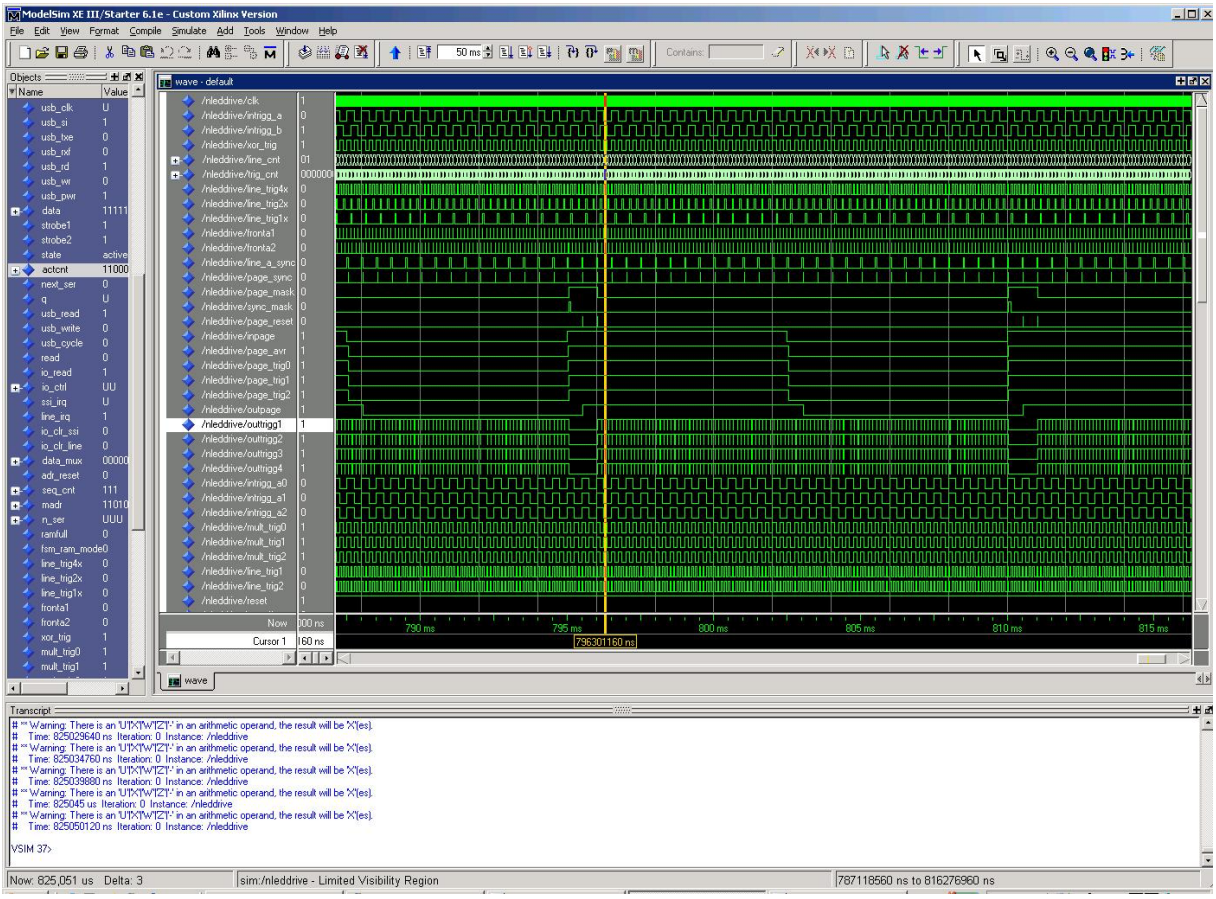
VHDL : Exercices sur Modelsim



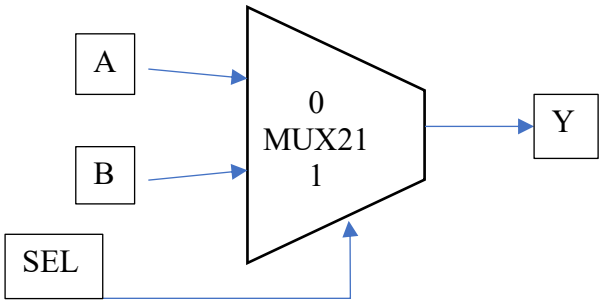
Exercices du C3

Créer un répertoire **/VHDL/Exercices/** et copier les sources « **ExoC3 - Sources.zip** »

Dézipper le répertoire **ExoC3 - Sources.zip**

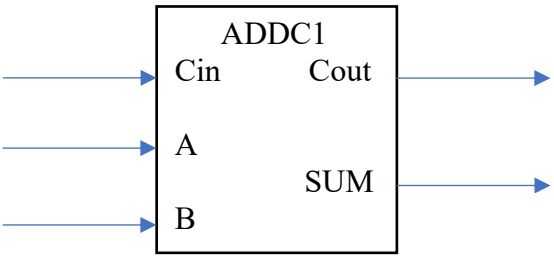
Exercice 1 : Multiplexeur 2 vers 1

On considère un multiplexeur à **2 entrées vers 1 sortie**. Ce multiplexeur est appelé **MUX21**, ses entrées sont appelées **A** et **B**. Sa sortie est appelée **Y**. La commande de sélection de l'entrée qui est transmise à la sortie est appelée **SEL**.



1. Faites la table de vérité puis le tableau de Karnaugh et déduisez en une fonction booléenne qui exprime la sortie Y en fonction des entrées A, B et de la commande SEL.
2. Écrivez en VHDL une entité pour le multiplexeur MUX21.
3. Ecrivez en VHDL une architecture flot de données (DATAFLOW) pour le multiplexeur MUX21.
4. Simulez votre description en éditant des stimuli avec Modelsim (**voir partie 2.1 de la notice**). Tester les 8 combinaisons possibles des signaux d'entrées et vérifier que les valeurs de sorties correspondent bien aux résultats attendus.

Exercice 2 : Additionneur 1 bit



On considère un **additionneur 1 bit**. Cet additionneur est appelé **ADDC1**, ses entrées sont appelées **A** et **B**. Sa sortie est appelée **SUM**. Nous rajoutons à cet additionneur une entrée de retenue (carry) **CIN** et une sortie de retenue **COUT**.

1. Faites la table de vérité puis la table de Karnaugh et déduisez-en une fonction booléenne qui exprime les sorties SUM et COUT en fonction des entrées A, B et CIN.
2. Écrivez en VHDL une entité pour l'additionneur ADDC1.
3. Ecrivez en VHDL une architecture flot de données (DATAFLOW) pour l'additionneur ADDC1.
4. Simulez votre description en éditant des stimuli avec Modelsim (**voir partie 2.1 de la notice**). Tester les 8 combinaisons possibles des signaux d'entrée et vérifiez que les valeurs des sorties correspondent bien aux résultats attendus.

Exercice 3 : Multiplexeur 4 vers 1

On considère un multiplexeur à 4 entrées vers 1 sortie. Ce multiplexeur est appelé **MUX41**, ses entrées sont appelées **I(0), I(1), I(2) et I(3) (bus à 4 entrées)**. Sa sortie est appelée **Y**. Les commandes de sélection de l'entrée qui est transmise à la sortie sont appelées **SEL(0) et SEL(1) (bus à 2 entrées)**.

1. Ouvrir le fichier **Exo3/mux41.vhd** qui contient le squelette du multiplexeur 4 vers 1. Complétez l'architecture comportementale (BEHAVIOUR).
2. Simuler votre description en utilisant un banc de test (**voir partie 2.2 de la notice**). Vous trouverez le banc de test (**mux41_tb.vhd**) dans le répertoire **Exo3**.
3. Vérifiez les résultats visuellement dans la fenêtre **wave**. Vérifiez dans la fenêtre **Transcript** qu'aucune erreur ou warning n'est reportée et que vous obtenez bien le message suivant :
   `# ** Note: End of test. Verify that no error was reported.`
4. Introduisez une erreur dans la description **MUX41**, refaites la simulation, regardez les messages dans la fenêtre Transcript. Qu'en déduisez-vous sur l'utilité du banc de test ?

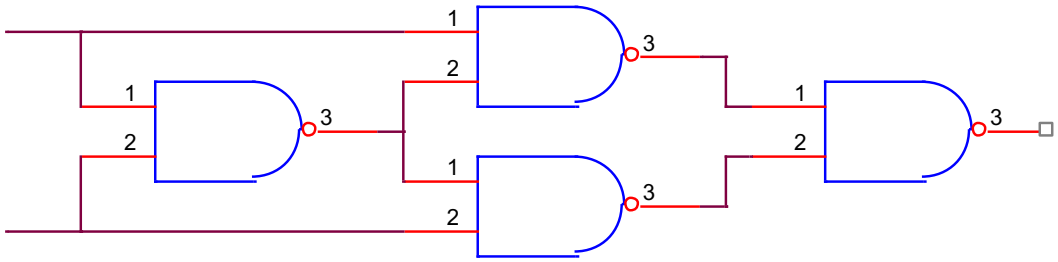
Exercice 4 : Additionneur 4 bits

On considère un additionneur 4 bits. Cet additionneur est appelé **ADDC4**, ses entrées sont appelées I1 et I2 (bus de 4 bits). Sa sortie est appelée SUM (bus de 4 bits). Nous rajoutons à cet additionneur une entrée de retenue CIN et une sortie de retenue COUT.

1. Faites un schéma qui décrit cet additionneur à l'aide de l'ADDC1 décrit précédemment.
2. Copiez dans le répertoire **/Exo4** le composant **ADDC1** que vous avez réalisé lors de **l'exercice 2**.
3. Ouvrir le fichier **/Exo4/addc4.vhd** qui contient le squelette de l'additionneur 4 bits. Complétez l'architecture structurelle (**STRUCT**).
4. Simulez l'additionneur 4 bits en écrivant un script de simulation (**voir partie 2.3 de la notice**) et en utilisant le banc de test fourni. Vous trouverez le banc de test (**addc4_tb.vhd**) dans le répertoire **/Exo4**.

Exercice 5 : Porte ou exclusif

1. Créer une entité / architecture VHDL qui représente une porte NON-ET (NAND) à deux entrées.
2. Écrire une porte OU-Exclusif (XOR) dans un autre fichier, en instanciant quatre exemplaires de la porte NAND qu'il faudra connecter conformément au schéma ci-dessous. Compléter le schéma avec des identifieurs appropriés.



3. Simuler votre composant en écrivant un script de simulation et en utilisant le Banc de Test fournit (xor2_tb.vhd). Vérifier que les résultats de simulation sont corrects.

Exercices du C4

Avant de commencer les exercices, il faut copier et dézipper dans votre répertoire de travail le dossier « ExoC4 – Sources.zip »

Exercice 1 : Multiplexeur 8 vers 1

On considère un multiplexeur à 8 entrées vers 1 sortie. Ce multiplexeur est appelé MUX81, son entrée est appelée I (bus à 8 entrées). Sa sortie est appelée Y. La commande de sélection de l'entrée qui est transmise à la sortie est appelée SEL (bus à 3 entrées).

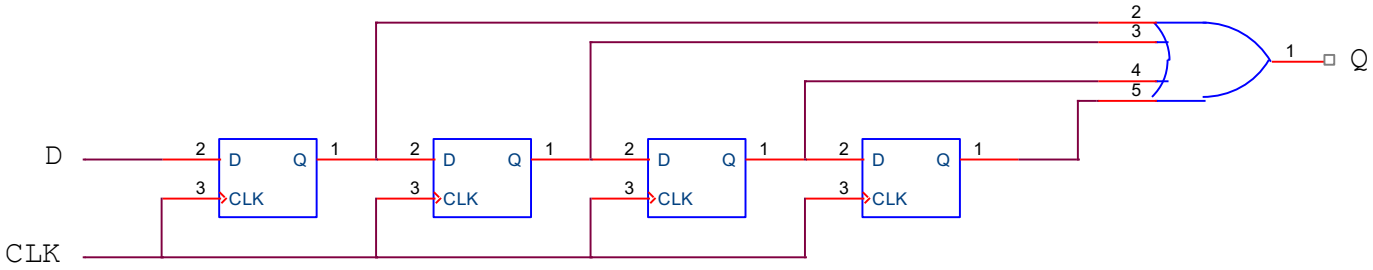
1. Ouvrir le fichier **mux81.vhd** et compléter l'architecture comportementale (BEHAVIOUR) en utilisant **un process combinatoire et l'instruction case**.
2. Simuler votre description en utilisant un script de simulation (**voir partie 2.3 de la notice**). Vous trouverez le banc de test (**mux81_tb.vhd**) dans le répertoire **Exo1**.
3. Vérifiez les résultats visuellement dans la fenêtre **wave**. Vérifiez dans la fenêtre **Transcript** qu'aucune erreur ou warning n'est reportée et que vous obtenez bien le message suivant :
   `# ** Note: End of test. Verify that no error was reported.`

Exercice 2 : Serial OR

Cet exercice permet de se familiariser avec l'emploi des signaux et des process synchrones. Écrire en VHDL et tester une fonction OU logique sur un registre à décalage de quatre bits, dont la déclaration d'entité est reproduite ci-dessous. Un fichier contenant le squelette se trouve sous **Exo2/serialor.vhd**. Le Banc de Test est déjà écrit dans le fichier **Exo2/serialor_tb.vhd**.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SerialOR is
  port (   CLK,rst    : in std logic;
           D     : in std logic;
           Q     : out std logic);
End entity;
end entity SerialOR;
```



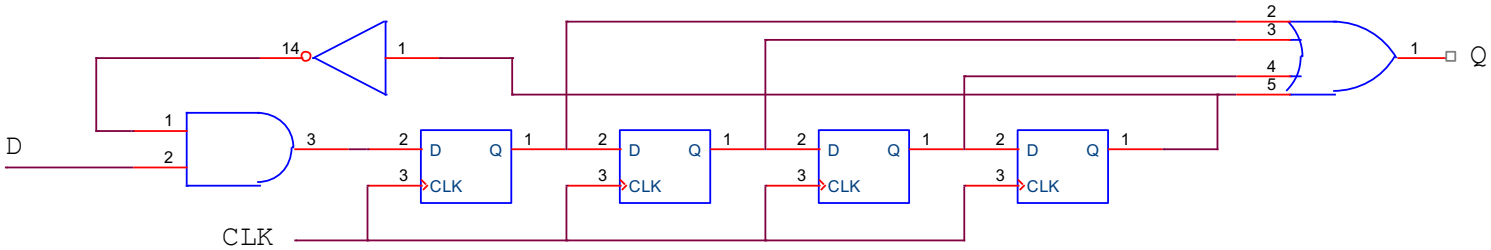
Le code VHDL doit décrire la fonction suivante. Simuler le serialor en utilisant un script de simulation (**voir partie 2.3 de la notice**).

Le Banc de Test déjà écrit élabore un signal « OK » qui devient faux si la sortie du module testé n'est pas conforme à la fonction demandée. Il faut donc, à la simulation, afficher ce signal OK et vérifier qu'il reste vrai pendant toute la simulation.

Attention ! Il y a deux architectures du Banc de Test, une pour chaque partie de cet exercice ! Assurez-vous bien de simuler la bonne architecture (**TB1**).

S'il vous reste du temps

Modifier le code VHDL pour décrire la fonction ci-dessous. Créer une deuxième architecture (RTL2) en laissant intacte la première ! Simuler là avec le deuxième Banc de Test.



Exercice 3 : Circuit MinMax

II faut concevoir et coder un circuit combinatoire qui calcule les valeurs Min et Max. Ce circuit doit être conforme à la déclaration d'entité reproduite ci-dessous, que vous trouverez dans le fichier **Exo3/minmax.vhd**. Vous trouverez également un squelette pour le Banc de Test dans **Exo3/minmax_tb.vhd**.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ail;

entity MinMax is
port( Min0Max1 : in STD LOGIC;    -- 0:min,   l:max
      X, Y     : in  STD LOGIC VECTOR (3 downto 0);
      Z        : out STD LOGIC VECTOR (3 downto 0));
end entity MinMax;
```

Quand **Min0Max1** vaut **'0'**, **Z** est le plus petit (Min) de **X** et de **Y**, et quand **Min0Max1** vaut **'1'**, alors **Z** doit être le plus grand (Max) de **X** et de **Y**. Sachant que l'on interprète **X**, **Y** et **Z** comme des vecteurs représentant des nombres binaires non signés.

En d'autres termes, **Z** doit recopier soit **X** soit **Y**, en fonction du plus grand des deux et de la valeur de **Min0Max1**. Noter que si **X** et Y sont égaux, alors **Minimum = Maximum = X = Y.**

Si vous avez encore des doutes sur la fonctionnalité demandée, reportez-vous à la table des vecteurs ci-dessous.

Astuce : Pour comparer des vecteurs en tant que nombres non signés, on peut simplement utiliser la comparaison en VHDL : **« if X > Y then... ».**

Vous pouvez implémenter cette fonction au choix en un seul process ou en plusieurs.

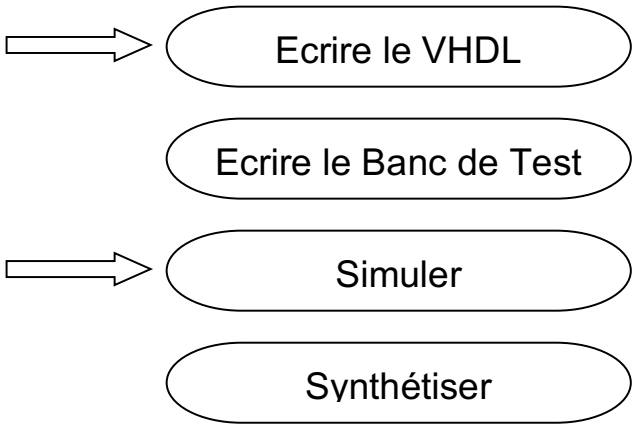
Pour vérifier le code VHDL, **écrire un Banc de Test** qui applique les vecteurs listés ci-dessous (ne pas tenter d'appliquer toutes les combinaisons possibles).
Le banc de test doit aussi tester la sortie et la comparer aux valeurs attendues ci-dessous. On peut par exemple écrire ok <= Z= " 0000 " avec un signal ok de type *boolean.*

Astuce : ne pas oublier d'insérer des délais entre l'assignation des entrées et la vérification de la sortie...

Min0Max1	X	Y	Z
0	0000	0000	0000
1	0000	0000	0000
1	0001	0000	0001
0	0001	0000	0000
0	0001	0010	0001
1	0001	0010	0010
1	0101	0010	0101
0	0101	0010	0010
0	0101	1010	0101
1	0101	1010	1010

Une fois que la simulation donne satisfaction, insérer un vecteur supplémentaire *incorrect* (dupliquer un vecteur existant et modifier la valeur attendue par exemple) et vérifier que la simulation trouve bien l'erreur.

Exercice C6 – Décodeur



Cet exercice permet de pratiquer les fonctions de conversion sur les vecteurs et d'utiliser un style pratique et performant pour ce type d'applications (décodage). Écrire et vérifier une entité VHDL (et son architecture) qui implémente une fonction de décodage 6 bits vers 64 bits. On adoptera le modèle d'entité ci-dessous qui est situé dans le fichier exoC6/decoder.vhd. Le Banc de Test est prêt à l'emploi sous exoC6/decoder_tb.vhd.

```
library IEEE;
  use IEEE.Std_logic_1164.all;
  use IEEE.NUMERIC_STD.all;

entity DECODER is
  port ( I   : in std logic vector{5 downto 0); -- Numéro du bit 0 .. 63
         EN  : in std logic;                    -- Enable, actif haut
         Y   : out std logic vector(63 downto 0));  -- Sortie "one hot"
end entity;
```

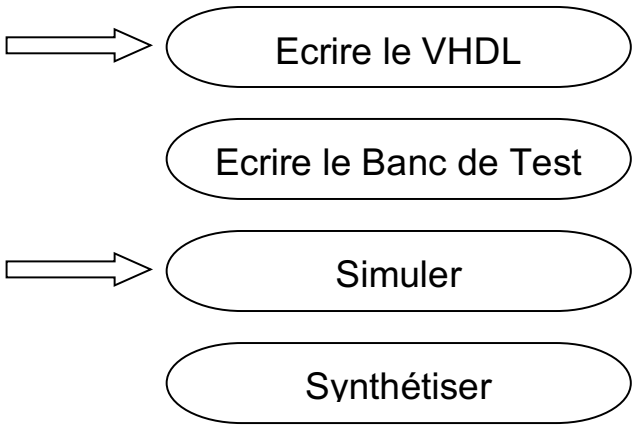
Fonction : le code d'entrée sur 6 bits `I(5 downto 0)` représente un nombre binaire non signé (que nous nommerons « k ») compris entre 0 et 63 (2^6-1). Le vecteur de sortie `Y(63 downto 0)` possède au plus un bit non nul Y(k) dans le cas ou l'entrée EN=1 (active). Si EN=0 (inactive), alors l'ensemble du vecteur Y est nul.
En d'autres termes, on veut que si EN=1, `Y(valeur_entière_de_I) = 1`, et `Y(autres)=0`, et que si EN=0, alors `Y(tous)=0`. Par exemple :

I	EN	Y
000000	1	0000000000000000000000000000000000000000000000000000000000000001
000001	1	0000000000000000000000000000000000000000000000000000000000000010
000010	1	0000000000000000000000000000000000000000000000000000000000000100
000011	1	0000000000000000000000000000000000000000000000000000000000001000
000100	1	0000000000000000000000000000000000000000000000000000000000010000
111110	1	0100000000000000000000000000000000000000000000000000000000000000
111111	1	1000000000000000000000000000000000000000000000000000000000000000
XXXXXX	0	0000000000000000000000000000000000000000000000000000000000000000

Ne PAS essayer de décrire cette fonction par une instruction case de 64 choix, ou par 64 équations booléennes! Pensez plutôt à utiliser l'entrée « I » en tant qu'entier par une (des) conversion(s) appropriée(s).

Le Banc de Test vérifie automatiquement que la sortie est conforme à la valeur attendue et positionne le signal OK. Il faut donc, à la simulation, afficher et vérifier ce signal OK.

Exercice C7 – Compteur / Décompteur



Première partie

Cet exercice met en pratique la conception et la synthèse de logique séquentielle synchrone. Il est demandé ici de coder en VHDL et de vérifier un compteur / décompteur binaire synchrone 8 bits. La déclaration d'entité demandée est reproduite ci-dessous. Vous la trouverez aussi dans le fichier exoC7/counter.vhd qu'il conviendra de compléter ainsi qu'un Banc de Test prêt à l'emploi dans le fichier exoC7/counter_tb.vhd.

```
library IEEE;
  use IEEE.Std_logic_1164.all;
entity Counter is
  port( Clock : in std_logic;  -- system clock
        Reset : in std_logic;  -- asynchronous reset, active high
        Enable: in std_logic;  -- synchronous enable, active high
        Load  : in std_logic;  -- synchronous load
        UpDn  : in std_logic;  -- 1=Up, 0=Down
        Data  : in std_logic_vector(7 downto 0);  -- loaded Data
        Q     : out std_logic_vector(7 downto 0)  -- counter's output
      );
end entity;
```

Le compteur est synchrone : il fonctionne sur front montant du signal d'horloge Clock. La direction de comptage est définie par UpDn : comptage si UpDn=1, et décomptage si UpDn=0. Les entrées Load et Enable sont synchrones et actives à 1. Reset a la priorité la plus forte et met le compteur à zéro immédiatement, sans attendre l'horloge (mise à zéro asynchrone). Enable est ensuite l'entrée la plus prioritaire, suivie par Load, puis UpDn. Quand Enable est inactive (0), le compteur reste inchangé (sauf s'il y a un reset). Quand Load est active ainsi que Enable, le vecteur d'entrée Data est chargé dans le compteur. Enfin, quand Load est inactif, le compteur compte si upDn=1 ou décompte si upDn=0.

La table de Vérité du compteur est reproduite ci-dessous.

Reset	Clock	Enable	Load	UpDn	Data	Q
1	-	-	-	-	-	0
0	Rise	0	-	-	-	Q
0	Rise	1	1	-	Data	Data
0	Rise	1	0	0	-	Q-1
0	Rise	1	0	1	-	Q+1

Le Banc de Test vérifie le bon fonctionnement du compteur et positionne le signal booléen OK. OK = false signifie que le Banc de Test a décelé une anomalie. Pour vérifier le module, il faudra donc afficher OK et s'assurer qu'il reste vrai pendant toute la simulation.

Deuxième partie

1. Modifier le compteur pour coder un reset synchrone (et non asynchrone).
2. Simuler à nouveau.

Troisième partie

Créer une entité RingCounter (compteur en anneau) en partant du code du compteur binaire. Ce « compteur » en anneau décrit cycliquement les huit états suivants :

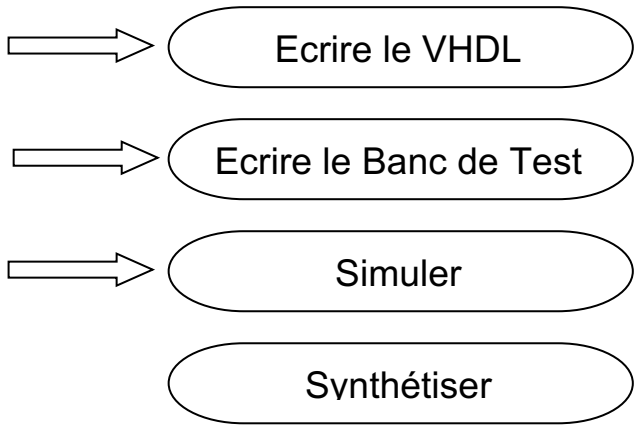
```
0   00000001
1   00000010
2   00000100
3   00001000
4   00010000
5   00100000
6   01000000
7   10000000
```

Les fonctions Enable, Load, UpDown doivent être conservées. Simuler avec le Banc de Test. Modifier le banc de test pour attacher RingCounter à l'instanciation du Banc de Test. Bien sûr, le booléen OK deviendra faux (ce n'est pas vraiment un compteur au sens binaire).

Votre code doit permettre au compteur de retrouver un des huit états ci-dessus même lorsqu'il est chargé par une valeur quelconque (avec plus d'un seul bit positionné à '1'). Ci-dessous un exemple de « re-convergence » possible :

```
11010000
10100000
01000000
10000000
00000001
00000010
```

Exercice C8 : RamChip (Composant mémoire)



Nous allons coder et vérifier ici un modèle *comportemental (behavioural)* pour un composant mémoire. Il ne s'agit donc pas d'une description synthétisable ce qui est confirmé par l'absence de flèche sur « Synthétiser », mais uniquement d'une description simulable.

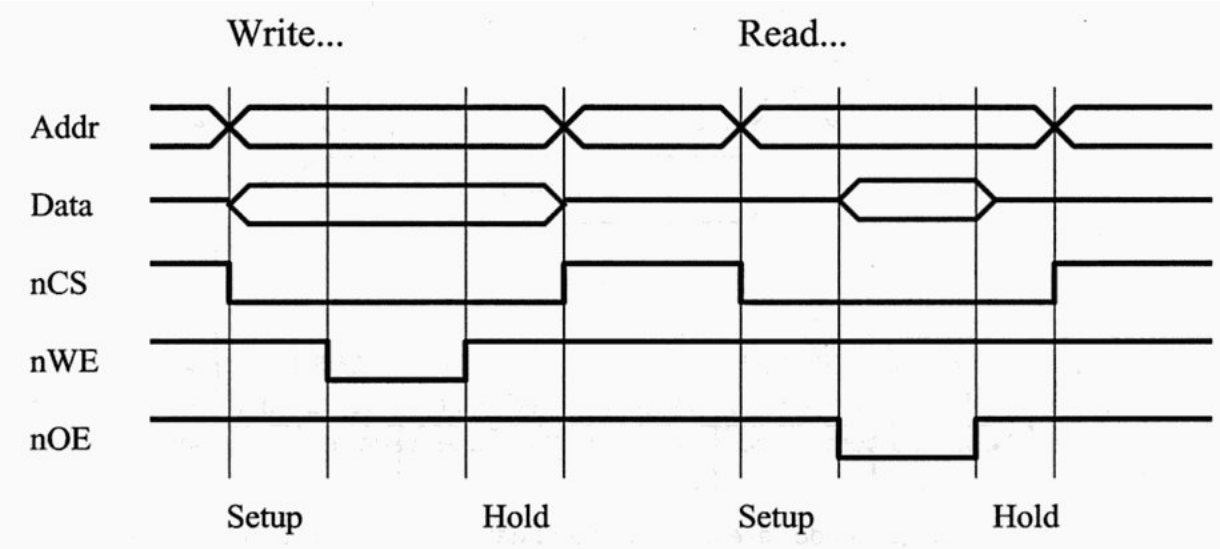
Le boîtier mémoire possède un bus 8 bits bi-directionnel, un bus d'adresse unidirectionnel de 4 bits, et trois signaux de contrôle : Chip Sélect (nCS), Output Enable (nOE) et Write Enable (nWE), tous trois actifs au niveau bas et actifs par état et non par front *(level sensitive).* On écrit la donnée Data dans la Ram (à l'adresse indiquée par Address) lorsque nWE = 0 et nCS = 0. La mémoire est lue et présente la donnée (située à l'adresse indiquée par Address) sur le bus Data lorsque nOE = 0 et nCS = 0. Lorsque nCS ou bien nOE sont hauts (inactifs), la mémoire laisse flotter (haute impédance) le bus Data.

En fait, la mémoire doit laisser le bus Data en haute impédance dans toutes les conditions, sauf dans le seul cas d'une opération légale de lecture ou d'écriture.

Voici la déclaration de l'entité :

```
entity RamChip is
  port (   Address    : in       std_logic_vector(3 downto 0);
           Data       : inout    std_logic_vector(7 downto 0);
           nCS        : in       std_logic; -- Chip Select
           nWE        : in       std_logic; -- Write Enable
           nOE        : in       std_logic  -- Output Enable, Read
       );
end entity;
```

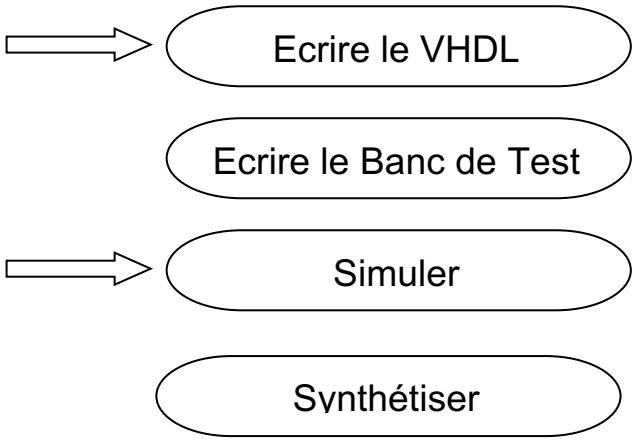
Le diagramme ci-dessous illustre le fonctionnement de la mémoire et les différentes phases des opérations de lecture et d'écriture. En aucun cas il n'est demandé de modéliser des délais *(timing)* pour la mémoire ! De ce point de vue, elle est considérée comme étant idéale (temps d'accès nul et contraintes *setup* et *hold* nulles). Les noms Setup et Hold dans le diagramme ci-dessous font référence à des phases des cycles de lecture et d'écriture, mais pas à des contraintes timing du modèle de mémoire.



Pour une vérification rapide, le Banc de Test (exoC9/ramchip.vhd) effectue les opérations suivantes :

1. Écrire une donnée à l'adresse 0,
2. Écrire une donnée différente à l'adresse 9,
3. Lire à l'adresse 0,
4. Lire à l'adresse 9.

Exercice C9 – RamBoard (Carte mémoire)



Cet exercice, va nous permettre d'améliorer le modèle de mémoire de l'exercice précédent en le rendant plus générique (paramétrable). Nous l'utiliserons alors pour constituer une carte en instanciant plusieurs mémoires et un décodeur.

Recopier le fichier ramchip.vhd de l'exercice précédent dans un nouveau répertoire exo10 (pas le banc de test). Modifier ramchip.vhd en ajoutant des paramètres génériques AddressSize et WordSize qui seront utilisés pour dimensionner les bus d'adresse et de données. Noter que la mémoire ne doit pas seulement avoir AddressSize cases mémoire !!! (Vérifiez votre modèle).

NE PAS modifier le banc de test unitaire de l'exercice précédent car le Banc de Test fourni teste déjà correctement votre travail (carte et RamChip).

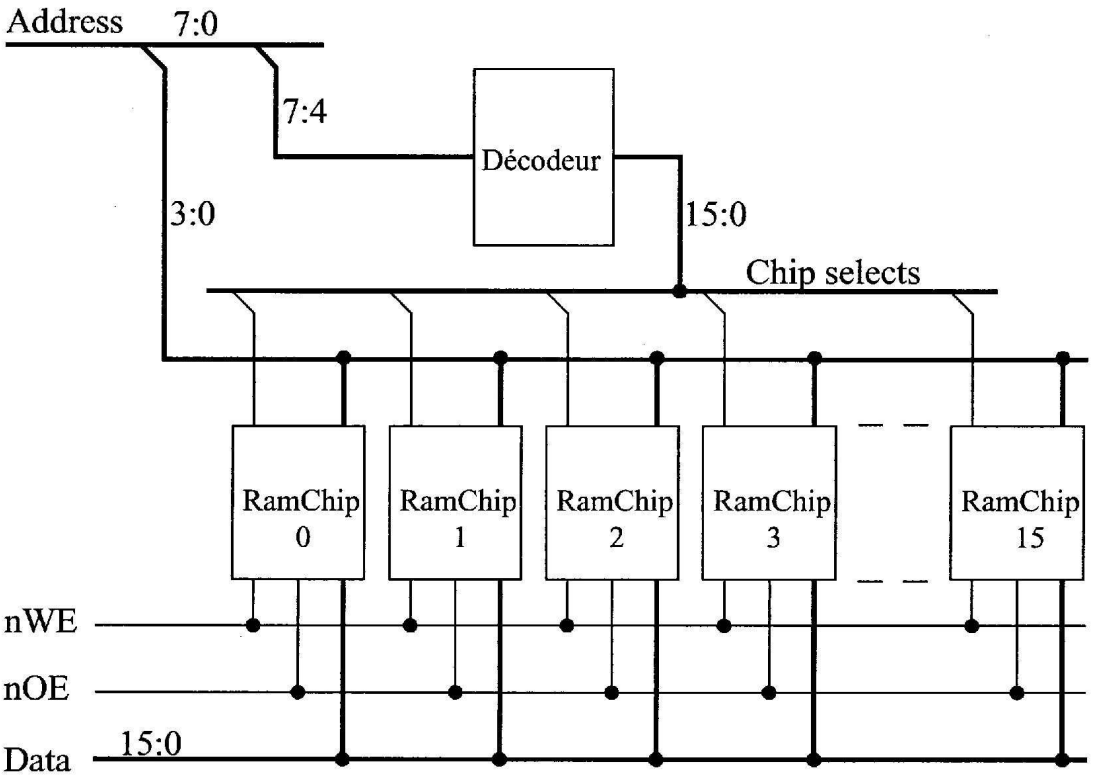
Écrire et vérifier par simulation la description de la carte (Board) dont le modèle est reproduit ci-dessous et fourni dans exoC9/board.vhd. Le Banc de Test est dans exoC9/board_tb.vhd.

```
entity Board is
      Port( Address :  in std_logic_vector(7 downto 0);
            Data :     inout std_logic_vector(15 downto 0);
            nWE,nOE :  in std_logic);
end entity Board;
```

La carte (Board) doit instancier 16 exemplaires identiques du composant mémoire RamChip. Chaque instance RamChip doit avoir un bus de données de 16 bits (comme la carte) et un bus d'adresse de 4 bits connecté aux bits de poids faible du bus d'adresse de la carte. Les quatre bits de poids fort du bus d'adresse de la carte doivent être décodés pour sélectionner une mémoire parmi les 16. Le schéma ci-après illustre la structure de la carte.

Remarque : nous ne vous recommandons pas de chercher à ré-utiliser le décodeur de l'exercice du C6. Un process de deux lignes suffit à assurer ce décodage, à comparer avec l'effort de modification et d'instanciation du décodeur précédent. A vous de décider.

Le Banc de Test fourni teste toutes les adresses possibles en écrivant l'adresse dans les données et en relisant l'ensemble (ce qui permet de vérifier qu'il n'y a pas d'interaction entre les cases mémoires, car chacune reçoit un contenu différent). En cas de problème, un message de violation d'assertion est reporté dans le transcript du simulateur : ne vous contentez pas de regarder les chronogrammes pour savoir si votre système fonctionne correctement !



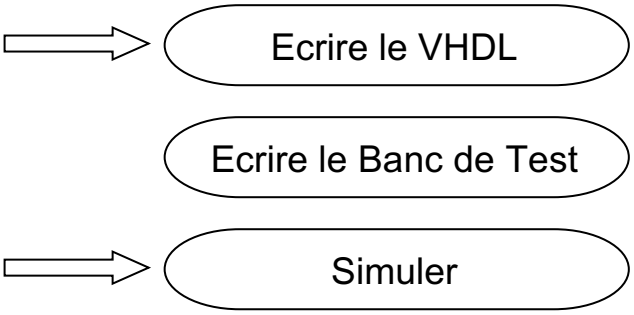
Deuxième partie :

Recopier RamChip, RamBoard et le Banc de Test dans un nouveau sous répertoire exoC6/generic car il est important de conserver une version fonctionnelle pour un exercice futur ! Dans le sous-répertoire exoC10/generic, modifier la carte *(Board)* pour la rendre générique elle aussi, en définissant les paramètres : BoardAddressSize, ChipAddressSize, Wordsize. Dans ce cas, ne pas oublier qu'il faut calculer le nombre de modules mémoire à instancier. Modifier le banc de test pour positionner les paramètres génériques aux bonnes valeurs.

Troisième partie :

Supprimer les paramètres génériques ! L'astuce consiste à utiliser des vecteurs non contraints et des attributs. En effet, les paramètres qui peuvent se déduire des bus Data et Address et de la taille du bus d'adresse du module mémoire (le seul paramètre explicite sera donc ChipAddressSize).

Exercice C10 : Séquenceur pour chronomètre



Nous allons construire ici le séquenceur (contrôleur) d'un chronomètre *(Watch Controller)* grâce à une description en machine d'états (FSM). Le diagramme d'états est fourni ci-après et représente le fonctionnement attendu. Le chronomètre possède deux boutons nommés « Start_Stop », et « Clear » (à ne pas confondre avec le reset du contrôleur !).

Le modèle de l'entité se trouve dans exoC10/fsm.vhd et le Banc de Test dans exoC10/fsmtb.vhd.

Il vous faut écrire le code VHDL, le vérifier par simulation avec le Banc de Test fourni. La déclaration d'entité reproduite ci-dessous est également dans le fichier fsm.vhd.

```
entity FSM is
  port (clk :  in STD_LOGIC;  -- system clock
        nRst      : in STD_LOGIC;  -- asynchronous reset, active low
        Start_Stop, Clear:  in STD_LOGIC; -- boutons du chronomètre
        Cnt_en, Cnt_rst: out STD_LOGIC);
end entity;
```

Ce séquenceur est bien sûr synchrone : il ne change d'état que sur un front montant d'horloge (qui s'appelle **`clk`**). Le reset asynchrone **`(nRst)`** est actif niveau bas et replace le séquenceur dans l'état appelé « zero ». Cette transition « par défaut » n'est pas représentée dans le diagramme mais doit bien sûr être codée !

Ne confondez pas l'état « reset » du séquenceur avec l'entrée **`nRst`** qui correspond à une remise à zéro hardware (changement de pile par exemple).

Diagramme d’états / transitions du séquenceur pour chronomètre :