-2);
33         end if;
34     end process;
35
36
37     multacc: process(clock)
38         variable sum : integer range -16384 to 16383 := 0;
39         variable s : coeff_type := (others => 0);
40     begin
41         if falling_edge(clock) then
42
43
44
45
46
47
48         end if;
49     end process;
50 end;
```

Dans l'architecture, les points nouveaux ou importants sont les suivants :

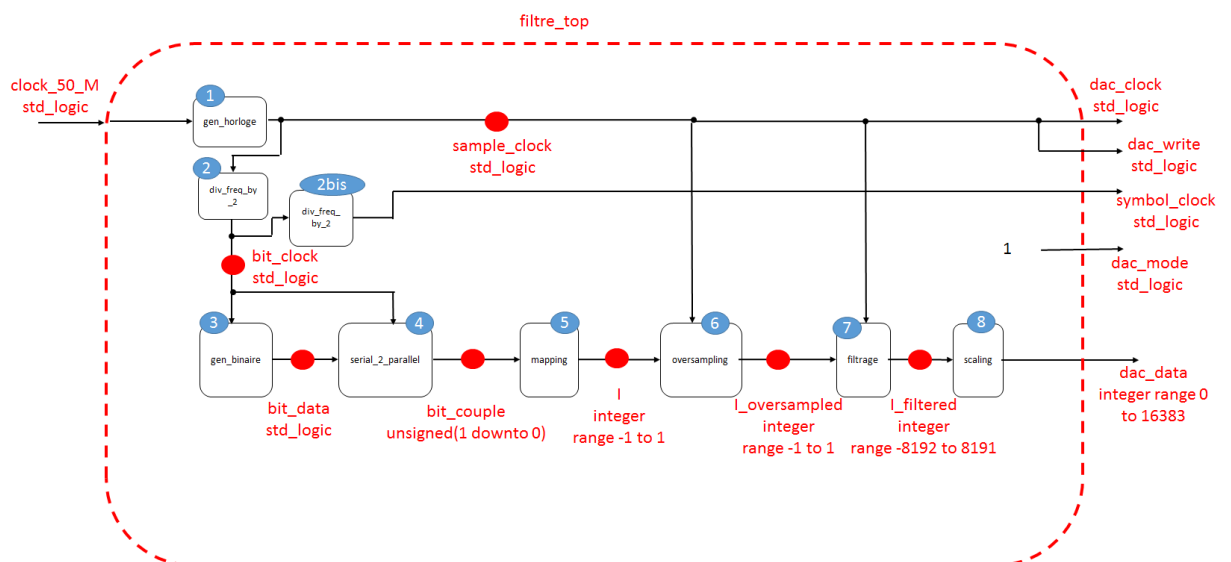
- aux lignes 16 et 17, on a déclaré deux types appelés `sample_type` et `coeff_type` afin de pouvoir stocker les échantillons  $x(k)$  et les coefficients  $h(k)$  comme des vecteurs d'entiers (array). Ces vecteurs sont déclarés aux lignes 20 à 23. L'intérêt d'utiliser des vecteurs est qu'on pourra accéder aux échantillons et aux coefficients en utilisant un index dans une boucle;
- à la ligne 23, `(others => 0)` permet d'initialiser les composantes de `x` à 0 quel que soit leur nombre. Cette construction est appelée *agrégat*; elle est plus simple, plus lisible et moins sujette aux erreurs que `(0, 0, 0, 0, ..., 0)` (les 33 composantes séparées par des virgules). Autre exemple : si l'on voulait que la composante d'index 10 soit à 1 et tous les autres à 0, on écrirait `(10 => 1, others => 0)`;
- dans l'architecture, on a deux process. Noter qu'on a donné un label (optionnel) à chaque process pour préciser leur rôle (shift register ou multiplication accumulation) sans avoir besoin de commentaire supplémentaire (lignes 29 et 37);
- ces deux process sont ici sensibles aux fronts descendants. Ceci est lié au fonctionnement du DAC;
- aux lignes 38 et 39, on a déclaré deux variables : `s` (array d'entiers) et `sum`. La variable `s` n'est pas utile (elle est restée à des fins de debug); utiliser la variable `sum` pour calculer la somme de tous les produits. Compléter les lignes 42 à 47 pour faire cela, en utilisant une boucle `for...loop` (rappel : cette instruction n'est utilisable que dans les process). Penser à initialiser `sum` à 0 au début de la boucle. A la sortie de la boucle, y doit prendre la valeur de `sum`.

- ☆ Lancer l'analyse et synthèse (CTRL+K). Puis ouvrir et ajouter au projet le fichier `filtrage_tb`, fourni. Le parcourir, puis créer un testbench correspondant dans **Assignment > Settings**, avec comme durée 4  $\mu$ s. Cliquer sur . Dans la fenêtre **Wave** obtenue, faire un clic droit sur le signal `y` et choisir **Format > Analog (automatic)**. Vous devez obtenir la réponse impulsionnelle du filtre pour que cette partie soit validée :



## 5 Fichier top level

- ☆ Ajouter au projet les fichiers correspondant aux modules fournis : `div_freq_by_2.vhd`, `serial_2_parallel.vhd`, `oversampling.vhd`, `scaling.vhd`, `gen_horloge.vhd` (il n'est pas nécessaire de les ouvrir). Ajouter au projet, déclarer comme **Top-level entity** et ouvrir le fichier `filtre_top.vhd`.
- ☆ Compléter le code VHDL en utilisant le schéma ci-dessous, pour que tous les modules soient instanciés. Les sorties `dac_clock`, `dac_write` et `dac_mode` sont nécessaires pour la commande du DAC ; la sortie `symbol_clock` pour la visualisation du diagramme de l'oeil. Les noeuds repérés par de gros points sont des signaux déclarés dans l'architecture, à utiliser pour l'instanciation.



Ci-dessous, on donne pour chacun des 8 modules la déclaration de l'entité (qu'il faut avoir sous les yeux pour l'instanciation) :

```

-- module 1 : gen_horloge.vhd
1      entity gen_horloge is
2
3      generic (
4      FREQ : frequency_type := freq_10_MHz

```

```
5         );
6
7         port(
8             clock : in std_logic;
9             pulse : out std_logic
10        );
11    end;
```

Le type `frequency_type` a été déclaré dans un package appelé `my_type` (présent dans le fichier `gen_horloge.vhd`), qu'il faudra inclure dans votre fichier top level en ajoutant au début la ligne :

```
1         use work.my_type.all;
```

La définition de ce type est :

```
1         type frequency_type is (freq_1_MHz, freq_2_MHz, freq_5_MHz,
                                freq_10_MHz, freq_25_MHz);
```

Autrement dit, quand on instancie l'entité `gen_horloge`, on lui passe un paramètre (`generic`) qui doit être l'une des 5 valeurs qui composent le type de `FREQ` : `freq_1_MHz`, ou `freq_2_MHz`, etc. On peut ainsi utiliser cette entité pour générer des horloges de différentes fréquences (revoir le schéma de la page 1 si vous ne voyez pas quelle valeur lui donner ici).

— module 2 : `div_freq_by_2.vhd`

```
1         entity div_freq_by_2 is
2             port(
3                 clock : in std_logic;
4                 clock_divided : out std_logic
5             );
6         end;
```

```

-- module 3 : gen_binaire.vhd
1      entity gen_binaire is
2          port (
3              clock : in std_logic;
4              register_output : out std_logic
5          );
6      end;

-- module 4 : serial_2_parallel.vhd
1      entity serial_2_parallel is
2          port (
3              serial_bit, clock : in std_logic;
4              parallel_bit : out unsigned(1 downto 0)
5          );
6      end;


-- module 5 : mapping.vhd
1      entity mapping is
2          port(
3              bit_couple : in unsigned(1 downto 0);
4              I : out integer range -1 to 1
5          );
6      end;

-- module 6 : oversampling.vhd
1      entity oversampling is
2          port(
3              clock_sample : in std_logic;
4              symbol : in integer range -1 to 1;
5              symbol_oversampled : out integer range -1 to 1
6          );
7      end;

-- module 7 : filtrage.vhd
1      entity filtrage is
2          port(
3              clock : in std_logic;
4              x0 : in integer range -1 to 1;
5              y : out integer range -8192 to 8191 := 0
6          );
7      end;

-- module 8 : scaling.vhd
1      entity scaling is
2          port(
3              centered_y : in integer range -8192 to 8191 := 0;
4              positive_y : out integer range 0 to 16383
5          );
6      end;

```

- ☆ Ouvrir et ajouter au projet le fichier `filtre_top_tb`, fourni. Le parcourir, puis créer un testbench correspondant dans **Assignment > Settings**, avec comme durée 50  $\mu$ s. Cliquer sur , et observer l'allure des sorties `dac_clock` et `dac_data`. Vérifier que les data changent sur le front descendant de `dac_clock`. Une fois validé le testbench, passer à l'implantation dans le FPGA.

## 6 Implantation dans le FPGA et connexion au DAC

☆ Implanter le circuit dans le FPGA. Pour cela, on rappelle qu'il faut :

- assigner les entrées/sorties sur les pins adéquates. Pour l'horloge à 50 MHz, on aurait utilisé la pin R20 ; pour les sorties, voir l'Annexe 1 ;
- lancer une compilation totale, c'est-à-dire incluant les étapes **Fitter** et **Assembler**. Pour cela, **Processing > Start Compilation** ou **CTRL+L** ;
- charger le programme dans le FPGA : connecter la carte par USB (connecteur **USB Blaster**), mettre sous tension la carte, et aller dans le menu **Tools > Programmer**. Vérifier que **USB-Blaster** apparaît en haut (sinon le sélectionner dans **Hardware Setup...**), ajouter votre fichier **filtre.sof** (**Add Files...**, dans **output\_files**) et cliquer sur **Start**.

☆ Connecter la carte fille, qui contient le DAC (voir Annexe 1). La sortie du filtre est disponible sur le connecteur **SMA DA-Channel A**. Utiliser un adaptateur **SMA - BNC** (voir Annexe 1).

## 7 Observation des signaux

☆ Visualiser sur l'oscilloscope (vérifier sa référence : Keysight DSOX1102G) la sortie du filtre, et également l'horloge symbole (voir Annexe 1). On vous donne les réglages à effectuer :

- horloge symbole sur la voie 1, sortie du filtre sur la voie 2 ;
- base de temps :  $1\mu s/div$  ; échelle verticale :  $2V/div$  sur les deux voies ;
- trigger sur la voie 1 ;
- bouton **Single** (sous **Run/Stop**) pour observer les signaux.

A quoi sont dues les marches d'escalier ?

Sur la sortie I filtrée, essayer de repérer un enchaînement de symboles  $1 \rightarrow -1 \rightarrow 1$  (rappelez-vous que l'horloge symbole est à 2.5 MHz, soit une période de 400 ns).

☆ Repasser en mode **Run** (bouton **Run/Stop**). Observer le spectre du signal I filtré. Pour cela : bouton **FFT** de l'oscilloscope ; **span** : 50 MHz ; fréquence centrale : 25 Mhz.

Quelle est la partie utile du spectre ? La bande occupée est-elle la bande prévue ?

A quoi sont dues les répliques ? Justifier les valeurs des fréquences correspondantes.

☆ On peut appliquer un filtre passe-bas sur le signal appliqué à l'entrée de l'oscillo. Pour cela :

- faire disparaître le spectre (bouton **FFT**) ;
- base de temps : 200 ns/div
- bouton **Math, Operator** : **Low Pass filter**, **Source** : 2.  
Une nouvelle trace (violet) apparaît (à ce moment là, vous pouvez faire disparaître la trace de la voie 2 pour mieux voir). Il reste à régler la bande passante du filtre : **Bandwidth** : 4MHz ; et l'échelle verticale de notre nouvelle trace : **scale** :  $2V/div$ .

- ☆ Les signaux étant toujours synchronisés par l'horloge symbole de la voie 1, il reste, pour observer le diagramme de l'oeil, à superposer les traces sur l'écran. Pour cela : bouton **Display**, **Persistence**,  $\infty$ .

## 8 Connaissances, Capacités

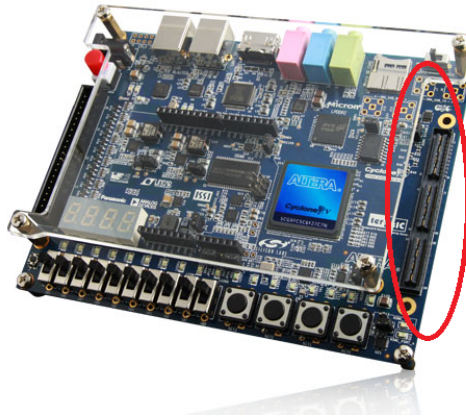
Une connaissance est un concept théorique (qu'on pourrait parfois chercher sur Wikipedia). Une capacité est associée à un verbe d'action : il s'agit de savoir faire quelque chose. Vérifier que vous avez acquis les connaissances et capacités visées.

Connaissance (Co)/ Capacité (Ca)	Acquise ?
Co : Principe du générateur pseudo-aléatoire à LFSR	
Co : VHDL : instructions concurrentes, process, instructions séquentielles, signal vs variables	
Co : Hardware correspondant à un process possédant seulement l'horloge dans sa liste de sensibilité	
Co : Hardware correspondant à un process possédant toutes les entrées dans sa liste de sensibilité	
Ca : VHDL : Décrire une fonction combinatoire basique	
Ca : VHDL : Décrire un circuit séq. simple ou compléter la description d'un circuit plus complexe	
Ca : VHDL : savoir instancier des modules dans un fichier top-level	
Ca : Utiliser Quartus pour gérer un projet simple (création, gestion fichiers, compil., testbench, etc)	
Co : Filtre en cosinus surélevé, Interférences entre symboles	
Ca : Observer un diagramme de l'oeil sur oscilloscope	

## Annexe 1 : La carte THD ADA et le DAC9767

### Description succincte

On utilise une carte fille (THD ADA) que l'on connecte à la carte FPGA au niveau du connecteur situé à droite (il y a un détrompeur donc pas d'erreur possible) :



La carte fille est alimentée par une tension de 3,3 Volts qui est ici fournie par la carte FPGA.

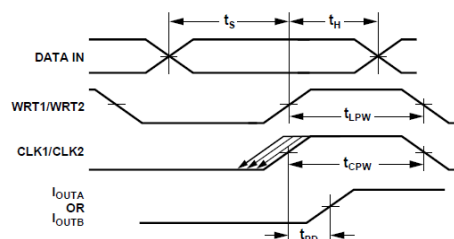
Elle contient deux convertisseurs : un ADC et un DAC. Chacun peut traiter deux signaux simultanément (deux canaux).

Le DAC, qui nous intéresse ici, est le AD9767 (les schémas ci-dessous sont extraits de sa documentation). Il a les caractéristiques suivantes pour chaque canal :

- résolution : 14 bits ;
- fréquence maximale : 125 Msamples/s.

Deux modes de fonctionnement sont possibles : dual port ou interleaved. Le choix se fait à l'aide d'une entrée `mode` sur 1 bit. La différence concerne les bits d'entrée des deux canaux, qui peuvent être présentés soit séparément (dual port si `mode` = 1) ou alternativement sur un seul port d'entrée (interleaved si `mode` = 0). Nous choisirons le mode dual port, et nous utiliserons un seul canal sur les deux.

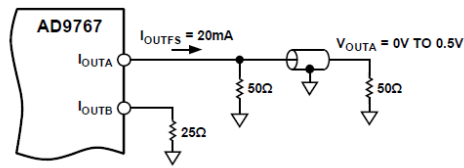
Avec le mode dual port, deux autres entrées de commande sont nécessaires pour chaque canal : CLK et WRT. Ces deux signaux peuvent être identiques, ou WRT peut avoir un léger retard sur CLK. Nous les prendrons identiques. Il est fortement recommandé que les DATA IN changent sur leur front descendant.



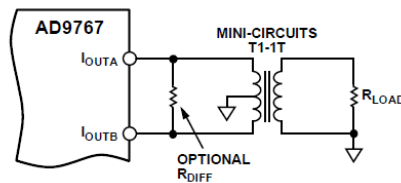
Remarquer que la sortie est un courant (proportionnel au mot binaire). Pour le transformer en une tension référencée par rapport à la masse, plusieurs configurations de sortie sont envisageables :



1. avec une simple résistance :



2. avec une transformateur : la linéarité est meilleure, mais les fréquences basses sont coupées :



C'est la seconde configuration qu'on trouve sur les cartes fille ; mais comme notre signal occupe un spectre centré sur 0 Hz, nous avons modifié celles que nous utilisons ; pour aboutir à la première.

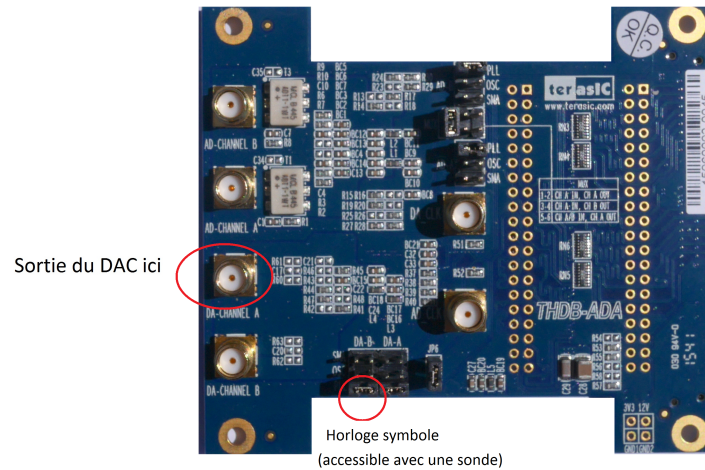
## Assignation des sorties du FPGA pour la commande du DAC

Utiliser pour cela le tableau ci-dessous :

Sortie	Pin Number
clock_50_M	R20
dac_mode	A21
dac_clock	A18
dac_write	B22
dac_data(13)	C15
dac_data(12)	B15
dac_data(11)	B19
dac_data(10)	C20
dac_data(9)	A11
dac_data(8)	B10
dac_data(7)	B11
dac_data(6)	A12
dac_data(5)	C10
dac_data(4)	D10
dac_data(3)	B9
dac_data(2)	C9
dac_data(1)	E11
dac_data(0)	E10
symbol_clock	A19

## Accès aux signaux

Les deux signaux utiles sont la sortie du DAC, accessible au niveau du connecteur DA CHANNEL A ; et l'horloge symbole. La pin A19 utilisée pour cette dernière est normalement l'horloge du second canal du DAC, ce qui permet de la récupérer au niveau d'un cavalier (jumper), comme le montre le schéma ci-dessous.



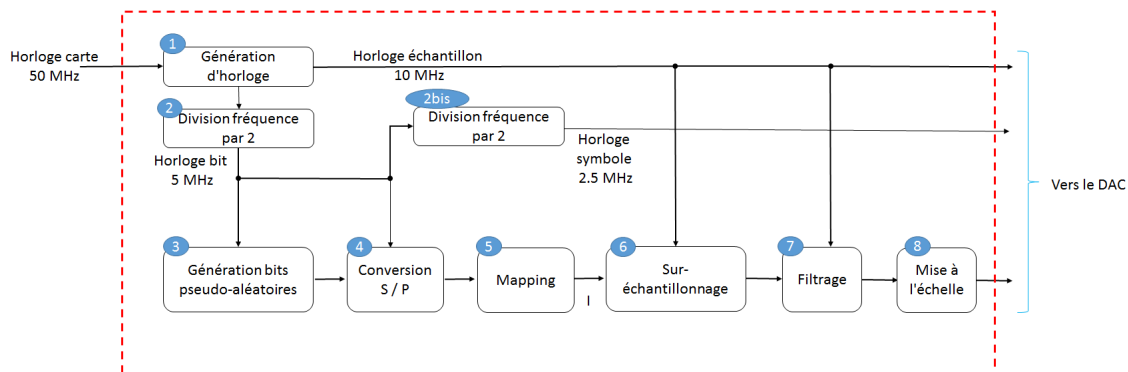
## Annexe 2 : Feuille de route

### Timing recommandé pour terminer le TP

14h -> 14h 45 : générateur binaire  
 14h45 -> 15h20 : mapping  
 15h20 -> 16h40 (incluant la pause de 20 minutes) : filtre  
 16h40 -> 17h15 : fichier top level  
 17h15 -> 18h : implantation & observation des signaux

### Auto-évaluation

Une tâche correspondant à l'écriture d'un module est validée en même temps que son testbench si celui-ci est un succès (c'est-à-dire si les chronogrammes de sortie vérifient les spécifications). Il y a en tout 5 tâches à valider (les testbenches tutorial ne donnent pas lieu à validation).



<i>n°</i>	Tâche	Modalité	Validation (O / N)
	Écriture Module (1)	Fourni	
	Écriture Module (2)	Fourni	
1	Écriture module (3)	A faire (en préparation)	
	Testbench module (3)	A faire (tutorial)	
	Écriture module (4)	Fourni	
2	Écriture module (5)	A faire	
	Testbench module (5)	A faire (tutorial)	
	Écriture module (6)	Fourni	
3	Écriture module (7)	A compléter	
	Testbench module (7)	Fourni	
	Écriture Module (8)	Fourni	
4	Écriture top_level_file	A compléter	
	Testbench top_level_file	Fourni	
5	Analyse des signaux	A faire	

**Votre niveau :** 0 tâches validées : D ; 1 ou 2 tâches validées : C ; 3 ou 4 tâches validées : B ; 5 tâches validées : A.