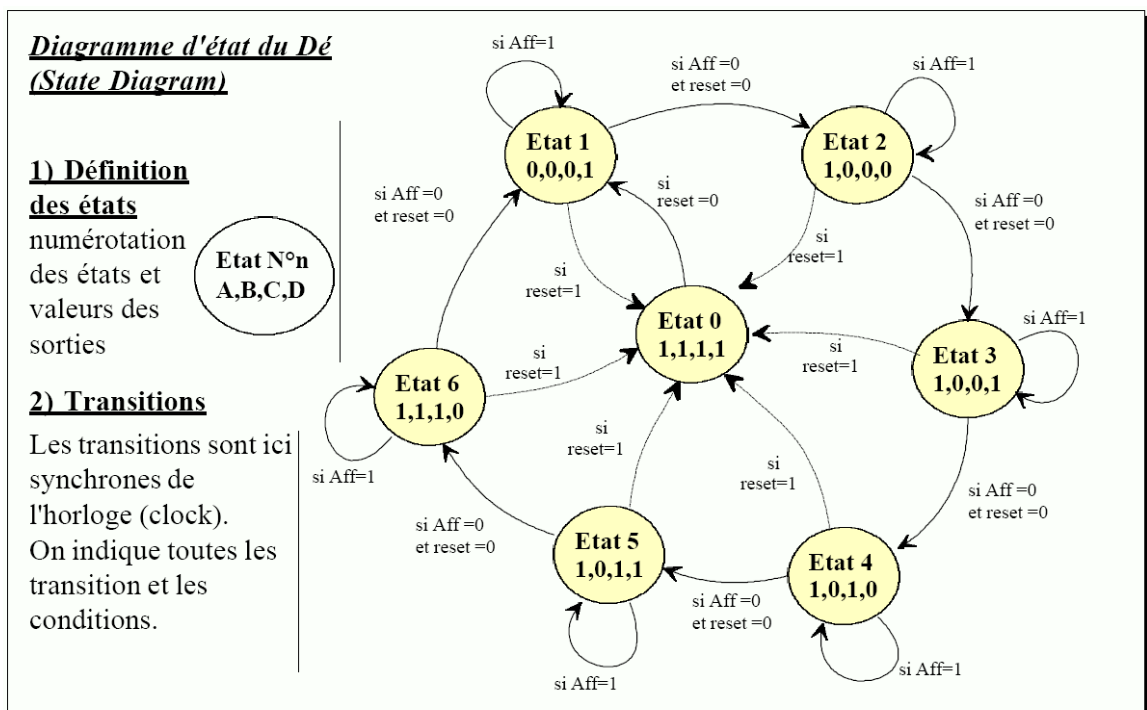
5 Montages en logique séquentielle

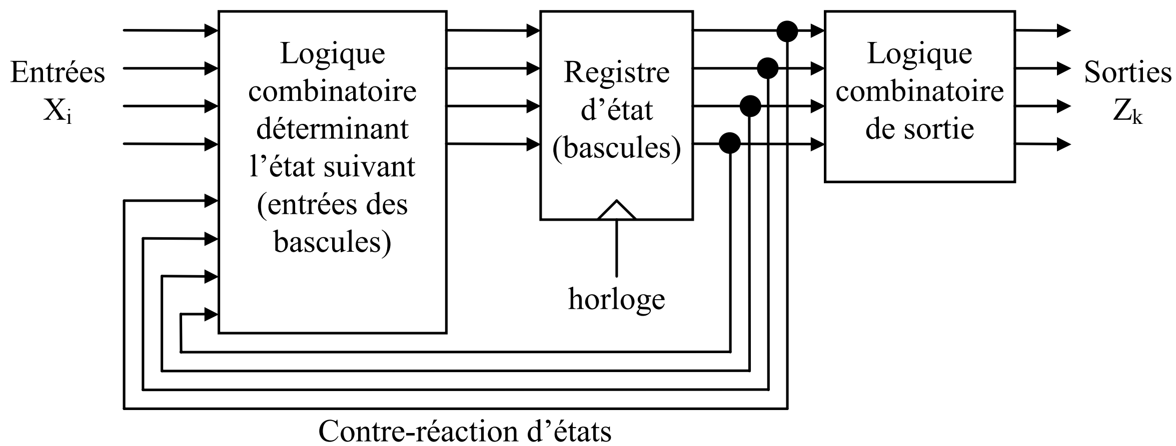
5.1 Machines d'états

Dans ce paragraphe, nous allons examiner une des familles les plus importantes de circuit séquentiel : la machine d'état ou machine à états finis (FSM : Finite State Machine). Une machine d'état est appelée ainsi car la logique séquentielle qui l'implémente ne peut prendre qu'un nombre fini d'états possibles. Il s'agit d'une manière de formaliser les automates qui est très utilisée en automatisme. En électronique, on peut s'en servir pour formaliser le fonctionnement d'un petit contrôleur. On utilise pour cela une table de transition d'état ou diagramme d'état. L'exemple suivant montre un diagramme d'état pour un montage électronique qui réalise un dé à jouer : Les cercles représentent les 7 états et les flèches représentent les transitions entre les états.

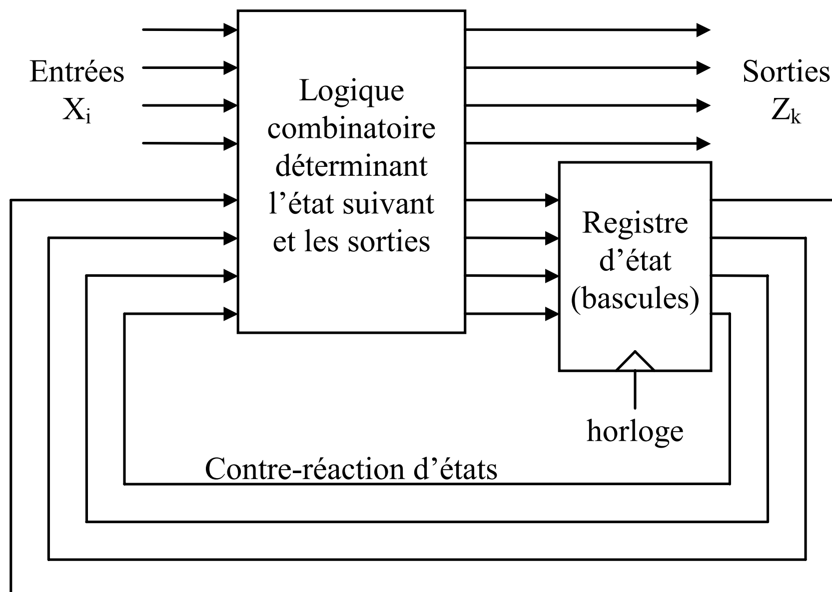


Il existe deux familles de machines d'état :

- la machine de Moore. Les sorties dépendent seulement de l'état présent n. Un circuit combinatoire d'entrée détermine, à partir des entrées et de l'état présent n, les entrées des bascules (D principalement) du registre d'état permettant de réaliser l'état futur n+1. Les sorties sont une combinaison logique de la sortie du registre d'état et changent de manière synchrone avec le front actif de l'horloge.



- la machine de Mealy. Les sorties dépendent de l'état présent n mais aussi de la valeur des entrées. La sortie peut donc changer de manière asynchrone en fonction de la valeur des entrées. Il existe une variante synchrone de la machine de Mealy avec un registre placé sur les sorties et activé par l'horloge. Toutefois, la machine de Moore est plus adaptée pour réaliser une machine d'état totalement synchrone.



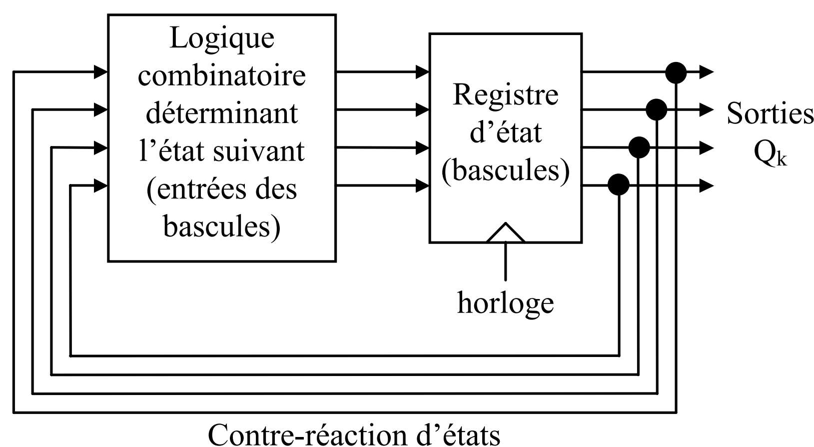
Les machines d'états servent à réaliser des automatismes complexes (contrôleur de feux tricolores par exemple) et ne nous intéressent pas directement. N'oublions pas que l'aboutissement ultime de la machine d'état en tant que système de contrôle, c'est bien sur le microprocesseur. Nous verrons en TP un exemple de petit contrôleur dans le domaine du traitement du signal.

5.2 Les générateurs de séquences synchrones

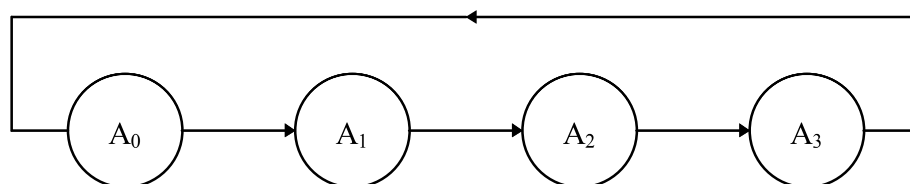
5.2.1 Compteur, décompteur

Nous allons maintenant étudier un cas particulier de machine d'état : le générateur de séquence synchrone. Il s'agit d'un cas simplifié de machine d'état de Moore dans laquelle :

- les sorties et les états internes sont identiques,
- la séquence est non-programmable,
- il n'y a pas d'entrées (il y a éventuellement des entrées de chargement, mais elles n'interviennent pas dans le déroulement de la séquence).



Les générateurs de séquences synchrones sont composés de <u>n bascules</u> synchronisées par une horloge qui est la seule commande extérieure du circuit. On peut ainsi coder 2^n états différents et réaliser un graphe d'évolution du type :



Le cycle comporte au plus 2^n états qui se succèdent toujours dans le même ordre. On débute l'étude par le codage des états. On peut généralement les classer parmi les séquences du tableau ci-dessous (sur 3 bits par exemple). Pour répondre à un besoin spécifique, tout autre combinaison des sorties est possible et peut constituer un cycle de m états avec $m \leq 2^n$.

compteur cycle complet (modulo 8)					compteur cycle incomplet					
binaire naturel			binaire quelconque							
compteur		décompteur		gray		Johnson		1 parmi 3		pseudo-aléatoire
Q_2 Q_1 Q_0		Q_2 Q_1 Q_0		Q_2 Q_1 Q_0		Q_2 Q_1 Q_0		Q_2 Q_1 Q_0		Q_2 Q_1 Q_0
0 0 0		1 1 1		0 0 0		0 0 0		0 0 1		0 0 1
0 0 1		1 1 0		0 0 1		0 0 1		0 1 0		1 0 0
0 1 0		1 0 1		0 1 1		0 1 1		1 0 0		1 1 0
0 1 1		1 0 0		0 1 0		1 1 1				1 1 1
1 0 0		0 1 1		1 1 0		1 1 0				0 1 1
1 0 1		0 1 0		1 1 1		1 0 0				1 0 1
1 1 0		0 0 1		1 0 1						0 1 0
1 1 1		0 0 0		1 0 0						

5.2.2 Cas particulier : les registres à décalages bouclés

Une fois le codage des états déterminé, on regarde dans la séquence s'il y a un décalage temporel. Si tel est le cas, alors il y a une solution évidente à base de bascules D d'équation $Q_{n+1} = D$. Parmi les séquences précédentes, les compteurs Johnson et 1 parmi 3 peuvent être réalisés à partir de bascules D.

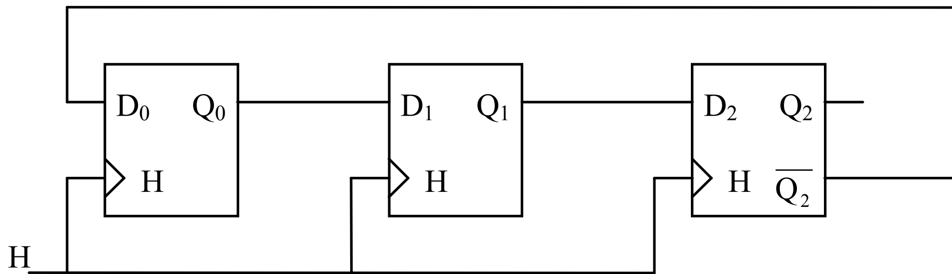
- Le compteur Johnson. On détermine à partir de la séquence d'états que $Q_1^{n+1} = D_1 = Q_0^n$, $Q_2^{n+1} = D_2 = Q_1^n$ et $Q_0^{n+1} = D_0 = f\left(Q_0^n, Q_1^n, Q_2^n\right)$. Il faut donc seulement chercher l'équation de D_0. Pour cela, on se demande quelle valeur mettre sur D_0 pour obtenir Q_0.

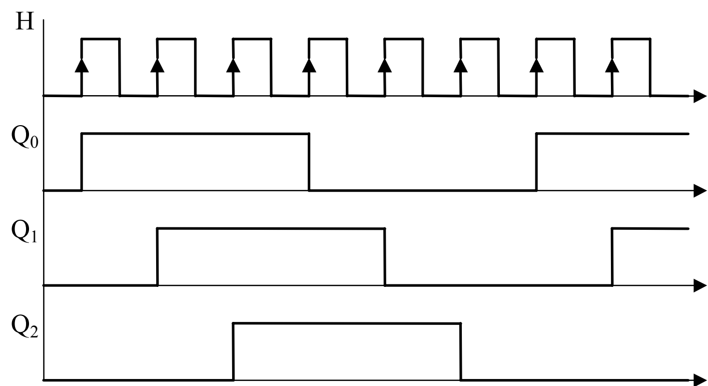
Q_2	Q_1	Q_0	D_0
0	0	0	1
0	0	1	1
0	1	1	1
1	1	1	0
1	1	0	0
1	0	0	0

D_0 →

Q_2 \ $Q_1 Q_0$	0 0	0 1	1 1	1 0
0	1	1	1	X
1	0	X	0	0

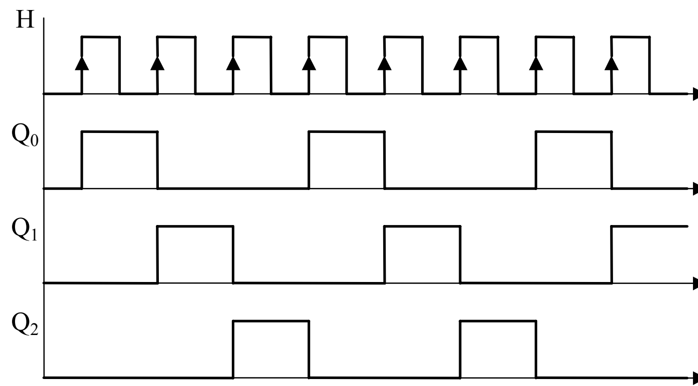
En simplifiant au maximum avec le tableau de Karnaugh, on obtient $D_0 = \overline{Q_2}$ ce qui donne le schéma et le chronogramme suivant :





Les bascules doivent être mises à 0 à la mise sous tension. On obtient des horloges de mêmes périodes, mais décalées en phase. Il s'agit d'un code jointif comme le code GRAY car il n'y a qu'une sortie qui change à chaque coup d'horloge.

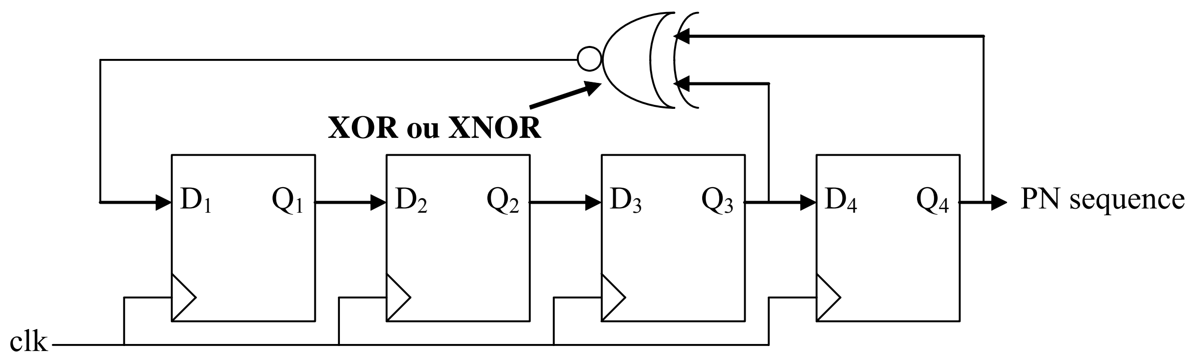
- Le compteur 1 parmi 3. Le schéma est identique au précédent avec D_0 connecté sur Q_2 (au lieu de $\overline{Q_2}$), et il faut mettre à 1 une des bascules à la mise sous tension (et les autres à 0). On obtient des horloges décalées à temps jointifs.



5.2.3 Le générateur pseudo-aléatoire (LFSR)

La génération d’une séquence aléatoire avec un système parfaitement déterministe est une opération a priori impossible. On s’en rapproche en générant une séquence pseudo-aléatoire, c'est-à-dire qui se rapproche du tirage au hasard. Les besoins sont très importants comme par exemple dans les transmissions numériques ou encore dans le domaine de la cryptographie. Une solution élégante à ce problème utilise un registre à décalage : c’est le Linear Feedback Shift Register ou LFSR.

Les séquences de longueur maximum MLS (Maximal length Sequence), également appelées PRBS (Pseudo Random Binary Sequence) ou PN-sequence (Pseudo Noise sequence), sont certaines séquences binaires de longueur $M = 2^n - 1$ où n indique l'ordre de la séquence. La réalisation d’un générateur de ce type de séquence est extrêmement simple. Elle associe un registre à décalage de longueur n et soit un XOR (ou exclusif) soit un XNOR. Voici un exemple pour n=4 :



Pour construire une MLS d'une longueur M donnée, nous avons besoin d'un polynôme primitif p(x) de degré n. La détermination de ces polynômes primitifs sort du cadre de ce

cours mais elle permet d'obtenir le polynôme caractéristique du LFSR, c'est-à-dire quelles sont les bonnes sorties du registre à décalage qu'il faut connecter au XOR (ou XNOR). Le tableau suivant indique les branchements à effectuer pour n allant de 3 à 168 :

n	XNOR from	n	XNOR from	n	XNOR from	n	XNOR from
3	3,2	45	45,44,42,41	87	87,74	129	129,124
4	4,3	46	46,45,26,25	88	88,87,17,16	130	130,127
5	5,3	47	47,42	89	89,51	131	131,130,84,83
6	6,5	48	48,47,21,20	90	90,89,72,71	132	132,103
7	7,6	49	49,40	91	91,90,8,7	133	133,132,82,81
8	8,6,5,4	50	50,49,24,23	92	92,91,80,79	134	134,77
9	9,5	51	51,50,36,35	93	93,91	135	135,124
10	10,7	52	52,49	94	94,73	136	136,135,11,10
11	11,9	53	53,52,38,37	95	95,84	137	137,116
12	12,6,4,1	54	54,53,18,17	96	96,94,49,47	138	138,137,131,130
13	13,4,3,1	55	55,31	97	97,91	139	139,136,134,131
14	14,5,3,1	56	56,55,35,34	98	98,87	140	140,111
15	15,14	57	57,50	99	99,97,54,52	141	141,140,110,109
16	16,15,13,4	58	58,39	100	100,63	142	142,121
17	17,14	59	59,58,38,37	101	101,100,95,94	143	143,142,123,122
18	18,11	60	60,59	102	102,101,36,35	144	144,143,75,74
19	19,6,2,1	61	61,60,46,45	103	103,94	145	145,93
20	20,17	62	62,61,6,5	104	104,103,94,93	146	146,145,87,86
21	21,19	63	63,62	105	105,89	147	147,146,110,109
22	22,21	64	64,63,61,60	106	106,91	148	148,121
23	23,18	65	65,47	107	107,105,44,42	149	149,148,40,39
24	24,23,22,17	66	66,65,57,56	108	108,77	150	150,97
25	25,22	67	67,66,58,57	109	109,108,103,102	151	151,148
26	26,6,2,1	68	68,59	110	110,109,98,97	152	152,151,87,86
27	27,5,2,1	69	69,67,42,40	111	111,101	153	153,152
28	28,25	70	70,69,55,54	112	112,110,69,67	154	154,152,27,25
29	29,27	71	71,65	113	113,104	155	155,154,124,123
30	30,6,4,1	72	72,66,25,19	114	114,113,33,32	156	156,155,41,40
31	31,28	73	73,48	115	115,114,101,100	157	157,156,131,130
32	32,22,2,1	74	74,73,59,58	116	116,115,46,45	158	158,157,132,131
33	33,20	75	75,74,65,64	117	117,115,99,97	159	159,128
34	34,27,2,1	76	76,75,41,40	118	118,85	160	160,159,142,141
35	35,33	77	77,76,47,46	119	119,111	161	161,143
36	36,25	78	78,77,59,58	120	120,113,9,2	162	162,161,75,74
37	37,5,4,3,2,1	79	79,70	121	121,103	163	163,162,104,103
38	38,6,5,1	80	80,79,43,42	122	122,121,63,62	164	164,163,151,150
39	39,35	81	81,77	123	123,121	165	165,164,135,134
40	40,38,21,19	82	82,79,47,44	124	124,87	166	166,165,128,127
41	41,38	83	83,82,38,37	125	125,124,18,17	167	167,161
42	42,41,20,19	84	84,71	126	126,125,90,89	168	168,166,153,151
43	43,42,38,37	85	85,84,58,57	127	127,126		
44	44,43,18,17	86	86,85,74,73	128	128,126,101,99		

Les propriétés principales de la séquence binaire obtenue sont :

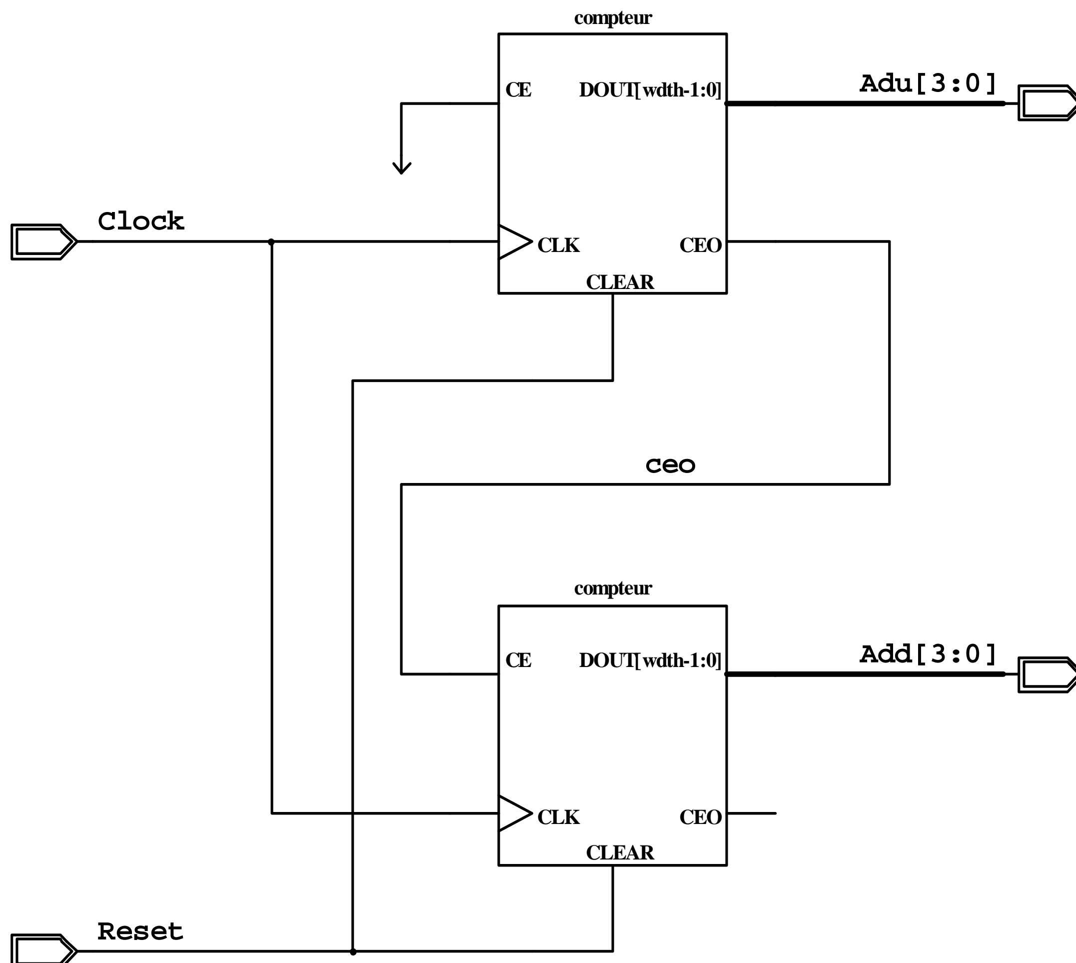
- La séquence est équiprobable (en fait, à une valeur près),
- Il y a une valeur interdite (tout à 0 avec un XOR, tout à 1 avec un XNOR),
- La fonction d'autocorrelation de la séquence est un dirac.

5.3 *Description en VHDL*

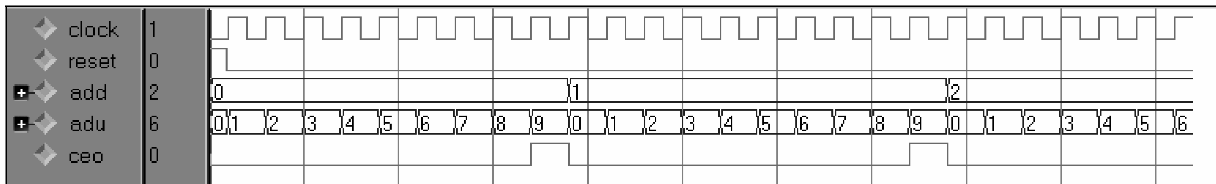
5.3.1 Description modulaire et paramètres génériques

Lorsque la complexité du design augmente, on le divise en blocs plus petits pour maintenir dans chaque bloc un niveau de complexité acceptable. On utilise pour cela une description modulaire hiérarchique, c'est-à-dire une arborescence hiérarchique de composants (un composant correspond un bloc). Le design principal à la racine de l'arborescence s'appelle généralement le top level design (parfois le root design). L'utilisation d'un composant dans le design s'appelle **l'instanciation d'un composant**. Instancier un composant en VHDL est équivalent en saisie de schéma à poser le symbole graphique du composant sur le schéma.

La description modulaire conduit assez naturellement à la notion de composant générique, c'est-à-dire un composant dont la taille est déterminée au moment où on l'instancie dans le design. Il existe deux méthodes en VHDL pour faire de la description modulaire. Nous allons maintenant voir la première, la description sans package. Nous souhaitons réaliser la fonction logique suivante :



Il s'agit de deux compteurs BCD montés en cascade pour compter de 0 à 99. Le chronogramme suivant montre le fonctionnement du montage :



La description modulaire sans package du design est la suivante. Dans cet exemple, on définit d'abord un composant `compteur` (lignes 1 à 31), puis le top level design `gen_cnt` (lignes 32 à 53). Dans l'architecture de `gen_cnt` (lignes 40 à 53), il faut définir les entrées-sorties du composant via l'instruction component (lignes 41 à 48) qui a la même forme que la déclaration de l'entité correspondante. Il ne reste plus ensuite qu'à instancier les 2 composants `compteur` (lignes 51 et 52).

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.std_logic_arith.all;
4. use IEEE.STD_LOGIC_UNSIGNED.all;

5. entity compteur is
6.   generic (WDTH : integer :=4; STOP : integer :=10);
7.   port( CLK : in std_logic ;
8.          CE : in std_logic ;
9.       CLEAR : in std_logic;
10.      CEO : out std_logic;
11.     DOUT : out std_logic_vector(WDTH -1 downto 0));
12.end compteur;

13.architecture a1 of compteur is
14.  signal INT_DOUT : std_logic_vector(DOUT'range) ;
15.begin
16.  process(CLK, CLEAR) begin
17.    if (CLEAR='1') then
18.      INT_DOUT <= (others => '0');
19.    elsif (CLK'event and CLK='1') then
20.      if (CE='1') then
21.        if (INT_DOUT=STOP-1) then
22.          INT_DOUT <= (others => '0');
23.        else
24.          INT_DOUT <= (INT_DOUT + 1);
25.        end if;
26.      end if;
27.    end if;
28.  end process;
29.  CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and
   INT_DOUT(3) and CE;
30.  dout <= int_dout;
31.end;
```

```
32.library IEEE;
33.use IEEE.std_logic_1164.all;

34.entity gen_cnt is
35.  port(Clock : in std_logic;
36.       Reset : in std_logic;
37.         add : out std_logic_vector(3 downto 0);
38.         adu : out std_logic_vector(3 downto 0));
39.end gen_cnt;

40.architecture comporte of gen_cnt is

41.  COMPONENT compteur
42.    generic (WDTH : integer :=4; STOP : integer :=10);
43.    port( CLK : in std_logic ;
44.           CE : in std_logic ;
45.        CLEAR : in std_logic;
46.          CEO : out std_logic;
47.         DOUT : out std_logic_vector(WDTH -1 downto 0));
48.  END COMPONENT;

49.  signal ceo : std_logic;

50.begin
51.  cd4d : compteur generic map(4,10) port map(Clock, ceo, Reset, open, add);
52.  cd4u : compteur generic map(4,10) port map(Clock, '1', Reset, ceo, adu);
53.end comporte ;
```

L'instanciation (l'appel) d'un composant (ligne 51 et 52) se déroule de la manière suivante :

Nom_instance : nom_composant generic map(paramètres génériques**)** **port map(**signaux interconnections**)**

a. Il faut d'abord écrire sur la ligne d'appel un **nom d'instance** (dans notre exemple, cd4d à la ligne 51 et cd4u à la ligne 52). Ce nom d'instance sert à différencier plusieurs instanciations d'un même composant. Ce nom est optionnel, mais le synthétiseur en créera un automatiquement si vous l'omettez.

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);
```

b. On indique ensuite le **nom du composant** à instancier. Il s'agit du nom donné à la ligne 41 qui doit être le même que le nom défini dans l'entité ligne 5.

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);
```

c. On écrit ensuite les **paramètres génériques** s'il y en a. Grace à ces paramètres, on peut écrire un modèle générique d'un composant (ici, un compteur) et définir ses caractéristiques au moment de son appel dans le design. Voyons notre exemple. L'entité `compteur` déclare deux paramètres, la largeur du compteur en bits WDTH et le nombre d'état du compteur STOP. On peut définir des valeurs par défaut pour ces deux paramètres (ici, 4 pour WDTH et 10 pour STOP) utilisés si on ne passe pas de valeurs au moment de son appel.

```
ligne 6. generic (WDTH : integer :=4; STOP : integer :=10);
```

Au moment de l'instanciation, on passe les deux valeurs dans l'ordre de la définition (ligne 6) après le mot clé « `generic map` ». Si on omet le generic map, les paramètres par défaut seront utilisés.

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);
```

Ces paramètres peuvent être utilisés soit dans la déclaration des entrées sorties (ici, on définit la largeur du bus de sortie) :

```
ligne 11. DOUT : out std_logic_vector(WDTH-1 downto 0));
```

soit dans l'architecture du composant (ici, la valeur d'arrêt du compteur) :

```
ligne 21. if (INT_DOUT=STOP-1) then
```

Les paramètres génériques apportent une grande flexibilité aux descriptions de composants et permettent l'écriture de bibliothèques standardisées. Vous noterez l'utilisation de l'attribut range.

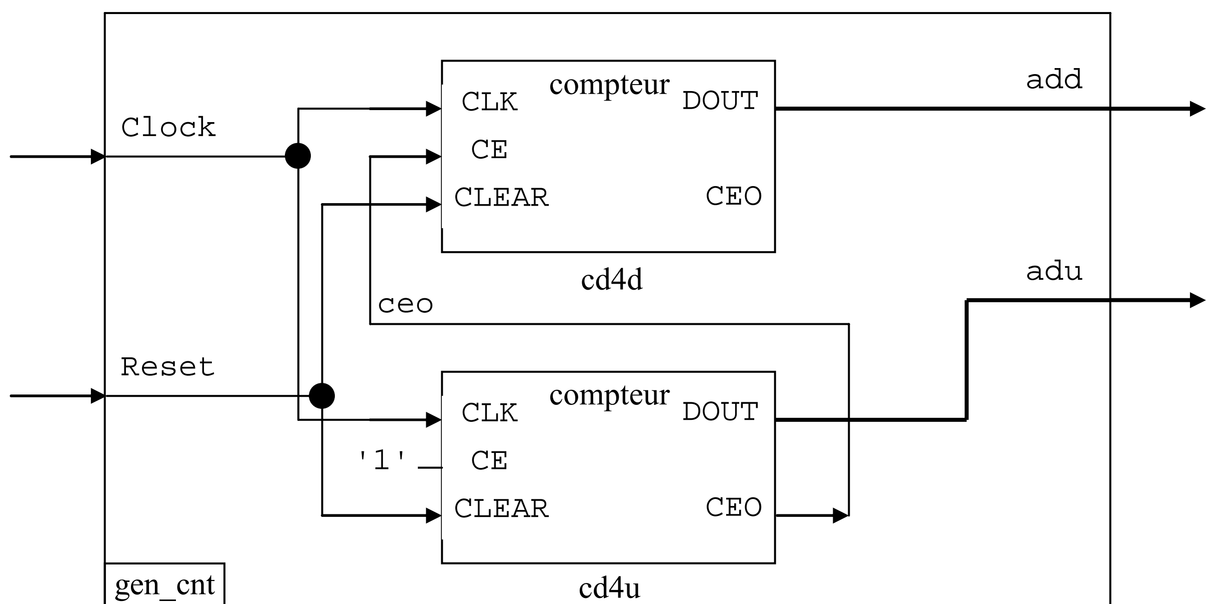
```
ligne 14. signal INT_DOUT : std_logic_vector(DOUT'range);
```

INT_DOUT a la même largeur que DOUT (de WDTH-1 à 0). L'attribut range simplifie l'écriture des modèles génériques.

d. Finalement, il faut relier les composants avec les autres éléments du design en utilisant des signaux d'interconnections. Au moment de l'instanciation, on passe les noms des signaux après le mot clé « `port map` ».

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);

ligne 52. cd4u : compteur generic map(4,10) port map(Clock,
'1', Reset, ceo, adu);
```

Voici les liaisons que nous souhaitons réaliser :



Prenons l'exemple du signal d'horloge `Clock`. On appelle paramètre formel le nom de la broche interne CLK du composant `compteur` et paramètre réel le nom du signal `Clock` dans `gen_cnt`. Il va falloir relier le signal `Clock` de `gen_cnt` avec le signal interne `CLK` de `compteur`. Deux méthodes sont possibles : la méthode implicite et la méthode explicite. Dans la méthode implicite, il suffit de mettre dans la parenthèse qui suit le `port map` les signaux de `gen_cnt` en suivant l'ordre de l'entité de `compteur` :

```
port(CLK : in std_logic;
      CE : in std_logic;
   CLEAR : in std_logic;
     CEO : out std_logic;
    DOUT : out std_logic_vector(WDTH -1 downto 0));
```

A la ligne 51, les liaisons se font automatiquement dans l'ordre de l'entité :

```
cd4d : compteur generic map(4,10) port map(Clock, ceo, Reset,
open, add);
```

Clock est reliée à CLK, ceo est relié à CE, … Quand la sortie d'un composant n'est reliée à rien, on utilise le mot-clef `open`.

Dans la méthode explicite, on indique pour chaque signal : `paramètre_formel => paramètre_réel`. On peut aussi utiliser la méthode explicite pour les paramètres génériques. Nous avons utilisé uniquement la méthode implicite jusqu'à maintenant. Voici les deux instanciations avec la méthode explicite :

```
cd4d : compteur
generic map(WDTH => 4,STOP => 10)
port map(CLK => Clock,
          CE => ceo,
       CLEAR => Reset,
         CEO => open,
        DOUT => add);
```

```
cd4u : compteur
generic map(WDTH => 4,STOP => 10)
port map(CLK => Clock,
          CE => '1',
       CLEAR => Reset,
         CEO => ceo,
        DOUT => adu);
```

La méthode explicite est plus longue à écrire, mais les signaux peuvent être mis dans le désordre. Cette solution peut être plus pratique quand le nombre de signaux à connecter devient élevé.

La description modulaire sans package devient relativement fastidieuse quand la taille du design augmente. D'où l'idée de regrouper les composants dans un paquetage (ou package) et dans un fichier séparé. Cela évite de redéclarer les composants dans le top level design et cela peut accélérer l'analyse du design au moment de la synthèse. En effet, avec plusieurs fichiers, le synthétiseur n'analyse que les fichiers qui ont été modifiés.

Dans la description modulaire avec package, la déclaration du composant `compteur` est effectuée dans un fichier séparé `cnt_pkg.vhd`. On trouve au début de ce fichier la déclaration des entrées sorties de tous les composants du package `gen_cnt_pkg` (lignes 3 à 12) puis la description complète de ces composants (entité et architecture, lignes 13 à 43).

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. package gen_cnt_pkg is

4.   COMPONENT compteur
5.     generic (WDTH : integer :=4; STOP : integer :=10);
6.     port( CLK : in std_logic;
7.            CE : in std_logic;
8.         CLEAR : in std_logic;
9.           CEO : out std_logic;
10.         DOUT : out std_logic_vector(WDTH -1 downto 0));
11.     END COMPONENT;

12.end gen_cnt_pkg;

13.library IEEE;
14.use IEEE.std_logic_1164.all;
15.use IEEE.std_logic_arith.all;
16.use IEEE.STD_LOGIC_UNSIGNED.all;

17.entity compteur is
18.  generic (WDTH : integer :=4; STOP : integer :=10);
19.  port( CLK : in std_logic;
20.         CE : in std_logic;
21.      CLEAR : in std_logic;
22.        CEO : out std_logic;
23.       DOUT : out std_logic_vector(WDTH -1 downto 0));
24.end compteur;

25.architecture a1 of compteur is
26.  signal INT_DOUT : std_logic_vector(DOUT'range) ;
27.begin
28.  process(CLK, CLEAR) begin
29.    if (CLEAR='1') then
30.      INT_DOUT <= (others => '0');
31.    elsif (CLK'event and CLK='1') then
32.      if (CE='1') then
33.        if (INT_DOUT=STOP-1) then
34.          INT_DOUT <= (others => '0');
35.        else
36.          INT_DOUT <= (INT_DOUT + 1);
37.        end if;
38.      end if;
39.    end if;
40.  end process;
41.  CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and
   INT_DOUT(3) and CE;
42.  dout <= int_dout;
43.end;
```

Le top level design se trouve dans le fichier `cnt.vhd` (les noms des fichiers utilisés sont sans importance). Le simulateur ou le synthétiseur va compiler le package dans la librairie par

défaut work. Pour pouvoir l'utiliser dans gen_cnt, il suffit de déclarer dans la liste des packages utilisés :

```
use work.gen_cnt_pkg.all;
```

Les composants peuvent ensuite être instanciés comme précédemment.

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use work.gen_cnt_pkg.all;

4. entity gen_cnt is
5.   port(
6.     Clock : in std_logic;
7.     Reset : in std_logic;
8.       add : out std_logic_vector(3 downto 0);
9.       adu : out std_logic_vector(3 downto 0));
10.end gen_cnt;

11.architecture comporte of gen_cnt is
12.  signal ceo : std_logic;
13.begin
14.  cd4d : compteur generic map(4,10) port map(Clock, ceo, Reset, open, add);
15.  cd4u : compteur generic map(4,10) port map(Clock, '1', Reset, ceo, adu);
16.end comporte ;
```

Il faut noter que :

a) un package peut contenir un nombre quelconque d'éléments.
b) un package peut contenir autre chose que des composants : par exemple, des définitions de types ou des fonctions.
c) on met en général dans un package des éléments utilisables dans des applications similaires.
d) Une description peut faire appel à plusieurs packages. Il suffit d'inclure autant de clause use qu'il y a de packages à utiliser.
e) Il est tout à fait possible de compiler un package dans une libraire autre que la librairie work et de le rendre accessible à d'autres développeurs. C'est nécessaire quand le projet nécessite plusieurs designers. C'est le cas notamment pour la librairie IEEE et les packages std_logic_1164 ou numeric_bit par exemple.

La description modulaire avec package est la méthode normale d'organisation du travail quand on développe un projet en VHDL.

Une question se pose encore : quel type de signal doit-on utiliser dans l'entité d'un composant ?

> En général, les paramètres génériques sont de type entiers ou bien `std_logic`. Pour les entrées-sorties, on utilise normalement des `std_logic`, mais pas de type `signed` ou `unsigned`. Cela simplifie l'instanciation des composants. Si nécessaire, on réalisera des conversions dans le composant pour travailler en signé ou en non-signé.

Il reste un dernier point à régler : le compteur n'est pas générique à cause du calcul du CEO. En effet, dans notre exemple, CEO vaut 1 si CE vaut 1 et si INT_DOUT = 9.

```
41. CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and INT_DOUT(3) and CE;
```

Le paramètre générique STOP n'intervient pas dans le calcul de CEO. Deux modifications sont possibles. La plus évidente est le remplacement de la ligne 41 par :

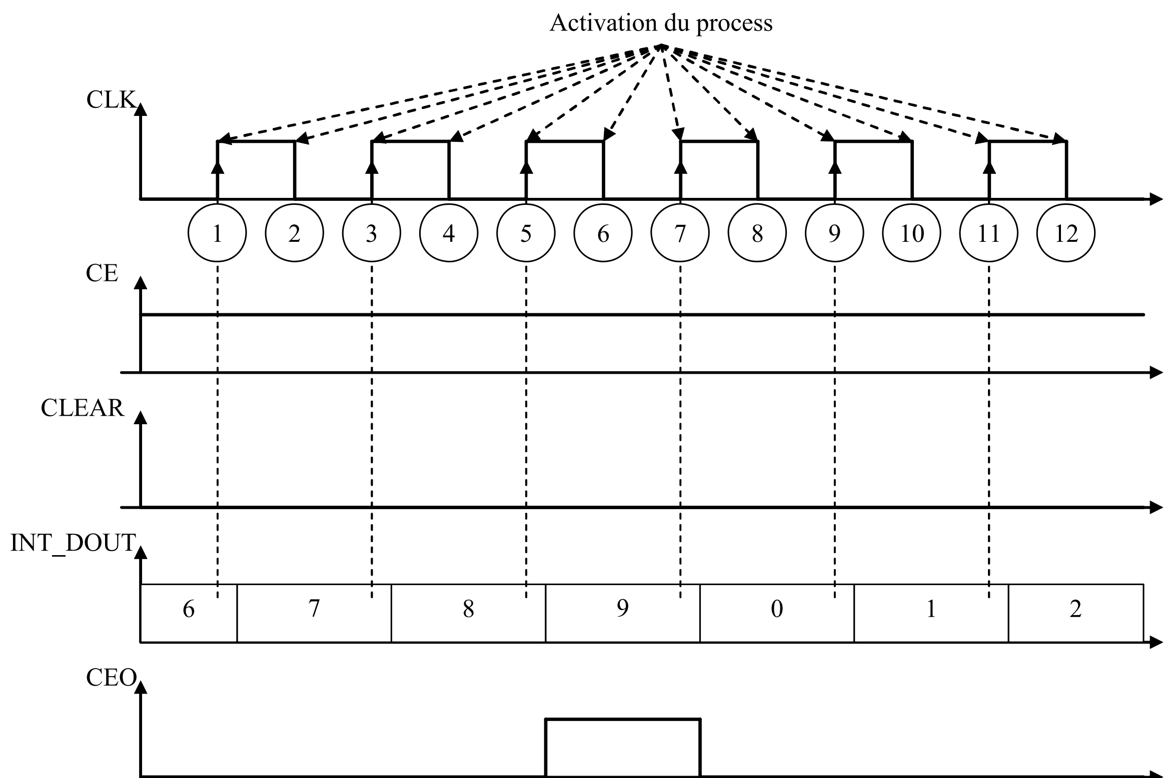
```
41. CEO <= CE when (INT_DOUT=STOP-1) else '0';
```

On obtient bien CEO = 1 si CE = 1 et INT_DOUT = STOP-1. La ligne étant hors process, la solution est purement combinatoire. On peut aussi essayer une solution séquentielle en modifiant le process de la manière suivante :

```
25.architecture a1 of compteur is
26.  signal INT_DOUT : std_logic_vector(DOUT'range);
27.begin
28.  process(CLK, CLEAR) begin
29.    if (CLEAR='1') then
30.      INT_DOUT <= (others => '0');
31.      CEO <= 0;
32.    elsif (CLK'event and CLK='1') then
33.      if (CE='1') then
34.        if (INT_DOUT=STOP-1) then
35.          INT_DOUT <= (others => '0');
36.        else
37.          INT_DOUT <= (INT_DOUT + 1);
38.        end if;
39.        if (INT_DOUT=STOP-2) then
40.          CEO <= 1;
41.        else
42.          CEO <= 0;
43.        end if;
44.      end if;
45.    end if;
46.  end process;
47.  dout <= int_dout;
48.end;
```

La compréhension est simple, mais les conditions sont un peu plus difficiles à comprendre. Pourquoi `if (INT_DOUT=STOP-2)` et pas `STOP-1`. Pour comprendre avec précision le problème, il suffit de tracer un chronogramme et de se rappeler que :

1. Le processus s'exécute à chaque changement d'état d'un des signaux auxquels il est déclaré sensible.
2. Les modifications apportées aux valeurs de signaux par les instructions prennent effet à la fin du processus.



Aux instants 2, 4, 6, 8, 10 et 12, le process est activé : comme CLEAR vaut 0, la première condition est fausse (ligne 29) et on passe au `elsif` (ligne 32). Il y a eu un événement sur CLK mais CLK vaut 0 donc la deuxième condition est fausse. On sort du process.

A l'instant 1, le process est activé : comme CLEAR vaut 0, on passe à la deuxième condition. Il y a eu un événement sur CLK et CLK vaut 1 donc la condition du `elsif` est vraie et on rentre dans la branche. Comme CE vaut 1, on va tester la valeur de INT_DOUT : mais attention, **c'est l'ancienne valeur de INT_DOUT que l'on va tester** (celle qui précède le front actif de l'horloge), c'est-à-dire 6. On suppose que STOP = 10. Le premier test, INT_DOUT = 9 est faux, donc on incrémente INT_DOUT mais **sa valeur ne change pas**

immédiatement : INT_DOUT vaut toujours 6. Le deuxième test, INT_DOUT = 8 est faux, donc CEO reste à 0. On sort du process et INT_DOUT passe à 7.

A l'instant 3, le process est activé. INT_DOUT vaut 7 et est incrémenté, CEO reste à 0. A la sortie du process, INT_DOUT passe à 8.

A l'instant 5, le process est activé. INT_DOUT vaut 8. Le premier test, INT_DOUT = 9 est faux, donc on incrémente INT_DOUT mais **sa valeur ne change pas** immédiatement : INT_DOUT reste à 8. Le deuxième test, INT_DOUT = 8 est vrai, donc CEO passera à 1 en sortant du process. On sort du process, INT_DOUT passe à 9 et CEO passe à 1.

A l'instant 7, le process est activé. INT_DOUT vaut 9. Le premier test, INT_DOUT = 9 est vrai, donc INT_DOUT passera à 0 en sortant du process : INT_DOUT reste à 9. Le deuxième test, INT_DOUT = 8 est faux, donc CEO passera à 0 en sortant du process. On sort du process, INT_DOUT passe à 0 et CEO passe à 0.

Aux instants 9 et 11, le process est activé. INT_DOUT sera incrémenté à la sortie du process et CEO reste à 0.

C'est ce principe qu'il faut comprendre pour savoir comment exprimer correctement les conditions dans un processus séquentiel.

5.3.2 Registre à décalage générique

Reprenons le registre à décalage vu au chapitre précédent et essayons de le rendre générique en réalisant un registre à N étages travaillant sur K bits.

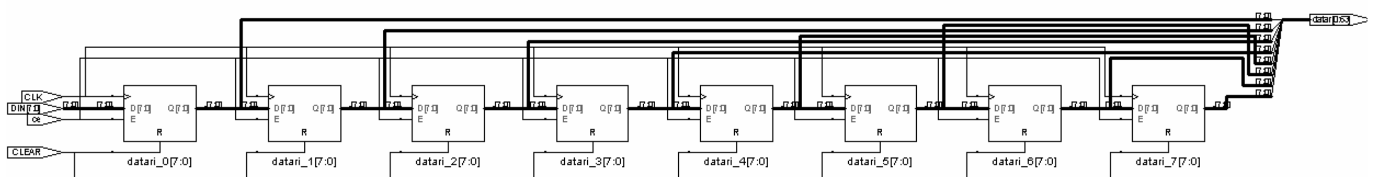
```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. package tp4_pkg is
4.   TYPE data8x8 IS ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);
5. end tp4_pkg;

6. library IEEE;
7. use IEEE.std_logic_1164.all;
8. use IEEE.std_logic_arith.all;
9. use IEEE.STD_LOGIC_UNSIGNED.all;
10.use work.tp4_pkg.all;
```

```
11.entity regMxN is
12.  generic (NbReg : integer :=8; NbBit : integer :=8);
13.  port( CLK : in std_logic ;
14.      CLEAR : in std_logic;
15.         CE : in std_logic;
16.        DIN : in std_logic_vector(NbBit -1 downto 0);
17.      datar : out data8x8);
18.end regMxN;

19.architecture RTL of regMxN is
20.  signal datari : data8x8;
21.begin
22.  process(CLK, CLEAR) begin
23.    if (CLEAR='1') then
24.      for i in 0 to 7 loop
25.        datari(i) <= (others => '0');
26.      end loop;
27.    elsif (CLK'event and CLK='1') then
28.      if (ce = '1') then
29.        for i in 1 to 7 loop
30.          datari(8-i) <= datari(7-i);
31.        end loop;
32.        datari(0) <= DIN;
33.      end if;
34.    end if;
35.  end process;
36.  datar <= datari;
37.end;
```

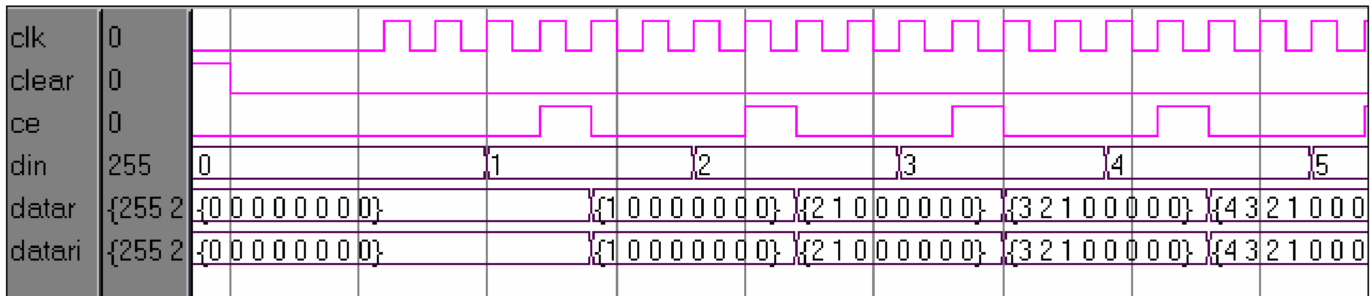
Le problème concernant la réalisation d'un registre N étages sur K bits, c'est qu'il va comporter N bus sur k bits en sortie. Il faut donc utiliser un tableau de N signaux de type `std_logic_vector(K-1 downto 0)`. Il est très difficile de rendre un tel tableau générique, d'autant que le type doit être connu avant de pouvoir l'utiliser dans l'entité. La solution la plus simple consiste à créer un nouveau type dans un package :

```
TYPE data8x8 IS ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);
```

Puis à l'utiliser dans l'entité du registre. Vous noterez à la ligne 10 que l'on peut utiliser un élément du package à un autre endroit dans le même package. Le type data8x8 définit un tableau de 8 signaux de largeur 8 bits. L'initialisation de ce signal se fait à l'aide d'une boucle for (lignes 24 à 26). Le décalage du registre est réalisé aux lignes 28 à 33. Si CE vaut 1, alors on copie datari(6) dans datari(7), puis datari(5) dans datari(6), puis datari(4) dans datari(5), …, puis datari(0) dans datari(1). Il suffit ensuite de copier din dans datari(0) pour terminer le décalage. Le design regMxN réalise donc la fonction logique suivante :



Le chronogramme suivant montre l'évolution des signaux avec les valeurs par défaut (8,8) :



Bien sur, ce composant n'est pas complètement générique puisque le tableau ne s'ajuste pas aux paramètres `NbReg` et `NbBit`. Il faut créer au démarrage un type dataNxK suffisamment grand pour contenir les valeurs. Les paramètres génériques ne peuvent résoudre tous les problèmes.

5.3.3 Conception avec plusieurs horloges ou avec plusieurs CE

5.3.3.1 Introduction

Plaçons-nous maintenant dans le cas d'un design qui comporte plusieurs parties fonctionnant à des rythmes différents. Il existe deux manières de traiter ce problème : la méthode localement synchrone (ou synchrone par bloc) avec autant d'horloges que de blocs travaillant à des fréquences différentes et la méthode totalement synchrone avec une seule horloge et plusieurs signaux de validation CE. Etudions plus en détail ces deux solutions :

1. Avec un circuit diviseur d'horloge, on peut créer à partir de l'horloge principale (généralement un oscillateur à quartz sur la carte) une nouvelle horloge qui sera utilisée dans le design. S'il y a dans le design plusieurs blocs qui travaillent à des fréquences différentes, on créera autant d'horloges que de blocs, chaque bloc étant synchrone avec son horloge mais pas avec les horloges des autres blocs. Cette méthode a pour avantage de minimiser la consommation du circuit, car chaque bascule travaille à sa fréquence minimale. Son principal inconvénient est que le design n'est plus synchrone avec une seule horloge, mais synchrone par bloc. Tout le problème (très délicat dans le cas général) va consister à échanger de manière fiable des données entre les différents blocs, chaque bloc travaillant avec sa propre horloge.
2. Il y a une autre manière de traiter le problème, la méthode totalement synchrone (fully synchronous). Tout le design travaille avec l'horloge du bloc le plus rapide, et on crée un

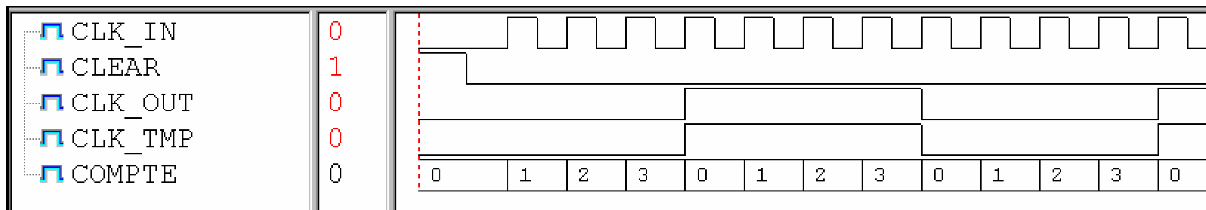
signal de validation (CE_x) pour chacun des blocs avec un circuit générateur de CE. Ce signal de validation qui vaut 1 toutes les N périodes de l'horloge sera connecté aux différentes bascules D des blocs. Les bascules seront donc activées à une cadence plus faible que l'horloge. Cette méthode a pour avantage de faciliter l'échange des données entre les blocs. En effet, tous les échanges sont synchrones puisqu'il n'y a plus qu'une seule horloge. Le seul inconvénient de cette méthode, c'est une consommation élevée puisque toutes les bascules du montage travaillent à la fréquence maximale.

Voyons maintenant les composants utilisés par ces deux méthodes.

5.3.3.2 circuit diviseur d'horloge

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.STD_LOGIC_UNSIGNED.all;

4. entity clk_div_N is
5.   generic (div : integer := 8);
6.   port (clk_in : in std_logic;
7.          clear : in std_logic;
8.        clk_out : out std_logic);
9. end clk_div_N;

10.architecture RTL of clk_div_N is
11.  signal clk_tmp : std_logic;
12.  signal compte : integer range 0 to (div/2)-1;
13.begin
14.  PROCESS (clk_in, clear) BEGIN
15.    if (clear = '1') then
16.      clk_tmp <= '0';
17.      compte <= 0;
18.    elsif (clk_in'event and clk_in='1') then
19.      if compte = ((div/2)-1) then
20.        compte <= 0;
21.        clk_tmp <= not clk_tmp;
22.      else
23.        compte <= compte + 1;
24.      end if;
25.    end if;
26.  END PROCESS;
27.  clk_out <= clk_tmp;
28.end;
```

A ce stade du cours, le fonctionnement de ce circuit diviseur d'horloge ne doit pas vous poser de problème de compréhension. Il ne fonctionne que pour des valeurs paires de `div`. Le signal `compte` évolue entre les valeurs 0 et div/2 – 1 (voyez sa déclaration à la ligne 12). Vous noterez qu'il s'agit d'un entier et pas d'un type std_logic. Cela permet une réalisation plus simple du diviseur. Quand `compte` atteint la valeur finale `div/2 - 1`, le signal `clk_tmp` est inversé. Le chronogramme suivant explique le fonctionnement du circuit avec une division par 8 (valeur par défaut) :

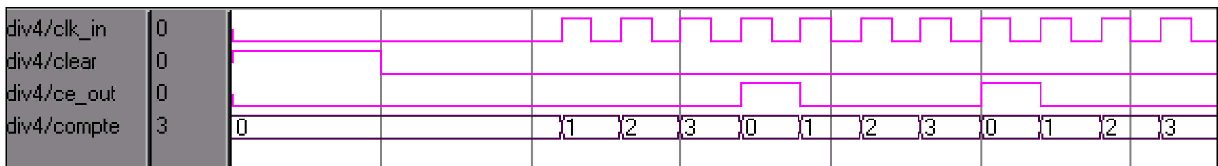


Si vous utilisez le type `std_logic`, les bascules sont toutes à l'état U (unknown) au démarrage. Vous devez obligatoirement prévoir une mise à zéro ou bien une mise à un pour pouvoir utiliser votre design. C'est un avantage de ce type vis à vis du type bit car il est proche du fonctionnement réel d'un circuit intégré. En effet, à la mise sous tension, les éléments de mémorisation (dont les bascules) sont dans un état indéterminé. **Vous ne devez jamais supposer que l'état de départ de vos bascules est connu quand vous concevez un design.** L'emploi du type `std_logic` vous y oblige, d'où son intérêt.

5.3.3.3 Circuit générateur de CE

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.STD_LOGIC_UNSIGNED.all;

4. entity gen_ce_div_N is
5.   generic (div : integer := 4);
6.   port (clk_in : in std_logic;
7.          clear : in std_logic;
8.         ce_out : out std_logic);
9. end gen_ce_div_N;

10.   architecture RTL of gen_ce_div_N is
11.     signal compte : integer range 0 to div-1;
12.   begin
13.     PROCESS (clk_in, clear) BEGIN
14.       if (clear = '1') then
15.         compte <= 0;
16.         ce_out <= '0';
17.       elsif (clk_in'event and clk_in = '1') then
18.         if (compte = div-1) then
19.           ce_out <= '1';
20.           compte <= 0;
21.         else
22.           ce_out <= '0';
23.           compte <= compte + 1;
24.         end if;
25.       end if;
26.     END PROCESS;
27.   end;
```

Le fonctionnement du montage est évident. ce_out vaut 0 sauf quand compte est égal à la valeur div-1 (3 dans le chronogramme suivant). Il passe alors à 1.

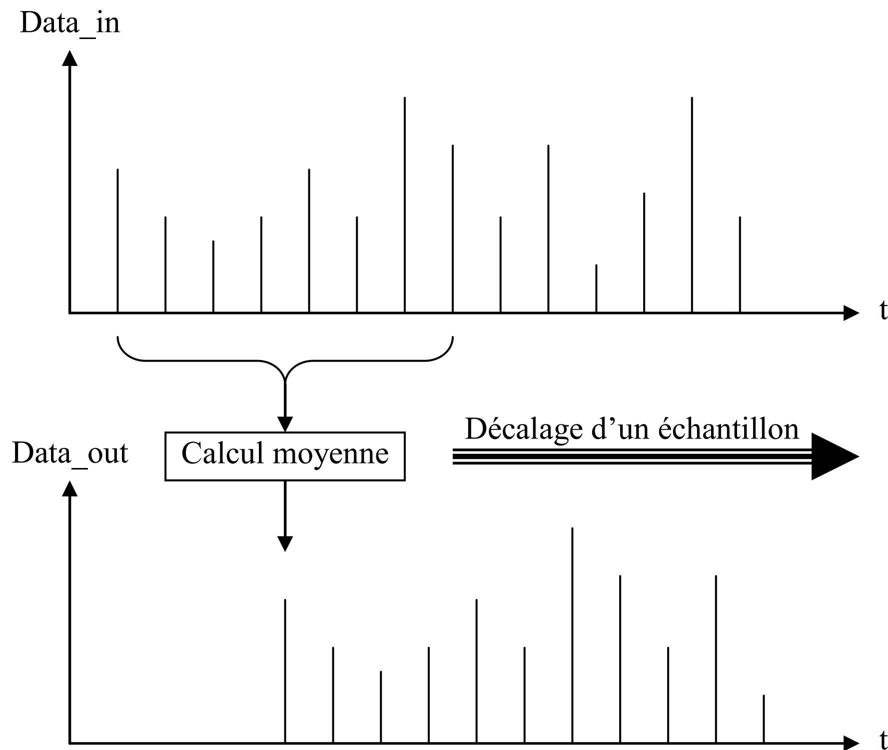


Une bascule D reliée à l’horloge clk_in dont le chip enable (CE) serait connecté à ce_out serait donc activée un front d’horloge sur 4.

5.3.4 Calcul de moyenne mobile

Nous avons maintenant vu tous les éléments qui vont nous permettre de créer des montages plus élaborés. A l’aide d’un registre 8 étages, d’un multiplexeur 8 vers 1, d’un accumulateur 8 bits et d’une machine d’état, nous allons réaliser une moyenne mobile avec des échantillons codés sur 8 bits.

Le but est de faire la moyenne des 8 derniers échantillons à chaque nouvel échantillon entrant sur `data_in`. Cela revient à faire une moyenne glissante avec une fenêtre de largeur 8.



Nous allons mettre dans un package `mobile_pkg.vhd` les 4 composants que nous allons instancier dans le top level `mobile.vhd`. Voici le package :

```
library IEEE;
use IEEE.std_logic_1164.all;

package mobile_pkg is

      TYPE data8x8 IS ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);

      COMPONENT regMxN
          generic (NbReg : integer :=8; NbBit : integer :=8);
          port( CLK : in std_logic ;
              CLEAR : in std_logic;
              ce : in std_logic;
                   DIN : in std_logic_vector(NbBit -1 downto 0);
                    datar : out data8x8);
      END COMPONENT;

      component Mux8vers1_8b
            port (DataIn : in  data8x8;
               MuxSelect : in  std_logic_vector(2 downto 0);
                  MuxOut : out std_logic_vector(7 downto 0) );
      end component;
```

```
      component accu_som is
            port( CLK : in std_logic ;
             CE : in std_logic ;
          CLEAR : in std_logic;
              X : in std_logic_vector(7 downto 0);
          Somme : out std_logic_vector(10 downto 0));
      end component;

      COMPONENT machine_etat
          port( CLK : in std_logic ;
              RESET : in std_logic;
            cesy : out std_logic;
           cemux : out std_logic_vector(2 downto 0);
            ceac : out std_logic);
      END COMPONENT;

end mobile_pkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.mobile_pkg.all;

entity regMxN is
    generic (NbReg : integer :=8; NbBit : integer :=8);
    port( CLK : in std_logic ;
        CLEAR : in std_logic;
           ce : in std_logic;
          DIN : in std_logic_vector(NbBit -1 downto 0);
        datar : out data8x8);
end regMxN;

architecture RTL of regMxN is
  signal datari : data8x8;
begin
  process(CLK, CLEAR) begin
    if (CLEAR='1') then
      for i in 0 to 7 loop
        datari(i) <= (others => '0');
      end loop;
    elsif (CLK'event and CLK='1') then
      if (ce = '1') then
        for i in 1 to 7 loop
          datari(8-i) <= datari(7-i);
        end loop;
        datari(0) <= DIN;
      end if;
    end if;
  end process;
  datar <= datari;
end;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.mobile_pkg.all;

entity Mux8vers1_8b is
      port (DataIn : in  data8x8;
         MuxSelect : in  std_logic_vector(2 downto 0);
            MuxOut : out std_logic_vector(7 downto 0) );
end Mux8vers1_8b;
architecture a1 of Mux8vers1_8b is
begin
```

```
    Muxout <= datain(CONV_INTEGER(UNSIGNED(Muxselect)));
end;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity accu_som is
  port( CLK : in std_logic ;
         CE : in std_logic ;
      CLEAR : in std_logic;
          X : in std_logic_vector(7 downto 0);
      Somme : out std_logic_vector(10 downto 0));
end accu_som;

architecture a1 of accu_som is
  signal Si : std_logic_vector(Somme'range) ;
  signal Somme_int : std_logic_vector(Somme'range) ;
begin

  Si <= "000"&X + Somme_int;
  Somme <= Somme_int;

  process(CLK) begin
    if (CLK'event and CLK='1') then
      if (CLEAR='1') then
            Somme_int <= (others => '0');
      elsif (CE='1') then
        Somme_int <= Si;
      end if;
    end if;
  end process;

end;

library IEEE;
use IEEE.std_logic_1164.all;

entity machine_etat is
      port( CLK : in std_logic ;
       RESET : in std_logic;
        cesy : out std_logic;
       cemux : out std_logic_vector(2 downto 0);
        ceac : out std_logic);
END machine_etat;

architecture a1 of machine_etat is
      TYPE state_type IS (STATE0, STATE1, STATE2, STATE3, STATE4, STATE5,
                          STATE6, STATE7, STATE8, STATE9);
      SIGNAL sreg : state_type;
BEGIN

      PROCESS (CLK, RESET)
      BEGIN
            IF ( RESET='1' ) THEN
                  sreg <= STATE0;
                  cesy <= '1';
                  cemux <= "000";
                  ceac <= '0';
            ELSIF CLK='1' AND CLK'event THEN

                  CASE sreg IS
                        WHEN STATE0 =>
                              sreg<=STATE1;
```

```
                        cesy <= '1';
                        cemux <= "000";
                        ceac <= '0';
                  WHEN STATE1 =>
                        sreg<=STATE2;
                        cesy <= '0';
                        cemux <= "000";
                        ceac <= '1';
                  WHEN STATE2 =>
                        sreg<=STATE3;
                        cesy <= '0';
                        cemux <= "001";
                        ceac <= '1';
                  WHEN STATE3 =>
                        sreg<=STATE4;
                        cesy <= '0';
                        cemux <= "010";
                        ceac <= '1';
                  WHEN STATE4 =>
                        sreg<=STATE5;
                        cesy <= '0';
                        cemux <= "011";
                        ceac <= '1';
                  WHEN STATE5 =>
                        sreg<=STATE6;
                        cesy <= '0';
                        cemux <= "100";
                        ceac <= '1';
                  WHEN STATE6 =>
                        sreg<=STATE7;
                        cesy <= '0';
                        cemux <= "101";
                        ceac <= '1';
                  WHEN STATE7 =>
                        sreg<=STATE8;
                        cesy <= '0';
                        cemux <= "110";
                        ceac <= '1';
                  WHEN STATE8 =>
                        sreg<=STATE9;
                        cesy <= '0';
                        cemux <= "111";
                        ceac <= '1';
                  WHEN STATE9 =>
                        sreg<=STATE0;
                        cesy <= '0';
                        cemux <= "000";
                        ceac <= '0';
                  WHEN OTHERS =>
                        sreg<=STATE0;
                        cesy <= '0';
                        cemux <= "000";
                        ceac <= '0';
            END CASE;

         END IF;
      END PROCESS;
end;
```

Vous avez normalement tous les éléments pour comprendre le fonctionnement des composants du package sauf la machine d'état dont nous verrons plus loin le fonctionnement. Le top level design instancie les 4 composants du package et décrit un registre 8 bits en sortie

pour sauver le résultat du calcul de la moyenne. `CLK_CAN` et `CLK_CNA` sont les 2 horloges des convertisseurs analogique-numérique (CAN) et numérique-analogique (CNA).

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.mobile_pkg.all;

entity mobile is
    port(
        H50_I : in std_logic;
        Reset : in std_logic;
        data_in : in std_logic_vector(7 downto 0);
        CLK_CAN : out std_logic;
        CLK_CNA : out std_logic;
        data_out : out std_logic_vector(7 downto 0));
end mobile;

architecture comporte of mobile is
   signal data : data8x8;
   signal ce_symb : std_logic;
   signal ce_accu : std_logic;
   signal MuxSel : std_logic_vector(2 downto 0);
   signal Sum : std_logic_vector(10 downto 0);
   signal data_o : std_logic_vector(7 downto 0);
begin
  CLK_CAN <= ce_symb; -- horloge CAN
  CLK_CNA <= ce_symb; -- horloge CNA
  meta : machine_etat port map(H50_I, Reset, ce_symb, MuxSel, ce_accu);
  mux8 : Mux8vers1_8b port map(data, MuxSel, data_o);
  reg8 : regMxN generic map(8,8) port map(H50_I, Reset, ce_symb, data_in, data);
  accu : accu_som port map(H50_I, ce_accu, ce_symb, data_o, Sum);

  process(H50_I, Reset) begin -- registre 8 bits de sortie
    if (Reset='1') then
      data_out <= (others => '0');
    elsif (H50_I'event and H50_I='1') then
      if (ce_symb = '1') then
        data_out <= Sum(10 downto 3);
      end if;
    end if;
  end process;

end comporte ;
```

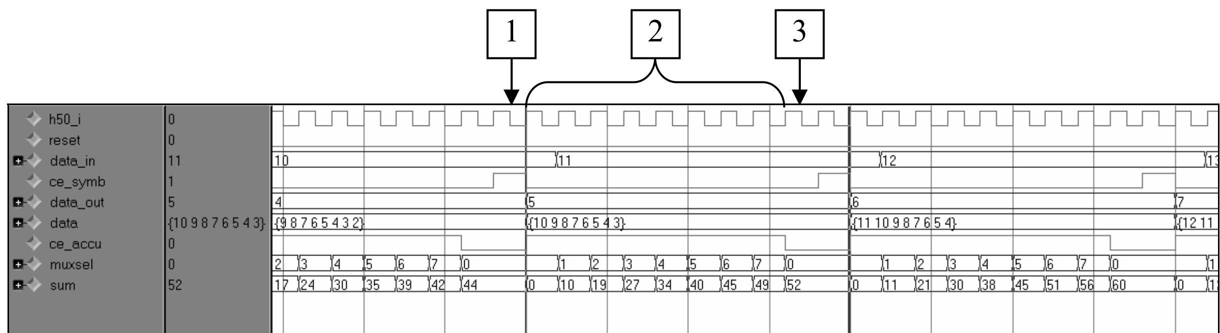
Le séquencement nécessaire pour le calcul de la moyenne est le suivant :

1) Il se produit trois évènements en même temps : lecture d'un échantillon sur `data_in` et décalage des échantillons stockés dans le registre 8x8 ; le registre de sortie lit les 8 bits de poids fort de la valeur stockée dans l'accumulateur (donc division par 8) ; le registre de stockage de l'accumulateur est remis à 0 pour pouvoir démarrer un nouveau calcul. Le signal de validation du registre `ce_symb` vaut 1 pendant une période d'horloge (`H50_I` : horloge 50 MHz sur la carte) et déclenche ces trois actions.

2) Lecture des 8 valeurs stockées dans le registre via le multiplexeur 8 vers 1 et calcul de leur somme. Il faut 8 périodes d'horloge pour que le signal de sélection du multiplexeur, `MuxSel`, passe de 0 à 7. Pendant ce temps, le signal de validation de l'accumulateur, `ce_accu`, vaut 1 pour accumuler la somme des échantillons.

3) Un temps mort d'une période d'horloge pendant lequel `ce_symb`, `MuxSel`, et `ce_accu` sont à 0. Il faut donc 10 périodes d'horloge pour traiter un échantillon, donc fech = 50 MHz/10 = 5 Mech/s.

Le chronogramme suivant montre ces trois phases :

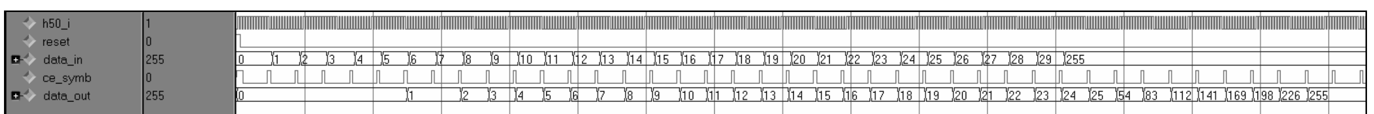


Phase 1 : lecture de la valeur 10 sur `data_in` et stockage (après décalage) dans la variable `data` qui contient (10, 9, 8, 7, 6, 5, 4, 3) ; lecture des 8 bits de poids fort de la valeur de la moyenne précédente (`sum = 44`) et copie dans `data_out` (valeur 5) ; mise à 0 de `sum`.

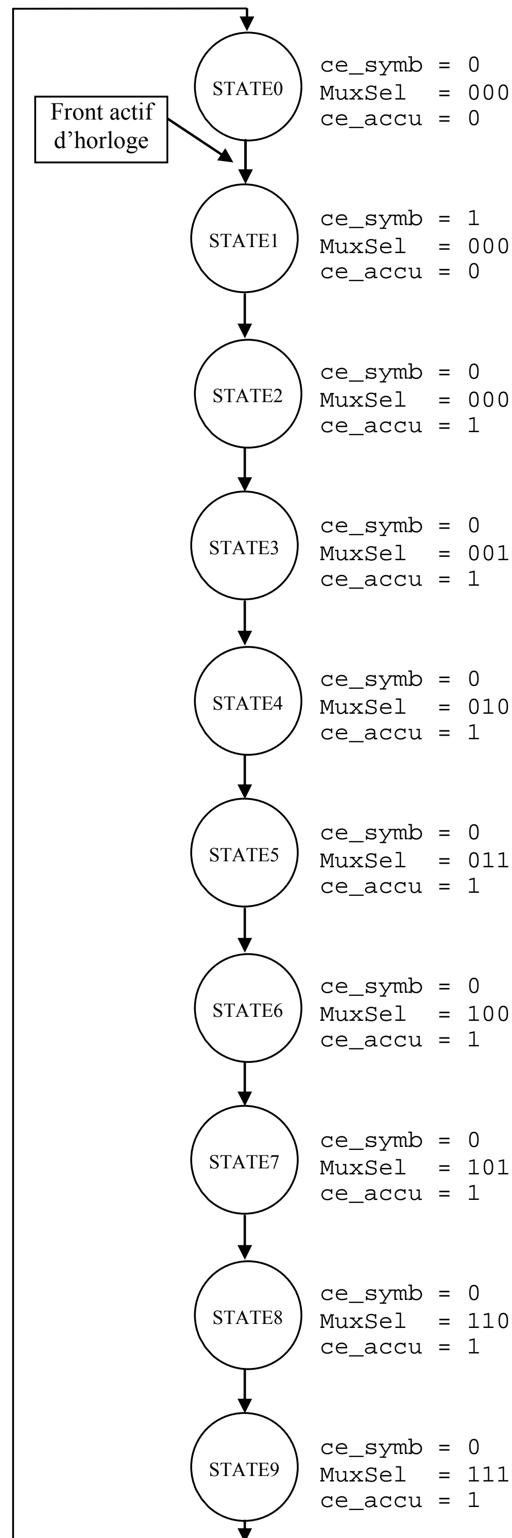
Phase 2 : calcul de la somme de la valeur des échantillons stockés, `sum` = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 = 52. `MuxSel` va de 0 à 7 et `ce_accu` vaut 1.

Phase 3 : temps mort. Une période d'horloge pendant laquelle `ce_symb`, `MuxSel`, et `ce_accu` sont à 0. `sum` contient 52 ce qui donnera 6 en sortie lors de la phase 1 suivante.

Le chronogramme suivant montre le résultat en sortie du montage sur plusieurs phases successives :



Le rôle des différents éléments étant expliqué, il reste à détailler le fonctionnement de la machine d'état qui met en œuvre le séquencement. Le diagramme suivant contient 10 états qui seront parcourus successivement et aucune entrée ne viendra modifier la séquence. Il s'agit en fait d'un générateur de séquence synchrone plus que d'une machine d'état :



Voyons maintenant le code VHDL correspondant :

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity machine_etat is
4.   port( CLK : in std_logic ;
5.       RESET : in std_logic;
6.        cesy : out std_logic;
7.       cemux : out std_logic_vector(2 downto 0);
8.        ceac : out std_logic);
9. END machine_etat;

10. architecture a1 of machine_etat is
11.   TYPE state_type IS (STATE0, STATE1, STATE2, STATE3, STATE4, STATE5,
                        STATE6, STATE7, STATE8, STATE9);
12.   SIGNAL sreg : state_type;
13. BEGIN

14.   PROCESS (CLK, RESET)
15.   BEGIN
16.     IF (RESET='1') THEN
17.       sreg <= STATE0;
18.       cesy <= '1';
19.       cemux <= "000";
20.       ceac <= '0';
21.     ELSIF (CLK='1' AND CLK'event) THEN

22.       CASE sreg IS
23.         WHEN STATE0 =>
24.           sreg<=STATE1;
25.           cesy <= '1';
26.           cemux <= "000";
27.           ceac <= '0';
28.         WHEN STATE1 =>
29.           sreg<=STATE2;
30.           cesy <= '0';
31.           cemux <= "000";
32.           ceac <= '1';
33.           ...
34.         WHEN STATE9 =>
35.           sreg<=STATE0;
36.           cesy <= '0';
37.           cemux <= "000";
38.           ceac <= '0';
39.         WHEN OTHERS =>
40.           sreg<=STATE0;
41.           cesy <= '0';
42.           cemux <= "000";
43.           ceac <= '0';
44.         END CASE;

45.       END IF;
46.     END PROCESS;
47. end;
```

Préparation de l'état suivant. Cesy =1 pendant STATE1 et pas pendant STATE0 (lignes 24 à 27)

On crée à la ligne 11 un type énuméré appelé `state_type` et on crée un signal `sreg` de ce type. `sreg` ne peut prendre que les états STATE0 à STATE9 qui seront traduit par le synthétiseur soit par un compteur binaire, soit par un compteur 1 parmi N ou encore par un compteur gray. Cela fait partie des paramètres du synthétiseur concernant les machines d'état.

A la ligne17, on initialise `sreg` à STATE0 puis on crée un `case` dans la branche sensible à l'horloge.

Lignes 23 à 27, si `sreg` vaut STATE0 alors `sreg` = STATE1 et on met `ce_symb` à 1, `MuxSel` et `ce_accu` à 0. En sortant du process, `sreg` passe à STATE1.

Lignes 28 à 32, si `sreg` vaut STATE1 alors `sreg` = STATE2 et on met `ce_symb` à 0, `MuxSel` à 0 et `ce_accu` à 1. En sortant du process, `sreg` passe à STATE2.

On va enchainer à chaque front d'horloge les états successifs du diagramme jusqu'à STATE9 à la ligne 34 où `sreg` repasse à STATE0. C'est le temps mort.

Si le registre d'état est codé en binaire (méthode par défaut en général), il va utiliser un compteur 4 bits. Comme on utilise que 10 états, les valeurs 10 à 15 seront inutilisées. Il ne faudrait pas être dans un état indéterminé si par malchance on se retrouvait avec une de ces valeurs inutilisées dans le registre d'état. Pour éviter tout problème, on ajoute une branche OTHERS dans le case (ligne 39 à 43) qui fait passer sreg à STATE0 et qui met `ce_symb`, `MuxSel` et `ce_accu` à 0. C'est une initialisation en cas d'état non prévu dans le diagramme.

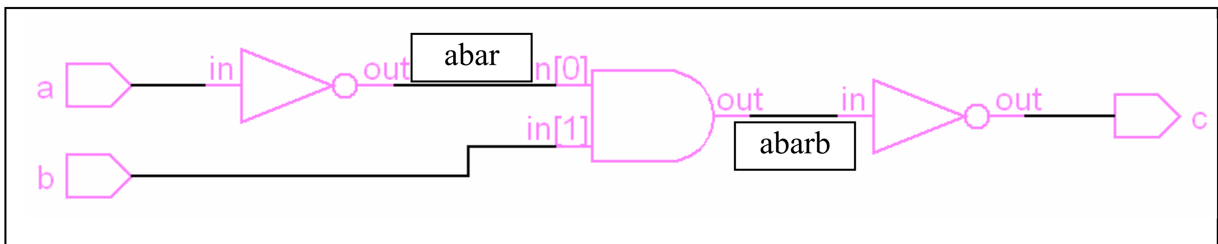
On pourrait améliorer notre montage dont la principale faiblesse consiste en l'utilisation d'un registre à décalage associé à un multiplexeur pour stocker et lire les N derniers échantillons. Si N devient grand, quelques centaines par exemple, ce type de montage devient très difficile à implémenter à cause de la taille croissante du multiplexeur. On utilisera plutôt une mémoire de type SRAM synchrone pour stocker les échantillons. La lecture et le décalage des échantillons stockés s'effectuera en même temps ce qui économisera des bascules et un multiplexeur de grande taille. Il ne serait pas très difficile de modifier la structure du montage pour multiplier chaque échantillon par un coefficient et réaliser ainsi un FIR MAC.

Avec ce type de filtre, il faudra au moins une période d'horloge par coefficient pour traiter un échantillon.

6 Simulation et testbench

6.1 Principe de la simulation fonctionnelle

Abordons maintenant la simulation du design. Nous avons jusqu'à présent utilisé en TP un utilitaire qui nous a permis de créer sous forme graphique les vecteurs de test à connecter sur les entrées du montage, puis grâce à ISE, la simulation s'est déroulée de manière automatique (avec un aspect un peu magique) afin d'obtenir les chronogrammes de sortie. La réalité est bien plus complexe. Il faut comprendre comment se déroule le temps de simulation dans le circuit modélisé. Avant le changement sur une entrée, la valeur actuelle du temps s'appelle `now`. Cet évènement va causer des changements à l'intérieur du circuit jusqu'à une situation future qui doit normalement être stable (sinon, le circuit oscille, ce que nous verrons tout à l'heure). Prenons un exemple simple :



La description en VHDL de ce circuit est la suivante :

```
entity essai is
     port(a, b : in bit;
             c : out bit);
end essai;


architecture beh of essai is
     signal abar : bit;
     signal abarb : bit;
begin
     abarb <= abar and b;
     abar <= not a;
     c <= not abarb;
end beh;
```

Nous n'utiliserons pas ISE dans ce chapitre, mais seulement le simulateur VHDL ModelSim. En l'état, ce design ne peut pas être simulé. Il faut d'abord créer un testbench qui va appeler `essai` comme si c'était un composant :

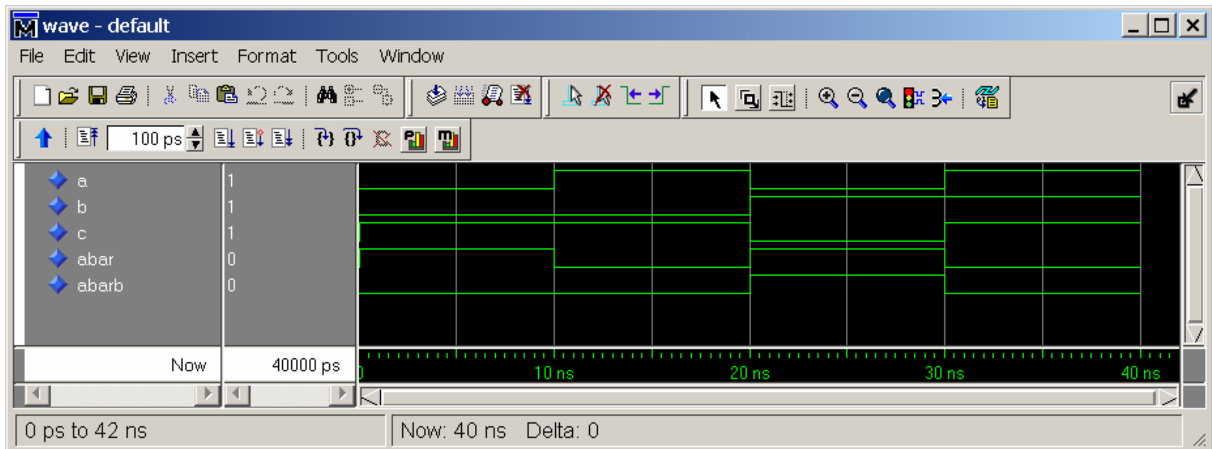
```
ENTITY essai_tb  IS
END;

ARCHITECTURE essai_tb_arch OF essai_tb IS
  SIGNAL a, b, c   :  bit ;
  COMPONENT essai
    PORT (c  : out bit;
          a  : in bit;
          b  : in bit );
  END COMPONENT;
BEGIN
  DUT  : essai      -- DUT (Device Under Test) est le
    PORT MAP (      -- nom d'instance du composant essai
      c   => c,
      a   => a,
      b   => b);
END;
```

Nom de l'entité

Vous noterez que l'entité du testbench est vide, car il n'a aucune entrée/sortie. Son seul but est d'instancier `essai`. Nous verrons plus tard comment créer les stimuli d'entrée dans le testbench. Dans un premier temps, nous les créerons avec ModelSim. Les commandes suivantes sont tapées directement dans la fenêtre de commande du simulateur.

```
1    # création de la librairie work
2    vlib work
3
4    # compilation du design
5    vcom -93 essai.vhd
6
7    # compilation du testbench
8    vcom -93 essai_tb.vhd
9
10   # lancement du simulateur et chargement du testbench
11   vsim -t 1ps -lib work  essai_tb
12
13   # affichage des fenêtres wave et list
14   view wave list
15
16   # on visualise tous les signaux dans dut
17   add wave dut/*
18   add list dut/*
19
20   # on applique les stimuli sur les entrées du composant
21   force a 0
22   force b 0
23   run 10 ns
24   force a 1
25   run 10 ns
26   force a 0
27   force b 1
28   run 10 ns
29   force a 1
30   run 10 ns
```

Le chronogramme obtenu dans la fenêtre `wave` est le suivant :



Nous retrouvons bien les changements sur les entrées ainsi que la réponse attendue du montage. Il est aussi possible d'obtenir du simulateur une liste des évènements qui affectent les différents signaux du design. C'est la fenêtre `list` :

list

ps	delta	/essai_tb/dut/a	/essai_tb/dut/b	/essai_tb/dut/c	/essai_tb/dut/abar	/essai_tb/dut/abarb
0	+0	0	0	0	0	0
0	+1	0	0	1	1	0
10000	+0	1	0	1	1	0
10000	+1	1	0	1	0	0
20000	+0	0	1	1	0	0
20000	+1	0	1	1	1	0
20000	+2	0	1	1	1	1
20000	+3	0	1	0	1	1
30000	+0	1	1	0	1	1
30000	+1	1	1	0	0	1
30000	+2	1	1	0	0	0
30000	+3	1	1	1	0	0

Dans la partie gauche de la fenêtre, nous trouvons d'abord le temps de la simulation (en ps), puis pour un temps donné les cycles de simulation qu'on appelle « **cycles delta** ». Sur la partie droite, nous trouvons les différents états des signaux du design.

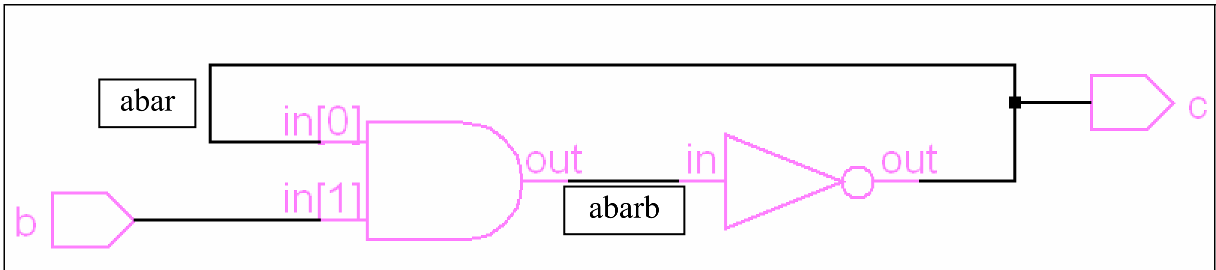
Il faut maintenant regarder les enchaînements des causes et des conséquences. Durant cette simulation, il y a 4 évènements sur les entrées (ce sont les causes), puis des conséquences à l'intérieur du design. Les cycles delta s'arrêtent quand le design est stable. **Tous les signaux du montage démarrent à 0**. Par précaution, nous allons forcer a et b à 0 au début de la simulation (même si c'est inutile dans ce cas).

temps	évènement	Delta	
0 ns	démarrage a = 0, b = 0	+0 +1	Tous les signaux valent 0. abar passe à 1, c passe à 1, le design est stable.
10 ns	a = 1	+0 +1	application de a = 1 sur `essai`. abar passe à 0, le design est stable.
20 ns	a = 0, b = 1	+0 +1 +2 +3	application de a = 0 et b = 1 sur `essai`. abar passe à 1. abarb passe à 1. c passe à 0, le design est stable.
30 ns	a = 1	+0 +1 +2 +3	application de a = 1 sur `essai`. abar passe à 0. abarb passe à 0. c passe à 1, le design est stable.

A t = 30 ns, l'entrée a passe à 1, provoquant une cascade d'évènements simultanés, mais dont le traitement correct, pour respecter les chaines de causalité, nécessite 4 cycles delta.

De delta en delta, le simulateur évolue de la situation actuelle (`now`) vers la situation future où le montage est stable. On pourrait, pour faciliter la compréhension, imaginer que delta représente un temps de propagation élémentaire dans une porte mais c'est faux. Il n'y a pas de temps de propagation dans les portes en simulation fonctionnelle et l'accumulation des cycles delta correspond toujours à un temps nul. Les cycles delta servent uniquement à assurer un traitement correct de la causalité.

Rien n'assure d'ailleurs que le processus de calcul de la situation future (où le montage doit être stable) converge. Modifions l'exemple précédent :



La description en VHDL du nouveau circuit est :

```
entity essai is
     port(b : in bit;
          c : out bit);
end essai;


architecture beh of essai is
     signal abar : bit;
     signal abarb : bit;
begin
     abarb <= abar and b;
     abar <= not abarb;
     c <= not abarb;
end beh;
```

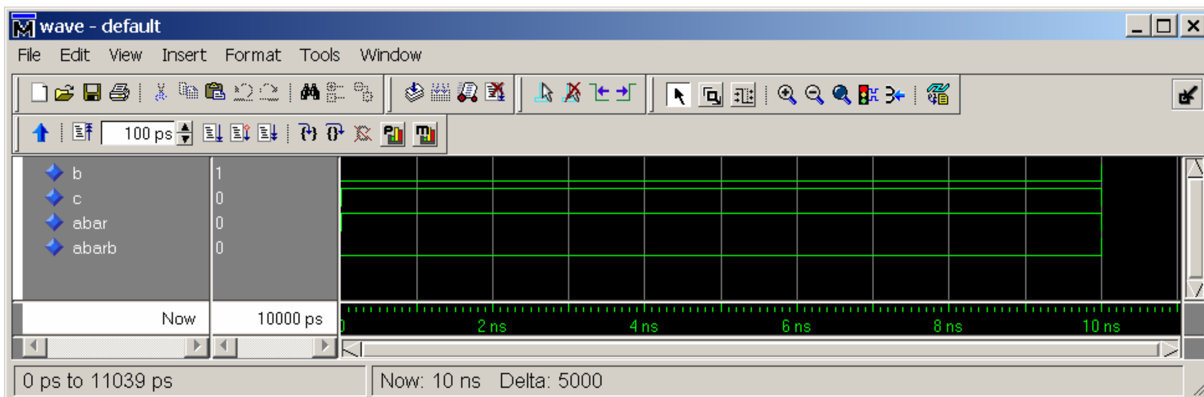
Le testbench suivant instancie le composant `essai` :

```
ENTITY essai_tb  IS
END ;

ARCHITECTURE essai_tb_arch OF essai_tb IS
  SIGNAL b, c : bit;
  COMPONENT essai
    PORT (
      c  : out bit;
      b  : in bit );
  END COMPONENT ;
BEGIN
  DUT  : essai
    PORT MAP (
      c   => c,
      b   => b);
END;
```

Les commandes suivantes sont tapées directement dans la fenêtre de commande du simulateur :

```
1     # création de la librairie work
2     vlib work
3
4     # compilation du design
5     vcom -93 essai.vhd
6
7     # compilation du testbench
8     vcom -93 essai_tb.vhd
9
10    # lancement du simulateur et chargement du testbench
11    vsim -t 1ps -lib work  essai_tb
12
13    # affichage des fenêtres wave et list
14    view wave list
15
16    # on visualise tous les signaux dans dut
17    add wave dut/*
18    add list dut/*
19
20    # on applique les stimuli sur les entrées du composant
21    force b 0
22    run 10 ns
23    force b 1
24    run 10 ns
```

Le chronogramme obtenu dans la fenêtre wave est le suivant :



Lors de la simulation de ce montage, le temps semble s'arrêter après 10 ns lorsque b passe à 1. Le simulateur a émis un message d'erreur nous signalant qu'il a atteint le nombre maximum de cycles delta qu'il est capable de simuler à t = 10 ns.

```
# ** Error: (vsim-3601) Iteration limit reached at time 10 ns.
```

Cette limite est fixée arbitrairement à 1000 et sert uniquement à détecter les oscillations. La liste de évènements nous confirme que la sortie c oscille indéfiniment entre les états 0 et 1 :

list

ps	delta	/essai_tb/dut/b	/essai_tb/dut/c	/essai_tb/dut/abar	/essai_tb/dut/abarb
0	+0	0	0	0	0
0	+1	0	1	1	0
10000	+0	1	1	1	0
10000	+1	1	1	1	1
10000	+2	1	0	0	1
10000	+3	1	0	0	0
10000	+4	1	1	1	0
10000	+5	1	1	1	1
10000	+6	1	0	0	1
10000	+7	1	0	0	0
10000	+8	1	1	1	0
10000	+9	1	1	1	1
10000	+10	1	0	0	1

La réalité physique est conforme à la simulation. Ce type de montage est un oscillateur dont la période est égale au temps de propagation dans les deux portes (plus les interconnections bien sur).

Revenons au premier design. Vous avez noté que le fonctionnement ne dépend par de l'ordre dans lequel on a écrit les instructions. Nous les avons d'ailleurs volontairement écrites dans le désordre pour bien indiquer qu'il n'y a aucun lien entre l'ordre d'écriture et une quelconque chronologie :

```
abarb <= abar and b;
abar <= not a;
c <= not abarb;
```

On indique simplement que :

- `abarb` est sensible aux évènements qui affectent `abar` et `b`,
- `abar` est sensible aux évènements qui affectent `a`,
- `c` est sensible aux évènements qui affectent `abarb`.

Et on décrit simultanément les expressions qui calculent les nouvelles valeurs des signaux. Ces expressions sont réévaluées chaque fois qu'un évènement affecte l'un de leurs opérandes.

Cela ressemble beaucoup à la notion de processus que nous avons vu depuis le chapitre 2. La conclusion est évidente : chacune des instructions est un processus déclaré de manière implicite. Le design peut être réécrit d'une manière strictement équivalente en faisant apparaitre les processus de manière explicite, chaque processus étant sensible aux signaux se trouvant à droite de l'affectation :

```
entity essai is
     port(a, b : in bit;
             c : out bit);
end essai;

architecture beh of essai is
     signal abar : bit;
     signal abarb : bit;
begin

   process(abar, b) begin
      abarb <= abar and b;
   end process;

   process(a) begin
      abar <= not a;
   end process;

   process(abarb) begin
        c <= not abarb;
   end process;

end beh;
```

La liste des évènements confirme bien cette équivalence :

ps	delta	/essai_tb/dut/a	/essai_tb/dut/b	/essai_tb/dut/c	/essai_tb/dut/abar	/essai_tb/dut/abarb
0	+0	0	0	0	0	0
0	+1	0	0	1	1	0
10000	+0	1	0	1	1	0
10000	+1	1	0	1	0	0
20000	+0	0	1	1	0	0
20000	+1	0	1	1	1	0
20000	+2	0	1	1	1	1
20000	+3	0	1	0	1	1
30000	+0	1	1	0	1	1
30000	+1	1	1	0	0	1
30000	+2	1	1	0	0	0
30000	+3	1	1	1	0	0

Que se passe-t-il si plusieurs process (implicites ou explicites) sont sensibles aux mêmes signaux ? C'est le cas par exemple des process synchrones qui peuvent se trouver dans plusieurs composants. Tous les process d'un design (hierarchique ou non) ayant la même liste de sensibilité sont activés simultanément, donc dans le même cycle delta. Prenons un exemple simple ayant deux process sensibles à l'horloge :

```
entity essai is
     port(din : in bit;
          clk : in bit;
           dout : out bit);
end essai;

architecture beh of essai is
     signal dinr : bit;
begin

   process(clk) begin
      if (clk'event and clk = '1') then
         dinr <= din;
      end if;
   end process;

   process(clk) begin
      if (clk'event and clk = '1') then
         dout <= dinr;
      end if;
   end process;

end beh;
```

Le testbench instancie le design :

```
ENTITY essai_tb  IS
END ;

ARCHITECTURE essai_tb_arch OF essai_tb IS
  SIGNAL din, dout, clk : bit;
  COMPONENT essai
     port(din : in bit;
          clk : in bit;
           dout : out bit);
  END COMPONENT ;
BEGIN
  DUT  : essai
    PORT MAP (
      din   => din,
      dout   => dout,
      clk   => clk);
END;
```

Les commandes suivantes sont envoyées au simulateur :

```
# création de la librairie work
vlib work

# compilation du design
vcom -93 multip.vhd

# compilation du testbench
vcom -93 multip_tb.vhd

# lancement du simulateur et chargement du testbench
vsim -t 1ps -lib work essai_tb

# affichage de la fenêtre list
view list

# on visualise les signaux dans dut
add list dut/*

# on génère les stimuli suivants
force din 1
force clk 0
run 10 ns
force clk 1
run 10 ns
force clk 0
run 10 ns
force clk 1
run 10 ns
```

La liste des évènements montre le fonctionnement d'un registre à décalage. Au temps 10000 ps, cycle 0, l'horloge monte et au cycle1, les deux process sont activés : din est copié sur dinr et simultanément dinr est copié sur dout, dout restant donc à 0. C'est sur le deuxième front montant (à t = 30000 ps) que dout passe à 1.

ps	delta	/essai_tb/dut/din	/essai_tb/dut/clk	/essai_tb/dut/dout	/essai_tb/dut/dinr
0	+0	1	0	0	0
10000	+0	1	1	0	0
10000	+1	1	1	0	1
20000	+0	1	0	0	1
30000	+0	1	1	0	1
30000	+1	1	1	1	1

En simulation, chaque processus implicite ou explicite communique avec le noyau au moyen des signaux. Celui-ci tient à jour la liste des évènements susceptibles de le réveiller. Pour chaque signal qui fait l'objet d'une affectation (qui se trouve à gauche d'un signe <=), le noyau tient à jour un **pilote** qui indique les formes d'ondes souhaitées. Le signal est mis à jour par le noyau en fonction du ou des pilotes qui lui sont attachés. Un processus ne modifie pas directement la valeur d'un signal, il introduit une nouvelle valeur dans un pilote. L'affectation effective de la valeur se fait au moment de la sortie du processus.

Rien n'empêche que plusieurs pilotes soient attachés à un signal, ce qui peut entrainer un conflit. A l'intérieur d'un processus, le conflit est évité puisque l'ordre d'affectation est séquentiel. C'est la dernière affectation qui impose son état. Mais si plusieurs processus (implicite ou explicite) attachent un pilote à un même signal, il peut y avoir conflit si un pilote tire le signal à 0 et un autre à 1. Prenons un exemple :

```
entity essai is
     port(a, b : in bit;
             c : out bit);
end essai;

architecture beh of essai is
     signal abar : bit;
     signal abarb : bit;
begin

   c <= a;
   c <= b;

end beh;
```

ModelSim refuse de compiler ce design avec le message suivant :

```
# -- Compiling architecture beh of essai
# ** Error: essai.vhd(3): Nonresolved signal 'c' has multiple sources.
#   Drivers:
#     essai.vhd(12):Conditional signal assignment line__12
#     essai.vhd(13):Conditional signal assignment line__13
```

On essaye de câbler deux sorties ensemble, ce qui ne peut pas se faire sans précaution (en mettant un des deux signaux à l'état haute impédance par exemple). Mais avec un type bit qui

ne peut prendre que deux valeurs 0 ou 1, c'est impossible. Il s'agit forcément d'une erreur que ModelSim rejette à la compilation.

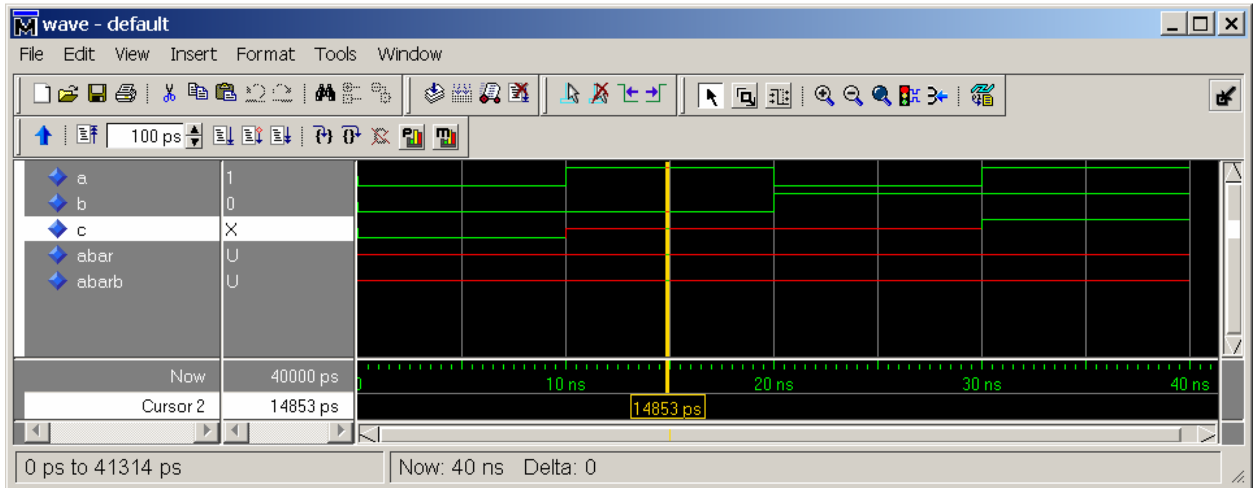
Pourtant, avec un type de données plus évolué comme `std_logic`, c'est possible car ce type à 9 états est accompagné d'une fonction de résolution des conflits. Voici le même exemple avec les signaux de type `std_logic` :

```
library IEEE;
use IEEE.std_logic_1164.all;

entity essai is
     port(a, b : in std_logic;
             c : out std_logic);
end essai;


architecture beh of essai is
     signal abar : std_logic;
     signal abarb : std_logic;
begin

   c <= a;
   c <= b;

end beh;
```

Le testbench doit aussi être modifié de la manière suivante :

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY essai_tb  IS
END;

ARCHITECTURE essai_tb_arch OF essai_tb IS
  SIGNAL a, b, c    :  std_logic ;
  COMPONENT essai
    PORT (c  : out std_logic;
          a  : in std_logic;
          b  : in std_logic );
  END COMPONENT;
BEGIN
  DUT  : essai
    PORT MAP (
      c   => c,
      a   => a,
      b   => b);
END;
```

La compilation s'exécute normalement et on obtient le chronogramme suivant :



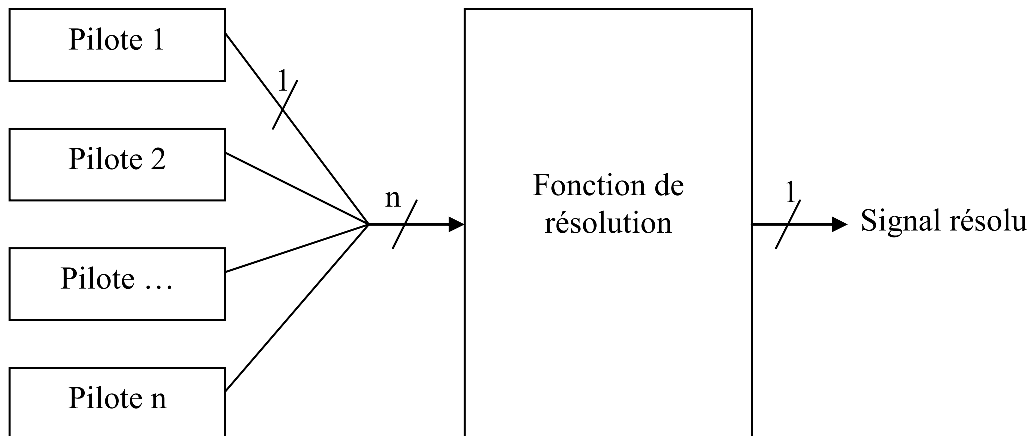
Ainsi que la liste d'évènements suivante :

ps	delta	/essai_tb/dut/a	/essai_tb/dut/b	/essai_tb/dut/c	/essai_tb/dut/abar	/essai_tb/dut/abarb
0	+0	U	U	U	U	U
0	+1	0	0	U	U	U
0	+2	0	0	0	U	U
10000	+0	1	0	0	U	U
10000	+1	1	0	X	U	U
20000	+0	0	1	X	U	U
30000	+0	1	1	X	U	U
30000	+1	1	1	1	U	U

Tous les signaux démarrent maintenant à `U` (Unknown = inconnu), `abar` et `abarb` restant à cette valeur pendant toute la simulation puisqu'ils ne sont plus utilisés. Tant que `a` et `b` ont la même valeur, `c` est égal à `a` et à `b`. Si `a` et `b` ont des valeurs différentes, la sortie passe à l'état `X` (indéterminé). Dans le package std_logic_1164, il y a une fonction de résolution qui utilise la table suivante pour résoudre les conflits entre les pilotes (en gras, la résolution du conflit : tirage à 0 et à 1 simultané) :

```
CONSTANT resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |);
```

C'est la raison pour laquelle ModelSim accepte de compiler le design, même si plusieurs sorties sont reliées ensemble. C'est la fonction de résolution qui va résoudre les conflits durant la simulation.



C'est cette méthode qui permet par exemple de créer des bus dans un design en utilisant des buffers trois états pour gérer les conflits.

6.2 La simulation post-implementation

La simulation post-implementation (ou post-place&route selon la terminologie Xilinx), que l'on appelle aussi post-layout ou encore post-timing, est un autre aspect un peu magique vu en TP. Le modèle physique du composant est créé après placement-routage dans le flot d'implémentation. Ce modèle, appelé modèle VITAL (pour VHDL Initiative Toward ASIC Libraries), est un modèle structurel utilisant les éléments physiques (LUT, bascules D, connexions, …) existant dans le FPGA. Ce modèle n'est jamais créé par le concepteur de circuit intégré numérique, mais il faut connaitre son principe de fonctionnement pour pouvoir analyser les problèmes éventuels. Prenons comme exemple un design très simple :

```
library IEEE;
use IEEE.std_logic_1164.all;

entity essai is
     port(a, b : in std_logic;
             c : out std_logic);
end essai;

architecture beh of essai is
begin
   c <= a and b;
end beh;
```

Ainsi que son testbench :

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY essai_tb  IS
END;

ARCHITECTURE essai_tb_arch OF essai_tb IS
  SIGNAL a, b, c    :  std_logic ;
  COMPONENT essai
    PORT (c  : out std_logic;
          a  : in std_logic;
          b  : in std_logic );
  END COMPONENT;
BEGIN
  DUT  : essai
    PORT MAP (
      c   => c,
      a   => a,
      b   => b);
END;
```

Après synthèse puis implémentation, deux fichiers sont créés dans le flot Xilinx :

- `essai_timesim.vhd` : c'est le modèle VITAL (les noms d'instance sont en gras).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library SIMPRIM;
use SIMPRIM.VCOMPONENTS.ALL;
use SIMPRIM.VPACKAGE.ALL;

entity essai is
  port (
    a : in STD_LOGIC := 'X';
    b : in STD_LOGIC := 'X';
    c : out STD_LOGIC );
end essai;

architecture Structure of essai is
  signal a_IBUF : STD_LOGIC;
  signal b_IBUF : STD_LOGIC;
```

```
  signal GSR : STD_LOGIC;
  signal GTS : STD_LOGIC;
  signal a_INBUF : STD_LOGIC;
  signal b_INBUF : STD_LOGIC;
  signal c_ENABLE : STD_LOGIC;
  signal c_O : STD_LOGIC;
  signal c_OBUF : STD_LOGIC;
  signal VCC : STD_LOGIC;
begin
  a_IBUF_0 : X_BUF_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => a,
      O => a_INBUF
    );
  a_IFF_IMUX : X_BUF_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => a_INBUF,
      O => a_IBUF
    );
  b_IBUF_1 : X_BUF_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => b,
      O => b_INBUF
    );
  b_IFF_IMUX : X_BUF_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => b_INBUF,
      O => b_IBUF
    );
  c_OBUF_2 : X_TRI_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => c_O,
      CTL => c_ENABLE,
      O => c
    );
  c_ENABLEINV : X_INV_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => GTS,
      O => c_ENABLE
    );
  c1 : X_LUT4
    generic map(
      INIT => X"A0A0"
    )
    port map (
      ADR0 => a_IBUF,
```

```
      ADR1 => VCC,
      ADR2 => b_IBUF,
      ADR3 => VCC,
      O => c_OBUF
    );
  c_OUTPUT_OFF_OMUX : X_BUF_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => c_OBUF,
      O => c_O
    );
  NlwBlock_essai_VCC : X_ONE
    port map (
      O => VCC
    );

  NlwBlockROC : X_ROC
    generic map (ROC_WIDTH => 100 ns)
    port map (O => GSR);
  NlwBlockTOC : X_TOC
    port map (O => GTS);

end Structure;
```

- `essai_timesim.sdf` : c'est le fichier qui contient les différents timings du circuit (SDF = Standard Delay Format). Les noms d'instance sont en gras.

```
(DELAYFILE
  (SDFVERSION "3.0")
  (DESIGN "essai")
  (DATE "Thu Oct 26 16:46:47 2006")
  (VENDOR "Xilinx")
  (PROGRAM "Xilinx SDF Writer")
  (VERSION "H.42")
  (DIVIDER /)
  (VOLTAGE 1.14:1.14:1.14)
  (TEMPERATURE 85:85:85)
  (TIMESCALE 1 ps)
  (CELL (CELLTYPE "X_BUF_PP")
    (INSTANCE a_IBUF_0)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 859 ))
        )
      )
  )
  (CELL (CELLTYPE "X_BUF_PP")
    (INSTANCE a_IFF_IMUX)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 1079 ))
        )
      )
  )
  (CELL (CELLTYPE "X_BUF_PP")
    (INSTANCE b_IBUF_1)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 859 ))
        )
```

```
    )
  )

  (CELL (CELLTYPE "X_BUF_PP")
    (INSTANCE b_IFF_IMUX)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 1079 ))
        )
      )
  )
  (CELL (CELLTYPE "X_TRI_PP")
    (INSTANCE c_OBUF_2)
      (DELAY
        (ABSOLUTE
          (PORT I ( 24 ))
          (PORT CTL ( 24 ))
          (IOPATH I O ( 3690 ))
          (IOPATH CTL O ( 3690 ))
        )
      )
  )
  (CELL (CELLTYPE "X_INV_PP")
    (INSTANCE c_ENABLEINV)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 8049 ))
        )
      )
  )
  (CELL (CELLTYPE "X_LUT4")
    (INSTANCE c1)
      (DELAY
        (ABSOLUTE
          (PORT ADR0 ( 518 ))
          (PORT ADR2 ( 297 ))
          (IOPATH ADR0 O ( 596 ))
          (IOPATH ADR1 O ( 596 ))
          (IOPATH ADR2 O ( 596 ))
          (IOPATH ADR3 O ( 596 ))
        )
      )
  )
  (CELL (CELLTYPE "X_BUF_PP")
    (INSTANCE c_OUTPUT_OFF_OMUX)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 982 ))
        )
      )
  )
)
```

On peut faire trois constats en lisant ces fichiers :

- Premier constat, l'entité du modèle VITAL est identique à l'entité du modèle fonctionnel. Cela signifie que, les entrées/sorties étant identiques, le testbench s'appliquera aux deux modèles.

- Deuxième constat, le modèle VITAL est une description structurelle utilisant les éléments du package `SIMPRIM` fourni par Xilinx.

- Troisième constat, à chaque instance d'un composant dans le modèle VITAL est associé un (ou plusieurs) temps dans le fichier SDF. Par exemple, au buffer `a_IBUF_0` placé sur l'entrée `a` est associé un temps de propagation égal à 859 ps :

<table>
<tr>
<td>

```
a_IBUF_0 : X_BUF_PP
    generic map(
      PATHPULSE => 757 ps
    )
    port map (
      I => a,
      O => a_INBUF
    );
```

</td>
<td>

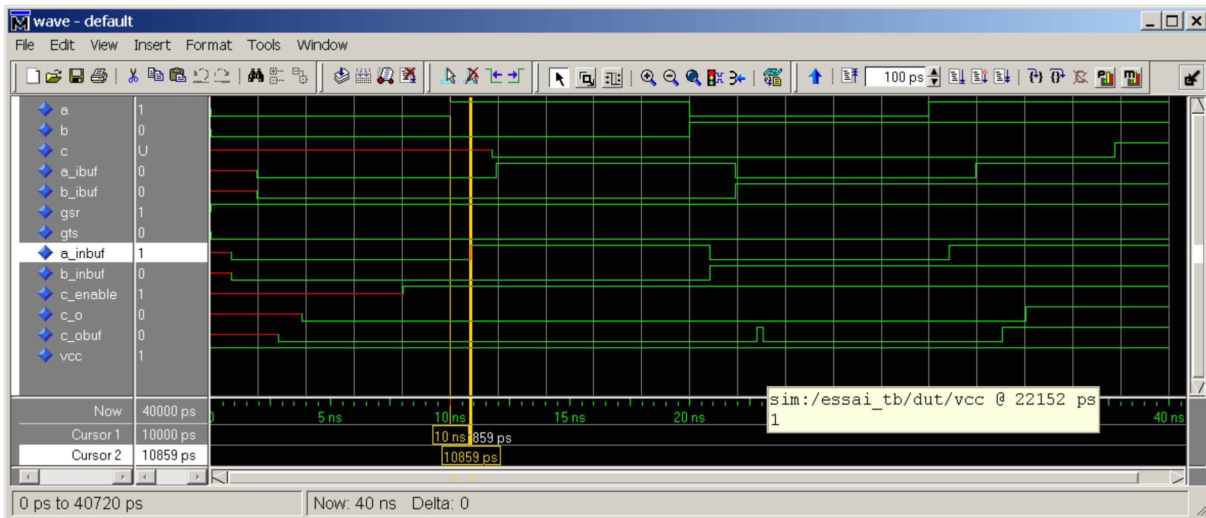
```
(CELL (CELLTYPE "X_BUF_PP")
    (INSTANCE a_IBUF_0)
      (DELAY
        (ABSOLUTE
          (IOPATH I O ( 859 ))
        )
      )
  )
```

</td>
</tr>
</table>

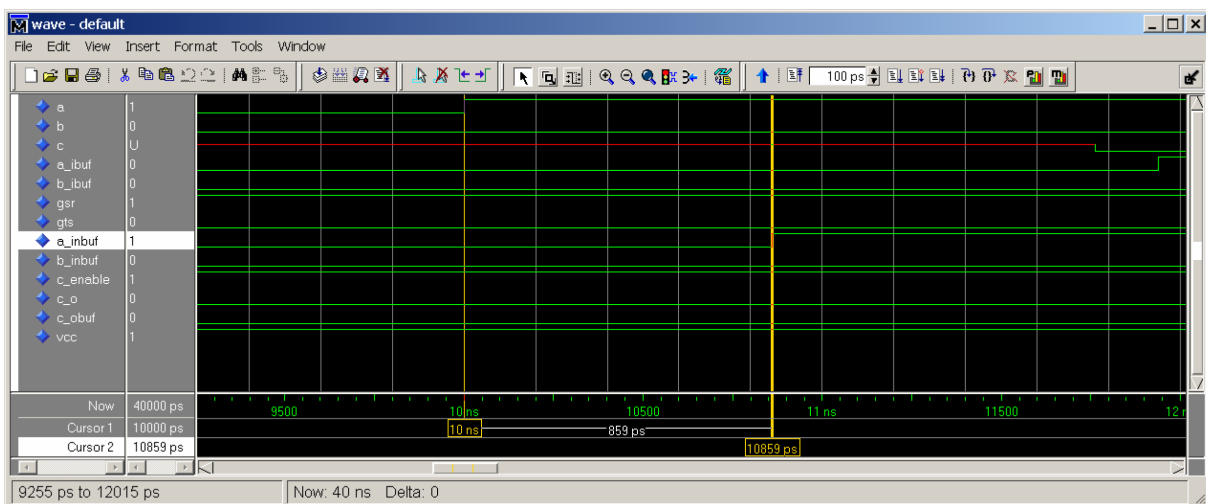
Grace à ModelSim, nous pouvons simuler le modèle VITAL à l'aide des commandes suivantes :

```
1    # création de la librairie work
2    vlib work
3
4    # compilation du modèle VITAL
5    vcom -93 essai_timesim.vhd
6
7    # compilation du testbench
8    vcom -93  essai_tb.vhd
9
10   # lancement du simulateur et chargement du testbench avec
11   # les timings MAX contenus dans le fichier SDF
12   vsim -t 1ps -sdfmax /DUT=essai_timesim.sdf  -lib work essai_tb
13
14   view wave
15   add wave dut/*
16
17   # on applique les stimuli sur les entrées du composant
18   force a 0
19   force b 0
20   run 10 ns
21   force a 1
22   run 10 ns
23   force a 0
24   force b 1
25   run 10 ns
26   force a 1
27   run 10 ns
```

On peut voir dans la fenêtre `wave` l'évolution des signaux internes du modèle VITAL :



Et vérifier qu'il y a bien un retard de 859 ps entre `a` et `a_INBUF` :



Le consortium VITAL a diffusé des librairies de modélisation d'ASICs et de FPGAs qui sont portables et assurent une modélisation suffisamment précise pour offrir une garantie de prévisibilité des résultats (modèle de qualité sign-off). Ces outils de modélisation s'appuient sur une représentation des paramètres dynamiques des circuits compatibles avec celle utilisée dans le monde VERILOG, autre langage de description matériel très utilisé pour concevoir des ASICs (le fichier SDF vient de là). Le standard IEEE 1076.4 ou VITAL95 définit :

* Un package qui contient des procédures de traitement des comportements temporels des circuits, `ieee.vital_timing`.
* Un package qui définit des primitives standard permettant de construire des fonctions combinatoires et séquentielles, `ieee.vital_primitives`.
* Les spécifications des fichiers SDF (Standard Delay Format) qui contiennent les paramètres dynamiques des unités de conception. L'utilisation du fichier SDF n'est d'ailleurs pas obligatoire, le modèle VITAL pouvant définir lui-même les timings. En pratique, tous les outils l'utilisent.

Chez Xilinx, le modèle post-implémentation fait appel à la librairie SIMPRIM qui fait appel elle-même aux librairies VITAL.

L'étude du standard VITAL sort largement du cadre de ce cours. Il faut toutefois savoir que le modèle VITAL ne se contente pas d'indiquer les temps de propagation à l'intérieur du modèle physique du circuit, mais qu'il effectue aussi un certain nombre de vérifications dont voici les principaux :

1. Vérification des temps de setup et de hold,
2. Test de largeur minimale d'impulsion, de période,
3. Détection de glitch.

Si le testbench n'est pas écrit correctement, il est possible que la simulation fonctionnelle marche correctement (puisqu'il n'y a aucune vérification), mais que la simulation post-implémentation détecte par exemple une violation de temps de setup sur les entrées de données. Ce problème peut aussi se poser en cas d'entrée asynchrone par rapport au montage. Il existe dans ce cas des possibilités de demander au simulateur de ne pas tout vérifier.

6.3 Ecriture d'un testbench simple

Dans les deux paragraphes précédents, nous avons généré les entrées directement avec ModelSim et la commande `force`. Nous allons maintenant écrire directement dans le testbench la génération des vecteurs de test. Reprenons le premier exemple du §6.1. La description en VHDL du circuit était la suivante :

```
entity essai is
     port(a, b : in bit;
             c : out bit);
end essai;

architecture beh of essai is
     signal abar : bit;
     signal abarb : bit;
begin
     abarb <= abar and b;
     abar <= not a;
     c <= not abarb;
end beh;
```

Le testbench ne se contente plus d'instancier `essai`. On ajoute, à la suite de l'instanciation, la génération des signaux sur les entrées a et b de la même manière que dans le §6.1 :

```
ENTITY essai_tb  IS
END;

ARCHITECTURE essai_tb_arch OF essai_tb IS
  SIGNAL a, b, c   :  bit;

  COMPONENT essai
    PORT (c  : out bit;
          a  : in bit;
          b  : in bit );
  END COMPONENT;

BEGIN

  DUT  : essai      -- DUT (Device Under Test) est le
    PORT MAP (      -- nom d'instance du composant essai
      c   => c,
      a   => a,
      b   => b);

   gen : process   -- boucle infinie
   begin
       a <= '0';
       b <= '0';
       wait for 10 ns;
       a <= '1';
       wait for 10 ns;
       a <= '0';
       b<= '1';
       wait for 10 ns;
       a <= '1';
       wait;      -- obligatoire sinon on refait un tour
   end process;   -- dans le process

END;
```

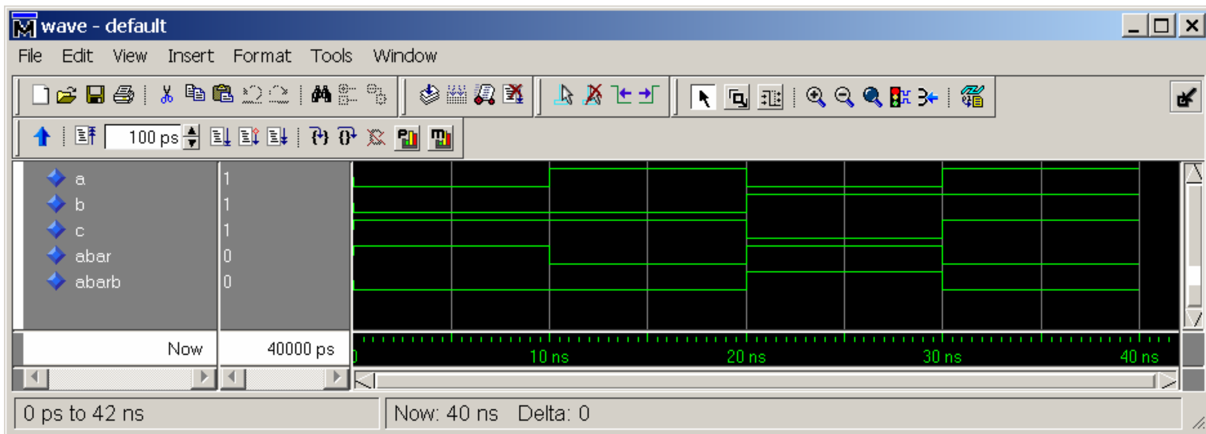
Le process n'a pas de liste de sensibilité, ce qui veut dire qu'il est activé en permanence. L'instruction `wait for XX;` demande au simulateur d'attendre pendant l'expression temporelle `XX` qui est de la forme : `valeur_entière` suivie d'une unité de temps (fs, ps, ns, us, ms, s, min, hr). Le déroulement du process est le suivant :

Temps simulation	process
0 ns	Activation, a=0, b=0, attente 10 ns
10 ns	a=1, attente 10 ns
20 ns	a=0, b=1, attente 10 ns
30 ns	a=1, attente infinie

Les commandes suivantes sont tapées directement dans la fenêtre de commande du simulateur :

```
1    # création de la librairie work
2    vlib work
3
4    # compilation du design
5    vcom -93 essai.vhd
6
7    # compilation du testbench
8    vcom -93 essai_tb.vhd
9
10   # lancement du simulateur et chargement du testbench
11   vsim -t 1ps -lib work essai_tb
12
13   # affichage des fenêtres wave et list
14   view wave list
15
16   # on visualise tous les signaux dans dut
17   add wave dut/*
18   add list dut/*
19
20   # on fait tourner le simulateur pendant 40 ns
21   run 40 ns
```

Après lancement du simulateur, on lui donne l'ordre de tourner pendant 40 ns et on obtient le même résultat qu'au §6.1 dans la fenêtre `wave` :



Ainsi que dans la fenêtre `list` (le cycle 0 a disparu car on est dans le process du testbench) :

ps	delta	/essai_tb/dut/a	/essai_tb/dut/b	/essai_tb/dut/c	/essai_tb/dut/abar	/essai_tb/dut/abarb
0	+0	0	0	0	0	0
0	+1	0	0	1	1	0
10000	+1	1	0	1	1	0
10000	+2	1	0	1	0	0
20000	+1	0	1	1	0	0
20000	+2	0	1	1	1	0
20000	+3	0	1	1	1	1
20000	+4	0	1	0	1	1
30000	+1	1	1	0	1	1
30000	+2	1	1	0	0	1
30000	+3	1	1	0	0	0
30000	+4	1	1	1	0	0

Il est souvent difficile de savoir à l'avance la durée du temps de simulation, notamment lorsqu'elle est longue. On aimerait pouvoir dire à ModelSim de simuler sans limite de temps (commande `run -all`) et arrêter la simulation dans le testbench. C'est possible en ajoutant le process suivant :

```
kill : process
begin
    wait for 40 ns;
    assert (FALSE)
       report "fin de simulation."
       severity Failure;
end process;
```

Il s’agit toujours d’un process sans liste de sensibilité activé dès le lancement de la simulation. L’instruction `assert` permet de tester une condition en VHDL, par exemple la valeur d’un signal. Dans notre cas, on force une condition fausse ce qui permet d’afficher un message à l’aide de l’instruction `report` puis d’indiquer la gravité du problème grâce à l’instruction `severity`. Il y a 4 niveaux de gravité (`severity_level` : `note`, `warning`, `error`, `failure`), mais seul le niveau `failure` arrête la simulation.

Dans la fenêtre ModelSim, on remplace la commande `run 40 ns` par `run -all` et le simulateur s’arrête avec le message suivant :

```
# ** Failure: fin de simulation.
#    Time: 40 ns Iteration: 0 Process: /essai_tb/kill File: essai_tb.vhd
```

Pour continuer à explorer les possibilités d’écriture d’un testbench, changeons d’exemple de design avec un montage séquentiel simple, un registre 8 bits comme celui du §4.9.2 :

```
library IEEE;
use IEEE.std_logic_1164.all;

entity reg is
  port(D : in std_logic_vector(7 downto 0);
   clear : in std_logic;
      ce : in std_logic;
     clk : in std_logic;
       Q : out std_logic_vector(7 downto 0));
end reg;

architecture comporte of reg is
begin

  process(clk, clear) begin
    if (clear ='1') then
      Q <= (others => '0'); -- tous les bits de Q à 0
    elsif (clk 'event and clk ='1') then
      if (ce ='1') then
```

```
        Q <= D;
      end if;
    end if;
  end process;

end comporte ;
```

Il nous faut maintenant décrire dans le testbench une horloge, un signal reset, un chip enable ainsi qu'un bus d'entrée. Voici un exemple de ce qui peut être fait :

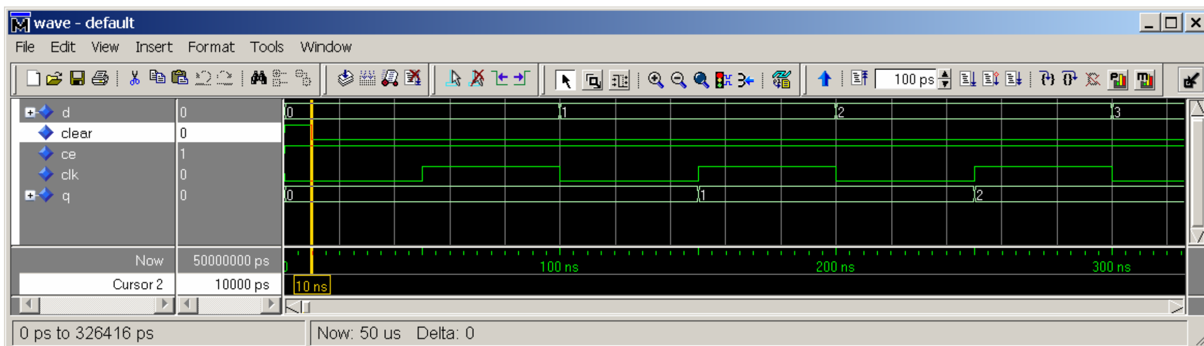
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY reg_tb  IS
END ;

ARCHITECTURE reg_tb_arch OF reg_tb IS
  SIGNAL ce   :  std_logic;
  SIGNAL d   :  std_logic_vector (7 downto 0);
  SIGNAL q   :  std_logic_vector (7 downto 0);
  SIGNAL clk   :  std_logic;
  SIGNAL clear   :  std_logic;

  COMPONENT reg
    PORT (
      ce  : in std_logic;
      d  : in std_logic_vector (7 downto 0);
      q  : out std_logic_vector (7 downto 0);
      clk  : in std_logic;
      clear  : in std_logic );
  END COMPONENT ;

BEGIN

  DUT  : reg
    PORT MAP (
      ce   => ce,
      d   => d,
      q   => q,
      clk   => clk,
      clear   => clear);

   Horloge : process
   begin
       clk <= '0';
       wait for 50 ns;
       clk <= '1';
       wait for 50 ns;
   end process;

   reset_enable : process
   begin
      ce <= '1';
      clear <= '1';
      wait for 10 ns;
      clear <= '0';
      wait;
   end process;
```

```
   data : process(clk, clear)
   begin
      if (clear = '1') then
         d <= (others => '0');
      elsif (falling_edge(clk)) then
         d <= d + 1;
      end if;
   end process;


   kill : process
   begin
    wait for 50000 ns;
    assert (FALSE)
       report "fin de simulation."
       severity Failure;
   end process;

END ;
```

Le process horloge ne pose pas de problème de compréhension. On crée un process activé en permanence qui met `clk` à 0, attend 50 ns puis met `clk` à 1 et attend 50 ns. Puis le process est à nouveau réactivé. Cela créé une boucle infinie de période d'horloge égale à 100 ns. Les process `reset_enable` et `kill` ont déjà été vu précédemment. Quand au process `data`, c'est un simple compteur dont la description a déjà été vue au §4.9.4. Les commandes suivantes lancent la simulation avec ModelSim :

```
1    # création de la librairie work
2    vlib work
3
4    # compilation du design
5    vcom -93 reg.vhd
6
7    # compilation du testbench
8    vcom -93 reg_tb.vhd
9
10   # lancement du simulateur et chargement du testbench
11   vsim -t 1ps -lib work reg_tb
12
13   # affichage des fenêtres wave et list
14   view wave list
15
16   # on visualise tous les signaux dans dut
17   add wave dut/*
18   add list dut/*
19
20   # on fait tourner le simulateur indéfiniment
21   radix -unsigned
22   run -all
```

On obtient comme prévu les signaux suivants. Le bus d'entrée `d` change sur le front descendant de `clk`, et est lu par le registre sur son front montant.



Vous noterez que le testbench n'a nullement vocation à être synthétisé. Les instructions du style `wait for 10 ns;` n'ont aucune équivalence matérielle. Cela n'est pas grave puisque seul le design a vocation à être traduit en portes. Le testbench doit plutôt ressembler à un programme informatique puisqu'il ne sert que pour la simulation. Le secret d'un testbench rapide est en général dans l'écriture informatique plutôt que dans une description de type matériel. **C'est le danger principal quand le designer du circuit écrit lui-même le testbench : il a tendance à rester sur une pensée hardware alors qu'il est souvent plus judicieux de penser software.** Par exemple, les signaux peuvent avec profit être remplacés par des variables ce qui rend la simulation plus rapide. En pratique, cela n'a guère d'importance tant que la complexité du testbench reste raisonnable. Mais quand il devient très complexe, une approche software améliore vraiment les temps de simulation.

On peut améliorer la génération de l'horloge en insérant un temps mort au début de la simulation pour créer par exemple un reset asynchrone sans que l'horloge soit active, ce qui peut être utile en simulation de timing. Il faut pour cela déclarer un signal initialisé à 0 :

```
SIGNAL flag_init : std_logic := '0';
```

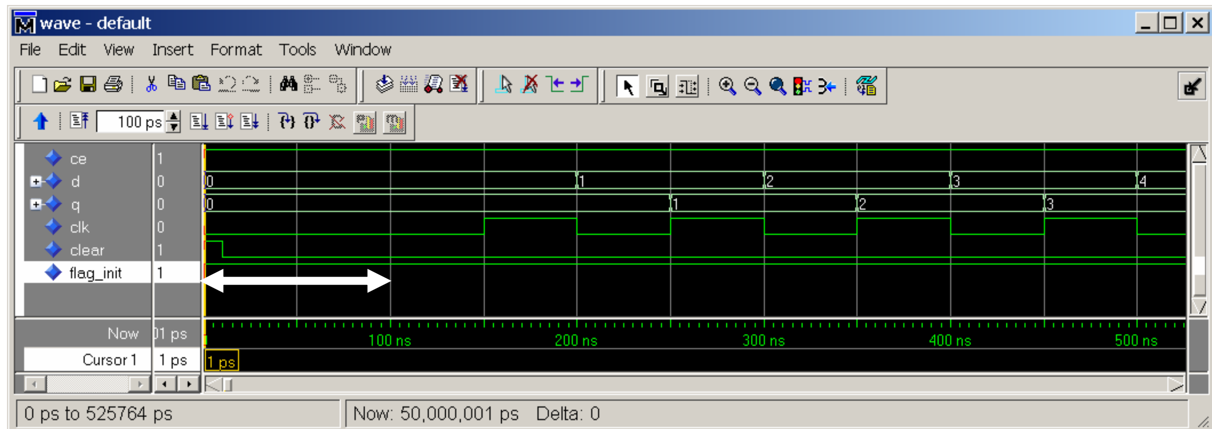
Et le faire passer à 1 un tout petit temps après le démarrage de la simulation (dans le process `kill` par exemple) :

```
kill : process
begin
 wait for 1 ps;
 flag_init <= '1';
 wait for 50000 ns;
 assert (FALSE)
    report "fin de simulation."
    severity Failure;
end process;
```

Il reste à modifier la génération de l'horloge pour insérer le temps mort :

```
Horloge : process
begin
   if (flag_init = '0') then
      clk <= '0';       -- valeur de clk pendant le temps mort
      wait for 100 ns; -- temps mort
   else
      clk <= '0'; -- fonctionnement normal de l'horloge
      wait for 50 ns;
      clk <= '1';
      wait for 50 ns;
   end if;
end process;
```

Dans la fenêtre wave, on voit bien apparaitre un temps mort de 100 ns avant le démarrage de l'horloge :



Il faut comprendre que les 2 process `kill` et `horloge` sont activés dès le début de la simulation puis fonctionnent en parallèle. Quand on rentre pour la première fois dans `horloge` (à t = 0), `flag_init` vaut 0. On va donc rentrer dans le `if`, mettre `clk` à 0 puis attendre 100 ns. Quand on rentre dans `kill` à t = 0, on attend 1 ps, on met `flag_init` à 1, on attend 50 µs, puis on arrête la simulation. Quand on rentre la deuxième fois dans

horloge (à t = 100 ns), flag_init vaut 1 et on entre dans le else. L'horloge démarre et dure jusqu'à la fin de la simulation puisque flag_init reste à 1. Le tableau suivant résume les différents évènements dans le testbench sans représenter les cycles delta:

Temps simulation	reset_enable	horloge	data	kill (flag_init démarre à 0)
0	ce = 1, clear = 1	clk = 0	d = 0	
1 ps				flag_init = 1
10 ns	clear = 0			
100 ns		clk = 0		
150 ns		clk = 1		
200 ns		clk = 0	d = 1	
250 ns		clk = 1		
300 ns		clk = 0	d = 2	
…				
50000.001 ns				Arrêt simulation

Il est aussi possible d'utiliser des fonctions mathématiques de type réel pour créer par exemple un signal sinusoïdal à l'entrée de notre design. Il faut pour cela utiliser le package math_real qui contient le type real ainsi que les fonctions mathématiques. On garde le design reg.vhd précédent et on modifie le tesbench pour obtenir :

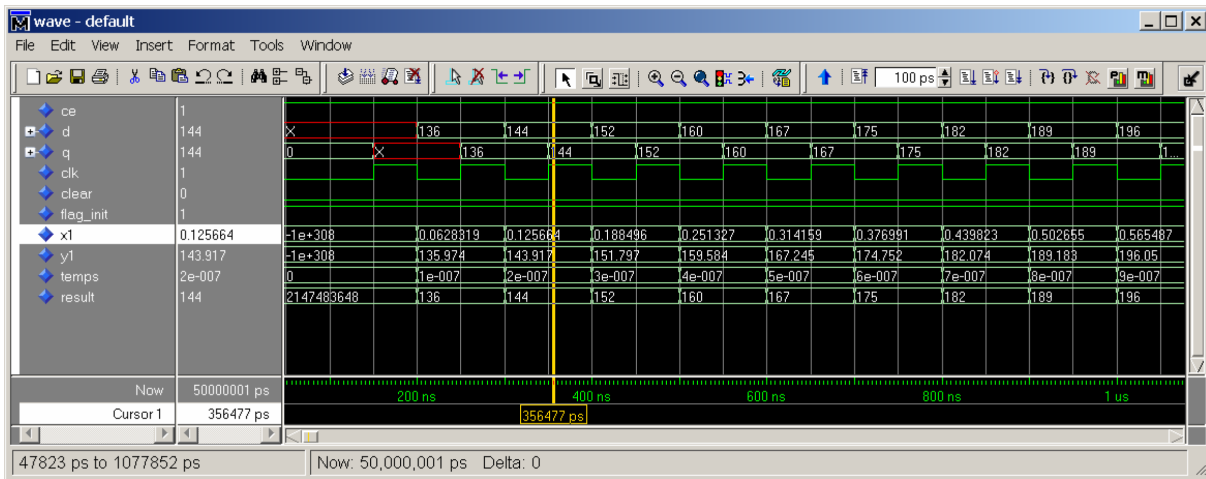
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
use IEEE.math_real.ALL;

ENTITY reg_tb  IS
END ;

ARCHITECTURE reg_tb_arch OF reg_tb IS
  SIGNAL ce   :  std_logic;
  SIGNAL d   :  std_logic_vector (7 downto 0);
  SIGNAL q   :  std_logic_vector (7 downto 0);

      …

    data : process(clk, clear)
      variable X1 : REAL;
      variable Y1 : REAL;
      variable temps : REAL;
      variable result : integer;
```

```
begin
   if (clear = '1') then
      temps := 0.0;
   elsif (falling_edge(clk)) then
      temps := temps + 100.0e-9;
      X1 := 2.0* MATH_PI*(1.0e5)*temps;
      Y1 := (sin(X1)*127.0)+128.0;
      result := integer(Y1);
      d <= conv_std_logic_vector(result,8);
   end if;
end process;

  …

END;
```

Seul le process data a été modifié. Trois variables locales de type `real` et une variable locale de type `integer` ont été créées. La variable temps est initialisée à 0 au démarrage et est incrémentée de 100 ns à chaque période d'horloge. Elle représente le temps `t` qui s'écoule durant la simulation. X1 est égal à $2.\pi.f_0.t$ avec f_0 la fréquence désirée pour le signal sinusoïdal (100 kHz dans notre exemple). Y1 contient la valeur de `sinus`(X1) qui est normalement comprise entre -1 et +1. On la multiplie par 127 et on additionne 128 pour obtenir finalement un sinus compris entre 0 et 255 (dans cet exemple, nous générons un sinus non signé codé sur 8bits). Il faut ensuite convertir Y1 en entier dans la variable `result`, puis convertir `result` en `std_logic_vector` sur 8 bits dans `d`. A l'aide des commandes suivantes :

```
1    # création de la librairie work
2    vlib work
3
4    # compilation du design
5    vcom -93 reg.vhd
6
7    # compilation du testbench
8    vcom -93 reg_tb.vhd
9
10   # lancement du simulateur et chargement du testbench
11   vsim -t 1ps -lib work reg_tb
12
13   # affichage des fenêtres wave et list
14   view wave
15
16   # on visualise tous les signaux dans le testbench
17   add wave *
18
19   # on visualise les locales dans le process data
20   add wave data/*
21
22   # affichage non signé pour tous les signaux et variables entiers
23   radix -unsigned
24
25   # on fait tourner le simulateur indéfiniment
26   run -all
```

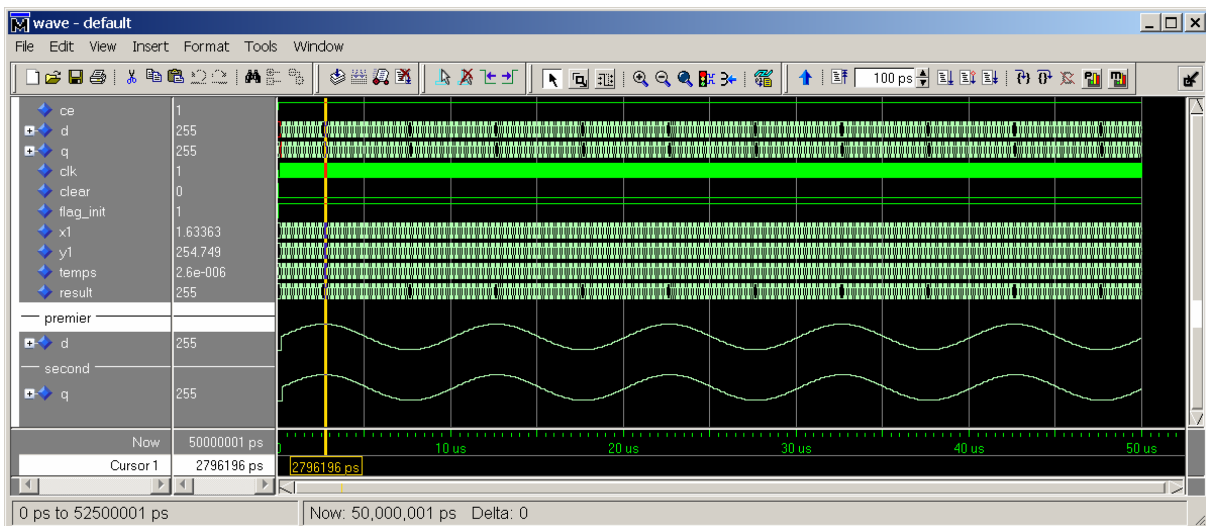
On obtient la fenêtre wave :



Il est possible de faire apparaitre `d` et `q` sous forme analogique en tapant les commandes suivantes :

```
add wave -divider -height 30 {premier}
add wave -format Analog-Step -radix unsigned -scale 0.1 /reg_tb/d
add wave -divider -height 30 {second}
add wave -format Analog-Step -radix unsigned -scale 0.1 /reg_tb/q
```

On obtient alors la fenêtre wave suivante :



6.4 Ecriture d'un testbench utilisant des fichiers

Il est souvent très pratique, notamment dans le domaine du traitement du signal, de créer un fichier contenant les échantillons d'un ou de plusieurs signaux avec un outil tel que Matlab. Le programme M suivant écrit dans un fichier texte 1000 échantillons d'un cosinus 100 kHz échantillonné à 10 MHz (cette fréquence doit être identique à celle de l'horloge dans le testbench) :

```
% effacement des variables précédentes de Matlab
% et fermeture de toutes les fenêtres ouvertes
close all;
clear all;

% 1000 échantillons
N=1000;

% n = 1, 2, 3, ..., 1000
n=1:N;

% fréquence d'échantillonnage = 10 MHz comme dans le testbench
fe=10e+6;

% fréquence du signal à créer
f1=100e+3;

% création du cosinus échantillonné
ss=cos(2*pi*f1/fe*n);

% création et ouverture du fichier in.dat dans le répertoire de
% simulation de ModelSim
f = fopen('in.dat','w');

% l'amplitude du cosinus passe de [-1, +1] à [-127, +127] puis est
% arrondi à l'entier le plus proche
ss = round(ss*127);

% écriture du cosinus dans le fichier in.dat en ASCII
fprintf(f,'%d\r\n', ss);

% fermeture du fichier
fclose(f);
```

On obtient donc le fichier in.dat les valeurs suivantes :

```
127
126
125
123
121
118
115
111
107
103
98
93
87
```

```
81
75
68
61
54
47
…
```

Nous allons garder le design `reg.vhd` précédent et créer un testbench qui va lire les valeurs dans `in.dat`, les appliquer sur l'entrée `d`, puis lire les valeurs obtenues sur la sortie `q` et les écrire avec le même format dans un fichier `out.dat`. Nous allons utiliser pour cela le package `textio` :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE ieee.std_logic_arith.all;
use std.textio.ALL;

ENTITY reg_tb  IS
END ;

ARCHITECTURE reg_tb_arch OF reg_tb IS
  SIGNAL ce   :  std_logic;
  SIGNAL d    :  std_logic_vector (7 downto 0);
  SIGNAL q    :  std_logic_vector (7 downto 0);
  SIGNAL clk   :  std_logic;
  SIGNAL clear   :  std_logic;
  SIGNAL flag_init : std_logic := '0';
  COMPONENT reg
    PORT (
      ce   : in std_logic;
      d   : in std_logic_vector (7 downto 0);
      q   : out std_logic_vector (7 downto 0);
      clk  : in std_logic;
      clear  : in std_logic );
  END COMPONENT ;

  function to_line (arg : integer) return line is
      variable result : string (1 to 12) := (others => ' ');
      variable v : integer := arg;
      constant negative : boolean := arg < 0;
      variable count : integer := result'right + 1;
      variable l : line;
  begin
      if negative then
            v := -v;
      end if;
      loop
            count := count - 1;
            result(count) := character'val(character'pos('0') + (v rem 10));
            v := v/10;
            exit when v = 0;
      end loop;
      if negative then
            count := count - 1;
            result(count) := '-';
      end if;
      write (l, result(count to result'right));
      return l;
  end;
```

```
  procedure writeSTDLVsigned(l : inout line; value : in std_logic_vector) is
      variable x : integer := 0;
      variable value_neg : std_logic_vector(value'range);
  begin
      if value(value'high) = '0' then
            for i in value'range loop
                  x := x + conv_integer(value(i))*(2**i);
            end loop;
            l := to_line(x);
      else
            value_neg := not value + '1';
            for i in value'range loop
                  x := x + conv_integer(value_neg(i))*(2**i);
            end loop;
            l := to_line(-x);
      end if;
   end;

  procedure writeSTDLVunsigned(l : inout line; value : in std_logic_vector) is
      variable x : integer := 0;
      variable value_neg : std_logic_vector(value'range);
  begin
      for i in value'range loop
            x := x + conv_integer(value(i))*(2**i);
      end loop;
      l := to_line(x);
  end;


  file InFile : TEXT open read_mode is "in.dat";
  file OutFile : TEXT open write_mode is "out.dat";

BEGIN

  DUT  : reg
    PORT MAP (
      ce   => ce,
      d    => d,
      q    => q,
      clk   => clk,
      clear   => clear);

  Horloge : process
  begin
      if (flag_init = '0') then
         clk <= '0';
         wait for 100 ns;
      else
         clk <= '0';
         wait for 50 ns;
         clk <= '1';
         wait for 50 ns;
      end if;
  end process;

  reset_enable : process
  begin
      ce <= '1';
      clear <= '1';
      wait for 10 ns;
      clear <= '0';
      wait;
  end process;
```

```
data_in : process(clk)
    variable ligne : line;
    variable result : integer;
begin
    if (falling_edge(clk)) then
       readline(InFile, ligne);
       read(ligne, result);
       d <= conv_std_logic_vector(result,8);
    end if;
end process;

data_out : process(clk)
    variable ligne : line;
begin
    if falling_edge(clk) then
       writeSTDLVsigned(ligne, q);
       writeline(OutFile, ligne);
    end if;
end process;

kill : process
begin
    wait for 1 ps;
    flag_init <= '1';
    wait for 50000 ns;
    assert (FALSE)
       report "fin de simulation."
       severity Failure;
end process;

END ;
```

En lecture, la procédure est assez simple. Le package `textio` fournit des types et des procédures qui facilitent les manipulations sur les fichiers en mode texte. Assez simples à mettre en œuvre, les opérations sur les fichiers sont limitées au strict nécessaire comme l'accès séquentiel aux données qui permet de lire des stimuli dans un fichier, d'initialiser une mémoire ou encore d'enregistrer des résultats de simulation. Les lignes suivantes permettent d'ouvrir les fichiers en lecture et en écriture. Les objets `Infile` et `OutFile` de type `file` permettront de manipuler les 2 fichiers `in.dat` et `out.dat` :

```
file InFile : TEXT open read_mode is "in.dat";
file OutFile : TEXT open write_mode is "out.dat";
```

Nous allons maintenant étudier les deux modes : lecture et écriture.

1. Lecture : c'est le fonctionnement le plus simple. Il faut lire une ligne en ASCII, la convertir en entier puis en `std_logic_vector`. La procédure `readline()` lit, ligne par ligne, du fichier représenté par `Infile` vers un buffer ASCII de type `line` alloué

dynamiquement en mémoire. La procédure `read()` convertit automatiquement le buffer `ligne` vers `result` pour les principaux type de base du langage (ici, `result` est un entier). Il suffit ensuite de convertir l'entier `result` en `std_logic_vector` sur 8 bits :

```
data_in : process(clk)
    variable ligne : line;
    variable result : integer;
begin
    if (falling_edge(clk)) then
       readline(InFile, ligne);
       read(ligne, result);
       d <= conv_std_logic_vector(result,8);
    end if;
end process;
```

2. Ecriture : c'est plus complexe car il faut convertir un `std_logic_vector` en buffer de type `line`, ce qui n'existe pas dans le package `textio`. La procédure `writeSTDLVsigned()` convertit en entier un `std_logic_vector` interprété comme signé en CA2, puis grâce à la procédure `to_line()`, convertit cet entier en un buffer ASCII de type `line` alloué dynamiquement. La procédure `writeline()` permet d'écrire ligne à ligne le buffer `ligne` dans le fichier représenté par `OutFile`.

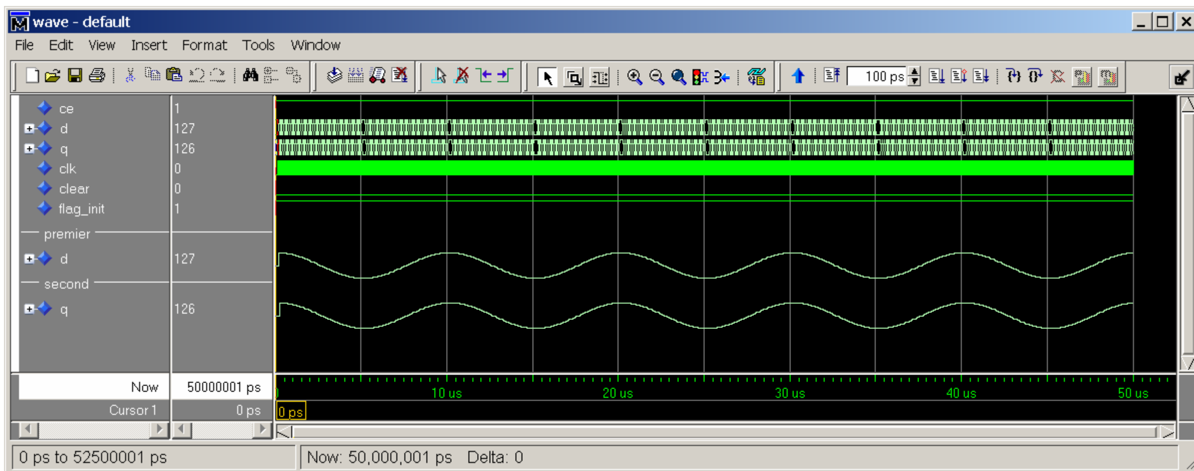
```
data_out : process(clk)
    variable ligne : line;
begin
    if falling_edge(clk) then
       writeSTDLVsigned(ligne, q);
       writeline(OutFile, ligne);
    end if;
end process;
```

Vous noterez que, comme le registre lit et écrit ses données sur le front montant de l'horloge, le testbench génère les stimuli et lit les résultats sur le front descendant de l'horloge. Cela garantit un bon fonctionnement y compris en simulation post-implementation. A l'aide des commandes suivantes :

```
# création de la librairie work
vlib work

# compilation du design
vcom -93 reg.vhd

# compilation du testbench
vcom -93 reg_tb.vhd
```

```
# lancement du simulateur et chargement du testbench
vsim -t 1ps -lib work reg_tb

# affichage des fenêtres wave et list
view wave

# on visualise tous les signaux dans le testbench
add wave *
add wave -divider -height 30 {premier}
add wave -format Analog-Step -radix signed -scale 0.1 /reg_tb/d
add wave -divider -height 30 {second}
add wave -format Analog-Step -radix signed -scale 0.1 /reg_tb/q

# affichage signé pour tous les signaux et variables entiers
radix -signed

# on fait tourner le simulateur indéfiniment
run -all
```

On obtient la fenêtre `wave` suivante :

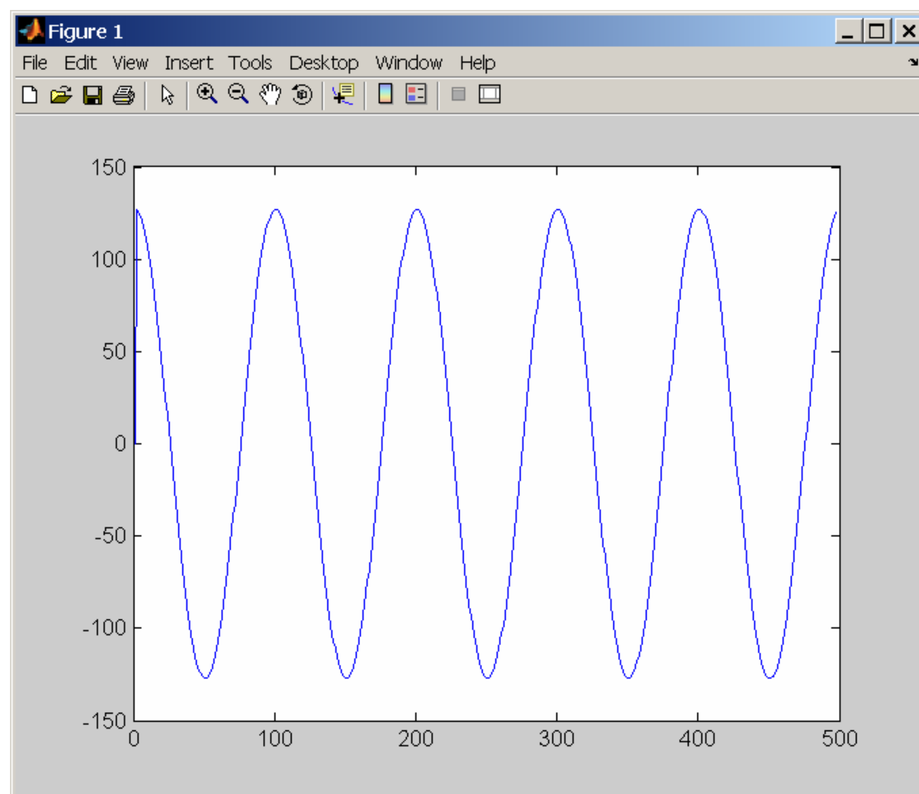


Le fichier `out.dat` contient les valeurs :

```
0
127
126
125
123
121
118
115
111
107
103
98
93
87
81
75
68
61
54
…
```

A l'aide du programme M suivant, on peut lire le contenu du fichier `out.dat` puis tracer sa représentation :

```
% effacement des variables précédentes de Matlab
% et fermeture de toutes les fenêtres ouvertes
clear all
close all

% ouverture du fichier out.dat dans le répertoire de
% simulation de ModelSim
fid=fopen('out.dat','rt');

% Lecture en ASCII du fichier
I=fscanf(fid,'%f');

% affichage du signal obtenu
plot(I)
```



Bien des choses plus complexes peuvent être réalisées avec les fichiers en VHDL. Le livre de J.Weber et M.Meaudre (cf §1.5.8) vous donnera des exemples plus détaillés au paragraphe 2.7.5 (outils de simulation).

7 Conception et performances

7.1 Introduction

Concevoir un circuit qui fonctionne correctement n'est que la première étape du design. Il faut aussi qu'il soit performant. C'est généralement là que l'on voit la différence entre un mauvais et un bon concepteur (avec la fiabilité du design). Trois caractéristiques principales déterminent la performance d'un design :

1. La surface de silicium utilisée, c'est-à-dire le nombre de portes logiques utilisées. La surface et le nombre d'entrées/sorties déterminent directement le prix du circuit (ainsi que le type de boitier).

2. La vitesse, c'est-à-dire la fréquence maximale de fonctionnement du circuit. Le fabricant de FPGA commercialise des versions plus ou moins rapides d'un circuit (option speed grade). A la fin de la fabrication, les circuits sont triés en trois ou quatre catégories en fonction de leur vitesse. Bien sur, plus le circuit est rapide et plus il sera cher. Un design plus rapide qu'un autre permettra donc d'utiliser un circuit plus lent et donc moins cher.

3. La consommation qui détermine la puissance dissipée. Le critère de consommation est très important pour les applications fonctionnant sur batterie. Le critère de puissance dissipée est important pour les applications embarquées. Même sans contrainte particulière, il faut veiller aux problèmes de refroidissement par l'ajout éventuel d'un radiateur sous peine de destruction du circuit. La détermination de la consommation d'un circuit CMOS n'est pas évidente. Elle peut être séparée en deux termes, la puissance statique et la puissance dynamique.

   P dissipée = P statique (indépendante de f) + P dynamique (proportionnelle à f)

   La consommation statique est déterminée par les courants de fuite des transistors (leakage current) alors que la consommation dynamique est déterminée par les changements d'états qui se traduisent par des charges et des décharges de condensateur.

En dessous du nœud 0.13 µm, la consommation statique et dynamique et la puissance dissipée deviennent prépondérantes dans la conception des circuits en technologie CMOS. Ce sont elles qui déterminent les performances, notamment dans le domaine des microprocesseurs.

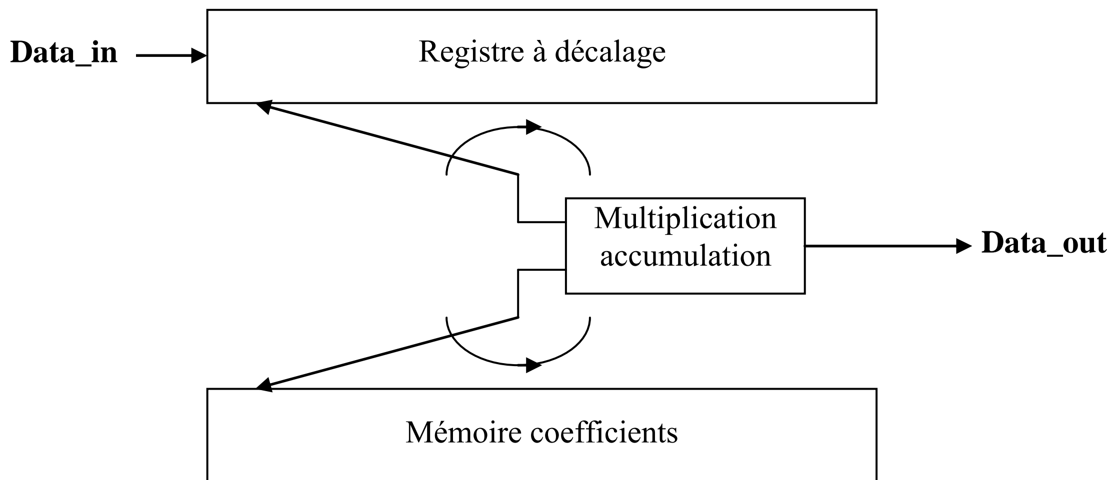
7.2 Compromis surface/vitesse

Ces deux critères agissent malheureusement en sens inverse. L'augmentation de la fréquence de fonctionnement conduit généralement à un design contenant plus de portes logiques et donc plus coûteux. Lorsque l'on veut implanter un algorithme, la conception d'un circuit intégré offre un degré de liberté supplémentaire par rapport à l'utilisation d'un microprocesseur (ou d'un DSP dans le cas d'un algorithme de traitement du signal) ; la spatialisation du traitement. Voyons le fonctionnement de l'algorithme dans les deux cas :

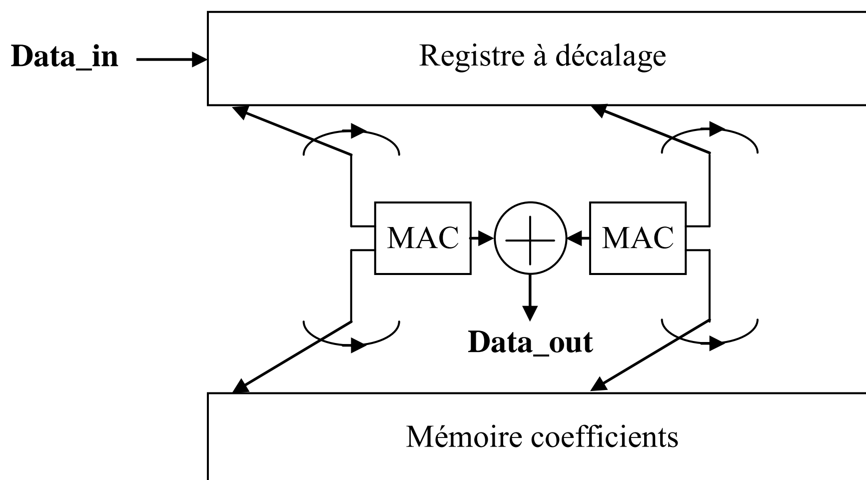
- Avec un processeur, il y a un nombre limité de périodes d'horloge pour réaliser un traitement. Par exemple, avec un DSP fonctionnant à 500 MHz, il y a 500 périodes d'horloge pour traiter chaque échantillon d'un signal 1 Ms/s. C'est le rapport entre la fréquence du processeur et la fréquence d'échantillonnage qui détermine la complexité du traitement réalisable. Le même type de raisonnement peut être tenu avec un traitement par bloc d'échantillons. Si l'algorithme devient trop complexe, le traitement est impossible sauf en changeant de processeur pour un modèle plus rapide ou comportant plusieurs unités travaillant en parallèle.

- En conception de circuit, la même limitation existe concernant la rapidité d'une unité de traitement (un MAC par exemple), mais on a la possibilité de spatialiser l'algorithme, c'est-à-dire de rajouter des unités de traitement supplémentaires qui travailleront en parallèle. On augmente la surface utilisée, mais, à fréquence de fonctionnement identique, on augmente la capacité de calcul.

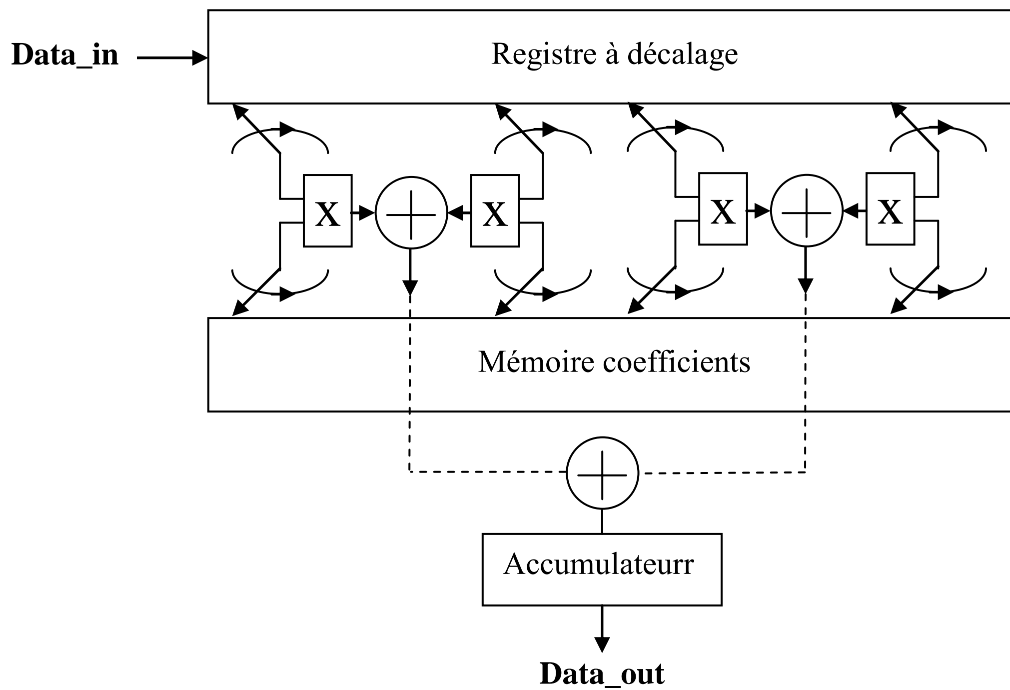
Prenons l'exemple d'un FIR pour illustrer cette différence. Dans TP4, nous avons réalisé un FIR MAC qui illustre bien le fonctionnement d'un DSP. D'un point de vue symbolique, le FIR est composé d'un registre à décalage (réalisé avec un tableau dans le cas d'un DSP), d'une mémoire contenant les coefficients du filtre et d'un MAC :



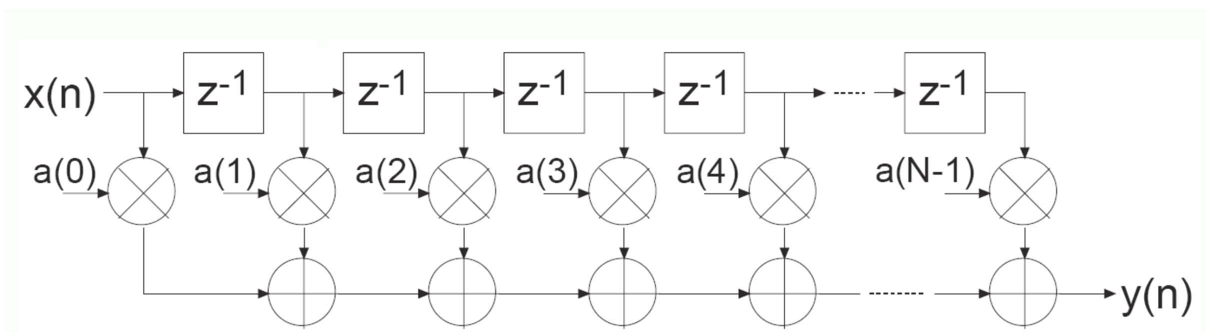
Pour un filtre à N coefficients, il faut N périodes d'horloge pour calculer un échantillon du signal filtré. Il doit donc y avoir un rapport N entre l'horloge du circuit et l'horloge d'échantillonnage. Pour un DSP, cela limite le nombre de coefficients du filtre. Dans le cas d'un circuit intégré, il est toujours possible d'utiliser deux MAC pour faire le calcul :



Chaque MAC calcule la moitié du produit de convolution et il suffit d'additionner les deux résultats pour obtenir l'échantillon de sortie. Le calcul ne nécessite plus que N/2 périodes d'horloge. On peut transformer ce montage afin de n'utiliser qu'un seul accumulateur et obtenir la structure suivante :



On peut augmenter de cette manière le nombre de multiplieurs jusqu'à avoir un multiplieur par coefficient. Le MAC en sortie disparait alors puisqu'il n'y a plus rien à accumuler. Il ne faut plus qu'une seule période d'horloge pour calculer l'échantillon filtré de sortie. C'est la structure parallèle traditionnelle, la plus rapide mais aussi la plus consommatrice de surface (et celle aussi qui consomme le plus) :



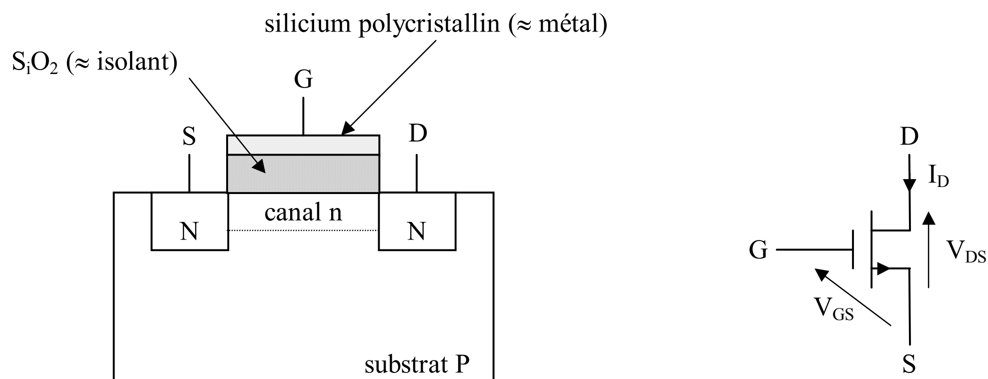
Vous voyez là le degré de liberté supplémentaire apporté par la conception matériel. L'algorithme ne se déroule plus seulement de manière temporelle, mais aussi de manière spatiale. Bien sur, plus l'algorithme est spatialisé, plus il consomme de surface. Mais en changeant l'architecture matérielle, sa fréquence de fonctionnement, ici la fréquence d'échantillonnage, peut tendre vers la fréquence système. Nous pouvons en quelque sorte régler le nombre de périodes d'horloge système consommé par le traitement (dans le cas d'un

FIR à N coefficients, de 1 à N périodes par échantillon) en spatialisant plus ou moins l'algorithme, la fréquence système restant constante. Nous verrons au §7.4.3 que la fréquence système peut aussi être augmentée en utilisant la méthode du pipeline.

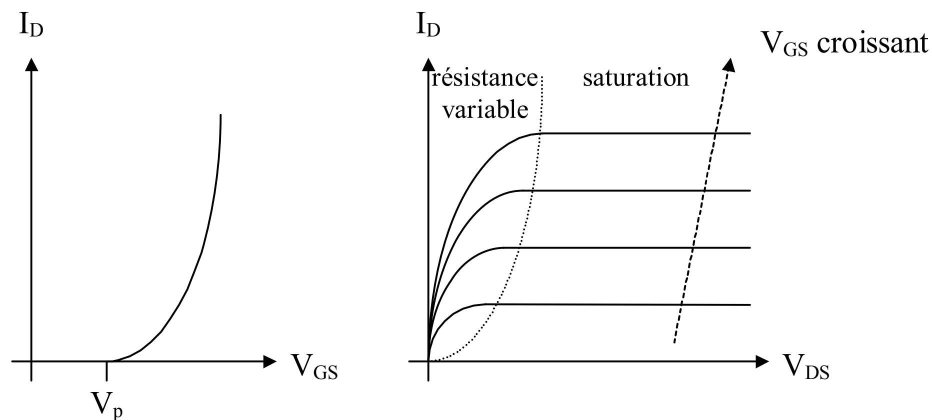
7.3 Problèmes liés à la consommation

7.3.1 Structure d'un circuit CMOS

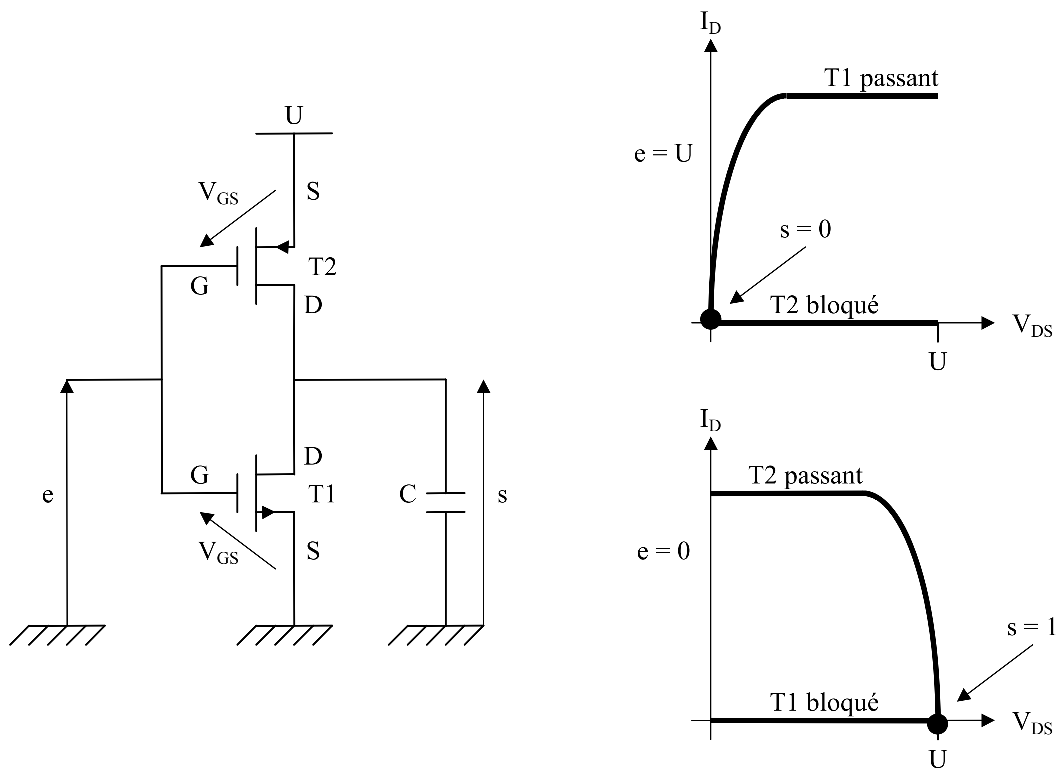
La structure d'un transistor MOS (Metal Oxyde Semiconductor) est rappelée ici. Nous ne travaillerons dans ce chapitre qu'avec des MOS à enrichissement (canal induit par V_{GS}).



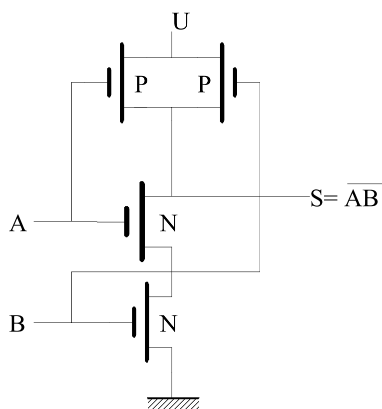
Ses caractéristiques électriques sont :



La logique CMOS (Complementary MOS) associe toujours des paires de transistors MOS, un canal N et un canal P placés en série entre l'alimentation V_{CC} et la masse GND. Quand le transistor canal N est passant, le canal P est ouvert et vice-versa. L'avantage est que si les transistors sont des interrupteurs parfaits, il n'y a jamais de courant qui circule dans la porte entre V_{CC} et GND. Voici le schéma d'un inverseur CMOS :



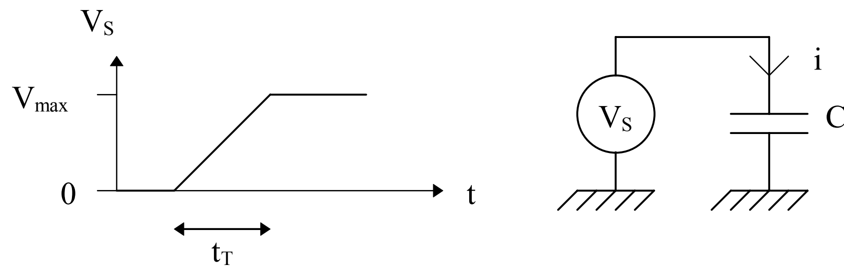
Tous les circuits logiques sont basés sur ce principe. Voici un exemple de fonction logique NAND à base de transistors CMOS.



7.3.2 Consommation dynamique

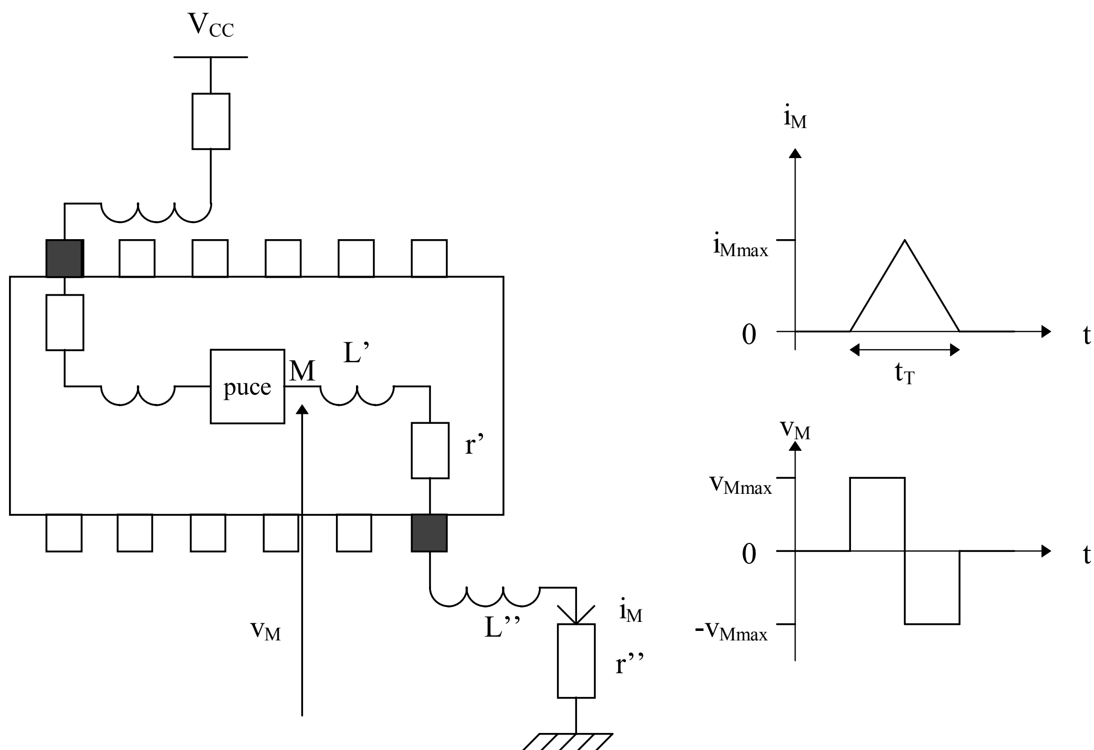
On peut considérer que l'entrée d'une porte CMOS est une capacité pure de l'ordre de quelques pF. La sortie d'une porte ne voit donc qu'une capacité C représentant les capacités d'entrées C_{GS} des entrées connectées plus la capacité de la ligne. Nous allons maintenant étudier quelle est la quantité de courant nécessaire à la charge et à la décharge de cette

capacité. Nous allons supposer, dans un calcul simplifié, que la charge se fait à courant constant avec une tension de la forme :

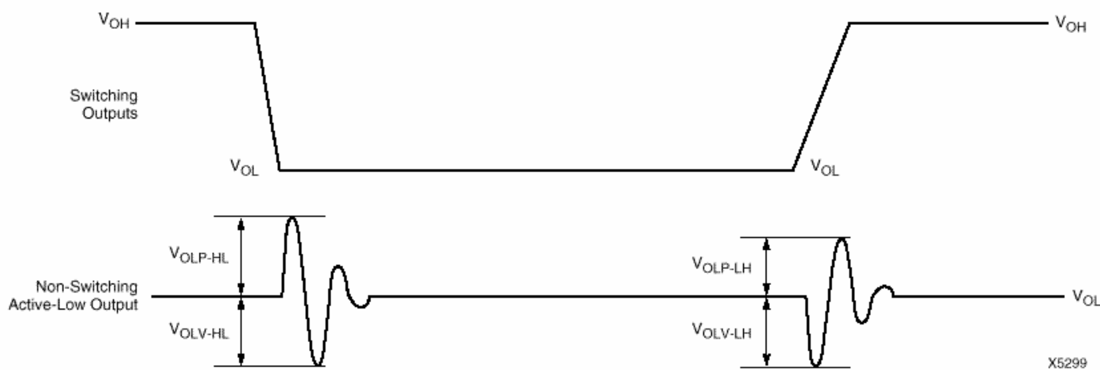


On a $i = C.\frac{dV_S}{dt}$, ce qui implique qu'au moment de la commutation, $i = C.\frac{V_{max}}{t_T}$ en supposant i constant durant la commutation. Par exemple, si C = 5 pF, V_{max} = 5V et t_T = 1 ns, on obtient i = 25 mA.

Lorsque plusieurs signaux commutent simultanément (SSO : Simultaneously Switching Outputs, cas d'un bus de données ou d'un bus d'adresses par exemple), il se produit un pic de courant élevé qui doit être fourni par l'alimentation du circuit. Or les liaisons vers V_{CC} et vers la masse d'un circuit présentent des résistances et des inductances parasites :



La masse vue par le circuit est le point M et on a $v_M = r.i_M + L.\frac{di_M}{dt}$ par rapport à la masse générale de la carte (avec L = L' + L'' et r = r' + r''). Si on reprend les valeurs de l'exemple précédent avec 10 sorties qui commutent simultanément et r = 0,01 Ω, L = 5 nH, i_{Mmax} = 250 mA et t_T = 1 ns, on a v_{Mmax} = 2,5 V ce qui est largement suffisant pour faire basculer une porte qui devrait rester à un niveau constant 0. Le phénomène réel provoque sur la masse une oscillation (Ground bounce) qui a la forme suivante :



Afin de réduire ce phénomène, il faut respecter les règles suivantes en logique rapide :

- utiliser un circuit imprimé avec un plan de masse et un plan d'alimentation connectés directement aux bornes des circuits intégrés afin de réduire les résistances et inductances parasites. Dans la mesure du possible, laisser le plan de masse intact car une faible modification (rangée de trous, fente) peut avoir un effet désastreux.

- Découpler tous les circuits intégrés avec un condensateur placé au plus près du boîtier sur chaque paire Vcc–GND (les alimentations représentent environ 25 % des broches du boîtier). Les condensateurs de découplage servent de réservoir de courant et fournissent au circuit le pic de courant nécessaire à la commutation. La valeur du condensateur n'a pas à être élevée (10 nF suffit) mais il ne faut pas utiliser de condensateurs chimiques à cause de leur mauvais comportement en hautes fréquences ni de condensateurs ayant une inductance parasite trop élevée.

- Le choix du type de boîtier a son importance. En termes d'inductance parasite, les boîtiers PGA et DIP sont les plus mauvais, les boîtiers QFP et BGA sont les meilleurs et les boîtiers PLCC se trouvent entre les deux.

La consommation dynamique d'un circuit intégré CMOS est donc impulsionnelle et peut varier très fortement en quelques μs. Comment peut-on la mesurer ? La méthode la plus simple mais qui ne donne qu'une valeur estimée (les pics de courant étant fournis par les condensateurs de découplage) est la suivante :

1. Placez une résistance de 1 Ω en série avec l'alimentation du circuit.
2. A l'aide de 2 sondes, effectuez une mesure différentielle de la tension aux bornes de la résistance.
3. Vous visualiserez sur l'oscilloscope l'évolution du courant au cours du temps.

Il reste encore une question à traiter : quels sont les signaux qui déterminent la consommation dynamique d'un circuit CMOS ? Il n'y en a que deux : l'horloge et les bus. Les autres signaux ont une faible influence sur la consommation totale du circuit.

- L'horloge. C'est un signal à sortance très élevée puisqu'il faut la distribuer sur toutes les bascules du circuit en même temps (problème du skew). Or un signal à forte sortance consomme toujours beaucoup de courant (les charges sont capacitives) à cause des nombreux amplificateurs utilisés.
- Les bus. Ce sont les commutations de bus qui créent les impulsions de courant dynamiques. Plus le bus sera large et le débit sera élevé, plus les appels de courant seront importants.

Les outils de CAO cherchent depuis longtemps à estimer la consommation dynamique d'un circuit au moment de la conception, ce qui est loin d'être évident. Hier, on cherchait à déterminer le nombre de bascules sur la ligne d'horloge, la fréquence de fonctionnement, le nombre de bus et leur largeur ainsi que le nombre moyen de commutations par seconde. Puis, à l'aide d'une feuille Excel, on estimait la consommation moyenne. Il existe aujourd'hui des outils d'analyse de consommation tel que Xpower chez Xilinx qui peut être couplé avec le simulateur ModelSim pour obtenir une évaluation plus précise. Il existe toujours un outil d'évaluation rapide sur le site web de Xilinx (Power Central), qui est le successeur de la feuille Excel des débuts.

7.3.3 Consommation statique

En théorie, la consommation statique d'un circuit logique CMOS est nulle. En réalité, des courants de fuite (leakage current) existent, mais ils sont restés très longtemps négligeables.

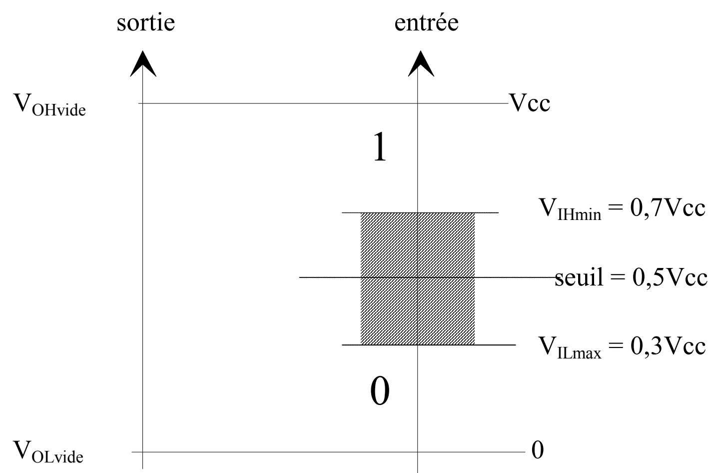
Depuis la technologie 0.13 µm, ce n'est plus le cas. Le problème est du à la réduction de l'épaisseur de la couche d'oxyde de silicium qui isole la grille du canal. Pour que l'isolant soit de bonne qualité, il faut quelques dizaines de couches atomiques, ce qui n'est plus le cas aujourd'hui. Deux types de courants de fuites existent :

1. Le courant grille source. L'entrée n'est plus purement capacitive, une résistance non infinie apparait entre la grille et la source.
2. Le courant drain source. Lorsque le transistor est ouvert, un courant subsiste entre drain et source.

Cette contrainte nouvelle met une forte pression sur le processus de fabrication. Elle deviendra de plus en plus prépondérante avec la diminution de la longueur du canal, et donc la diminution de l'épaisseur d'oxyde sur la grille. Les outils de calcul de la consommation vus au paragraphe précédent estiment aussi la consommation statique du circuit.

7.3.4 Caractéristiques électriques

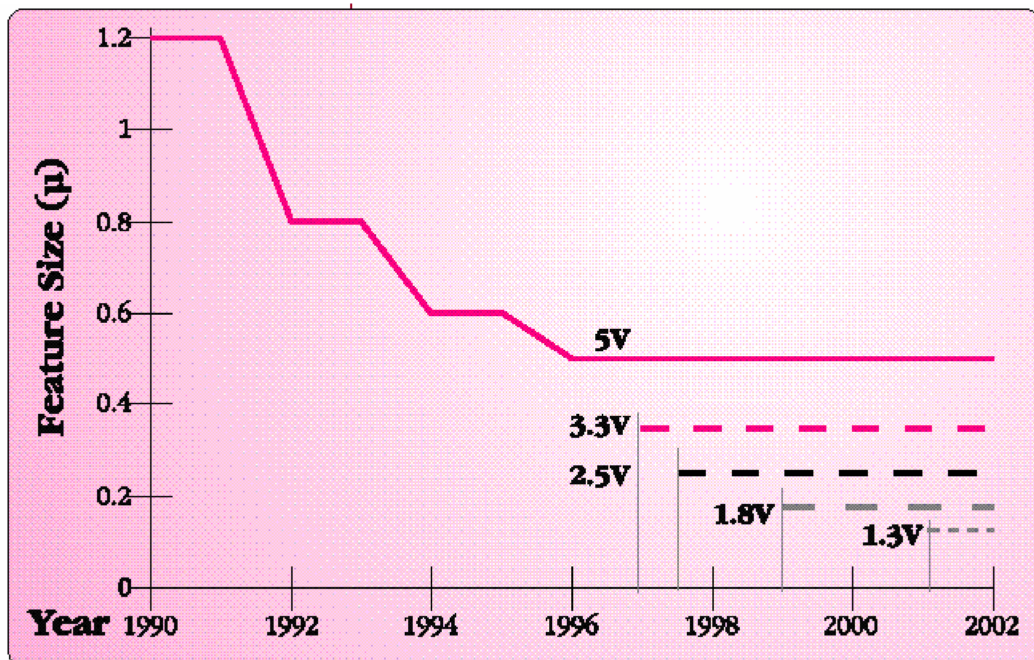
Tant que les circuits CMOS restent chargés par d'autres circuits CMOS, les niveaux hauts et bas restent à V_{CC} et 0 (car les courants d'entrées sont très faibles). Les immunités aux bruits sont élevées et proportionnelles à VCC : $\Delta H = \Delta L = 0{,}3.Vcc$.



La sortance (**fanout** en anglais) en CMOS est déterminée par le temps de propagation (via le temps de transition) maximum du circuit et non par le rapport entre le courant de sortie et le courant d'entrée comme en TTL. En effet, le temps de propagation du circuit est fixé en pratique par la capacité de charge et donc par le nombre d'entrées connectées sur la sortie, c'est-à-dire la sortance. Ceci est bien entendu vrai à l'intérieur comme à l'extérieur du circuit.

7.3.5 Tension d'alimentation et puissance

Trois facteurs sont essentiels en technologie CMOS : la longueur du canal du transistor MOS, la tension d'alimentation du circuit et sa puissance dissipée maximale. La longueur du canal est passée de 10 µm dans les années 70 à 0.5 µm vers 1995. Jusqu'à cette date, la tension d'alimentation est restée fixée à 5 V en règle générale (sauf pour des séries spéciales basse consommation). En dessous de 0.5 µm, un transistor MOS rapide est incapable de supporter 5 V : il est donc obligatoire de baisser la tension d'alimentation (même sans tenir compte des problèmes de puissances dissipées). La figure suivante indique l'évolution de la tension d'alimentation en fonction de la longueur du canal :

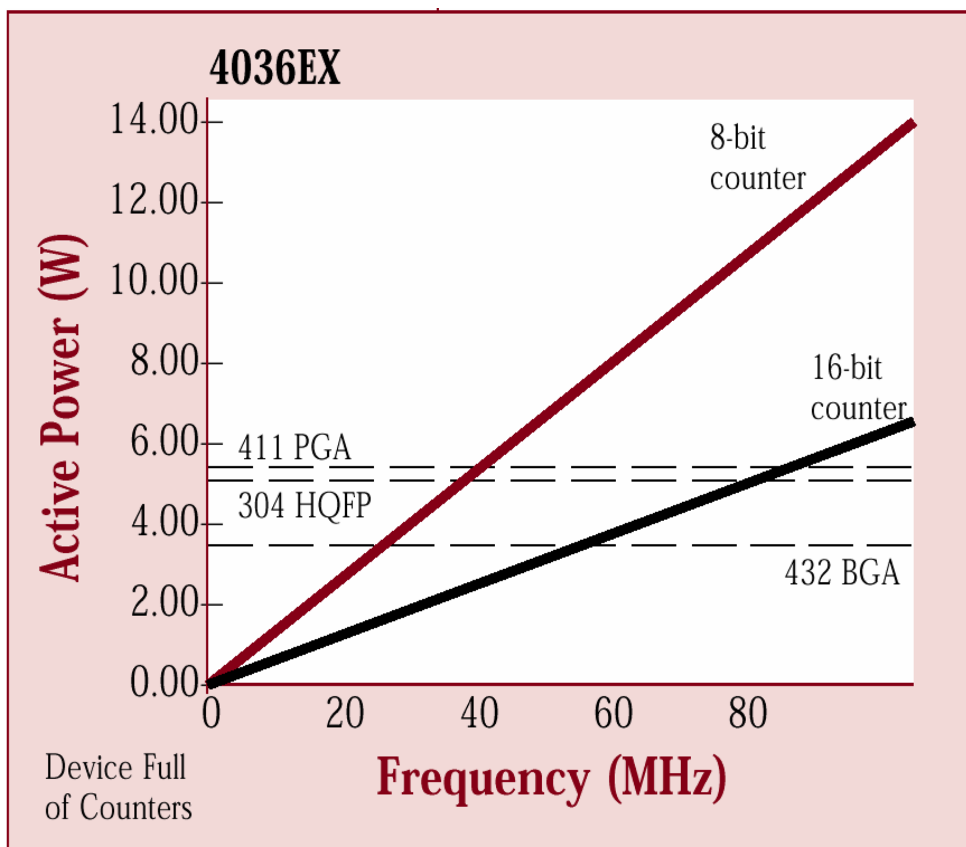


Cette chute de la tension d'alimentation a pour conséquence une réduction notable de la consommation du circuit (qui est proportionnelle à Vcc^2), ce qui est plutôt une bonne chose comme nous allons le voir. Elle a aussi pour conséquence une baisse des performances car la fréquence de fonctionnement d'un circuit CMOS est proportionnelle à Vcc. Malgré cela, la diminution de la longueur de canal augmente les performances du circuit (mais moins que si la tension d'alimentation restait la même). Le tableau suivant indique (en fréquence normalisée) la vitesse de fonctionnement dans 4 configurations :

	0.5 µm	0.35 µm
5 V	1	2 (en théorie)
3.3 V	0.66	1.3

On voit qu'en passant de 0.5 µm (5 V) à 0.35 µm (3.3 V), on gagne 30 % en vitesse et on diminue de 60 % la consommation.

Il y a un deuxième problème important à prendre en compte en dessous (environ) de 0.5 µm. La puissance maximale dissipée par le circuit intégré peut être supérieure à celle dissipée par son boîtier. On peut donc réussir à détruire un circuit en fonctionnement normal pour la simple raison qu'il fonctionne trop rapidement. La figure suivante en est une illustration. On a pris un circuit logique programmable (XC4036EX de chez Xilinx) que l'on a entièrement rempli soit de compteurs 8 bits, soit de compteurs 16 bits. Les traits en pointillés représentent la puissance maximale dissipée pour trois types de boîtiers.



On voit nettement qu'au-delà d'une certaine fréquence de fonctionnement (25 à 40 MHz en 8 bits), il est nécessaire d'utiliser un radiateur et/ou une ventilation pour augmenter la puissance dissipée par le boîtier sous peine de destruction du circuit intégré. Ce phénomène est bien connu dans le domaine des microprocesseurs avec l'apparition d'un ventilateur intégré dans le radiateur des Pentium d'Intel par exemple.

7.3.6 Méthodes pour limiter la consommation

Il y a peu de méthodes permettant de réduire la consommation d'un circuit intégré CMOS. Elles concernent principalement le domaine des applications portables fonctionnant sur batterie (contrainte de consommation) et le domaine des applications embarquées (contrainte sur la puissance dissipée). On peut en citer quatre :

- L'alimentation par bloc ; on n'alimente le bloc logique considéré que lorsqu'il est utilisé. Quand il est au repos, on coupe son alimentation. Cette méthode est efficace car elle est la seule qui traite correctement la puissance statique dissipée. Elle doit être prévue au moment de la conception de l'ASIC et est inapplicable aux FPGA. On l'utilise surtout pour les téléphones type GSM.
- La conception synchrone par bloc ; on créé autant d'horloges qu'il y a de blocs fonctionnels dans le design, chaque bloc fonctionnant à sa fréquence d'horloge minimale. Il faut veiller aux échanges de données entre les différents domaines d'horloge. Cette méthode fonctionne aussi bien dans les ASICS que dans les FPGA.
- La conception full synchrone sans utiliser d'arbre de distribution d'horloge ; on route l'horloge comme une donnée et on gère le skew avec une contrainte sur le net et en plaçant la logique à la main dans le circuit. Ce genre de méthode est réservé au designer très expérimenté (interdit au débutant).
- La conception asynchrone ; cette méthode efficace est réservée au designer très expérimenté (interdit au débutant).

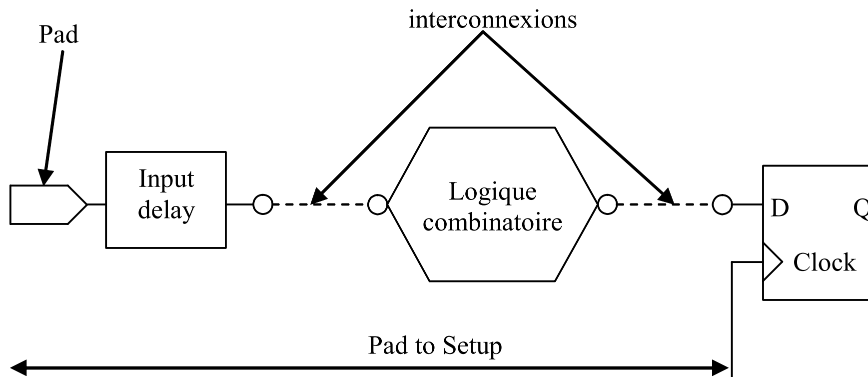
En pratique, seules les deux premières méthodes sont utilisées.

7.4 Augmentation de la fréquence de fonctionnement

7.4.1 Modèles de timing

Quelque soit le type de circuit numérique utilisé, il n'existe que 4 modèles de timing permettant de déterminer les caractéristiques temporelles d'un design synchrone :

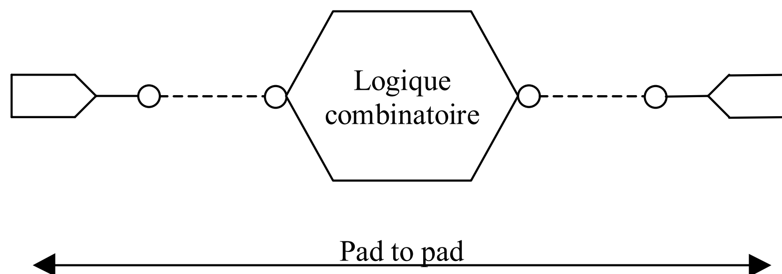
- Pad to Setup



On a un signal sur une broche d'entrée (pad) et l'on souhaite qu'il soit pris en compte sur le front actif d'une bascule D. Le temps « pad to Setup » est le temps de setup à prendre en compte entre la broche d'entrée et le front actif de l'horloge. Le retard Idelay permet de faciliter la lecture d'un bus de données accompagné de son horloge. Dans l'exemple général ci-dessus, il est égal à :

$$T_{\text{Pad to Setup}} = T_{\text{propagation Pad}} + \text{Input delay} + T_{\text{prop interconnexion}} + T_{\text{prop logique}} + T_{\text{prop interconnexion}} + T_{\text{setup bascule}}$$

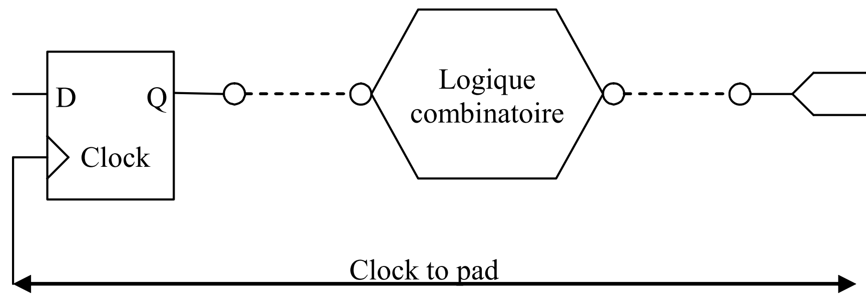
- Pad to pad



C'est le temps de propagation entre une broche d'entrée et une broche de sortie dans le cas d'un circuit purement combinatoire. Dans l'exemple général ci-dessus, il est égal à :

$$T_{\text{Pad to pad}} = T_{\text{prop Pad}} + T_{\text{prop interconnexion}} + T_{\text{prop logique}} + T_{\text{prop interconnexion}} + T_{\text{prop Pad}}$$

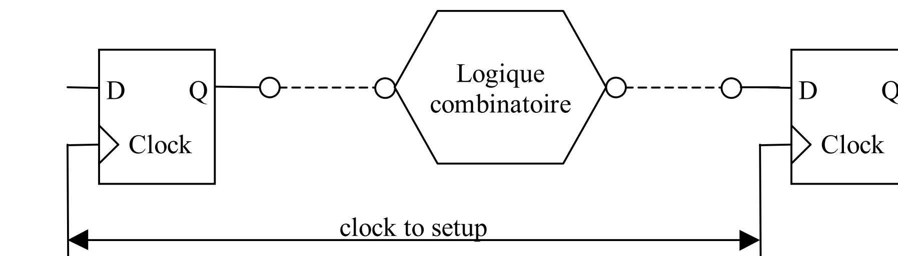
- Clock to pad



On souhaite mesurer le temps qui sépare le front actif de l'horloge et l'arrivée du signal de sortie de la bascule sur une broche de sortie. Le temps « clock to pad » est le temps de propagation du signal entre le front actif de l'horloge et son arrivée sur une broche de sortie. Dans l'exemple général ci-dessus, il est égal à :

$$T_{\text{clock to pad}} = T_{\text{prop clock to Q}} + T_{\text{prop interconnexion}} + T_{\text{prop logique}} + T_{\text{prop interconnexion}} + T_{\text{prop Pad}}$$

- Clock to setup



On a un signal sur une sortie de bascule D et l'on souhaite qu'il soit pris en compte sur le front actif d'une deuxième bascule D, les deux bascules étant commandées par la même horloge. Le temps « clock to Setup » est le temps à prendre en compte entre le front actif de l'horloge et le front suivant de cette même horloge. Dans l'exemple général ci-dessus, il est égal à :

$$T_{\text{clock to setup}} = T_{\text{prop clock to Q}} + T_{\text{prop interconnexion}} + T_{\text{prop logique}} + T_{\text{prop interconnexion}} + T_{\text{setup bascule}}$$

L'analyse temporelle permet de calculer les différents temps de propagation à l'intérieur du circuit en vue de calculer par exemple la fréquence de fonctionnement du design ou bien le temps de propagation d'un signal sur une entrée ou sur une sortie. Après placement routage, l'outil de CAO construit une base de données contenant toutes les caractéristiques temporelles du circuit. Le concepteur peut accéder à cette base de données pour effectuer **l'analyse statique de timing** (outil « timing analyzer » chez Xilinx). Ce logiciel permet d'analyser le comportement temporel du circuit afin de modifier l'architecture du design en vue d'améliorer ses caractéristiques temporelles.

7.4.2 Mise en œuvre des contraintes

Les contraintes permettent à l'utilisateur de contrôler les opérations de synthèse, de mapping et de placement-routage. Les contraintes peuvent avoir trois origines :

1. elles peuvent être spécifiées par l'utilisateur en configurant les propriétés du synthétiseur par le biais de l'interface graphique (optimisation surface, vitesse, effort d'optimisation, fréquence d'horloge, …). Le synthétiseur déterminera les contraintes de mapping et de placement-routage et les passera automatiquement à l'outil d'implémentation.
2. elles peuvent être spécifiées par l'utilisateur dans un fichier séparé (fichier XCF) pour guider le synthétiseur. Le synthétiseur déterminera les contraintes de mapping et de placement-routage et les passera automatiquement à l'outil d'implémentation.
3. elles peuvent être spécifiées par l'utilisateur dans un fichier séparé (fichier UCF) pour guider directement l'implémentation.

Dans notre cas, nous mettrons toujours les contraintes dans un fichier UCF séparé du design afin de guider directement l'implémentation. Vous pouvez principalement spécifier dans un fichier de contraintes :

- l'affectation d'une broche logique du design sur une broche physique du FPGA,
- le slew-rate d'une broche physique,
- l'emplacement d'une primitive à l'intérieur du composant (placement relatif ou absolu),
- le respect d'une période d'horloge maximale,
- le respect d'un temps de propagation maximal d'un point à un autre du circuit.

Attardons-nous sur les contraintes temporelles. Dans le cas général, la fréquence de fonctionnement est déterminée par :

$$f_{\max} = \frac{1}{t_{p\,clock\,to\,Q} + t_{p\,critique\,logique} + t_{p\,critique\,routage} + t_{setup}}$$

Le temps critique peut être décomposé en deux parties : le temps de propagation dans les circuits logiques combinatoires (les LUT principalement) et le temps de propagation dans les interconnexions. Les temps $t_{p\ clock\ to\ Q}$, $t_{p\ critique\ logique}$ et t_{setup} sont fixes. Ils sont déterminés par le processus de fabrication du circuit et ne peuvent en aucun cas être modifiés par l'outil d'implémentation. Mettre une contrainte sur l'horloge supérieure à :

$$f_{\max\max} = \frac{1}{t_{p\,clock\,to\,Q} + t_{p\,critique\,logique} + t_{setup}}$$

n'a aucun sens. En effet, on considère que les délais d'interconnexion sont nuls. $f_{max\ max}$ est donc la limite supérieure que l'on ne peut pas dépasser. Mais alors à quoi sert donc la contrainte d'horloge ? Elle sert à optimiser les délais d'interconnexion. En jouant sur le mapping, le placement de la logique et le routage, on peut réduire les délais d'interconnexion pour respecter une contrainte d'horloge ou encore un temps de propagation d'un signal dans le circuit. Mais il ne faut rien exagérer. Si par exemple vous placez une contrainte irréaliste sur l'horloge, deux cas sont possibles :

- La contrainte est supérieure à $f_{max\ max}$. Au moment du mapping, l'outil d'implémentation détecte l'incohérence et s'arrête avec un message d'erreur.
- La contrainte est inférieure à $f_{max\ max}$, mais non réalisable. Le temps d'exécution du placement-routage devient très grand, voire infini.

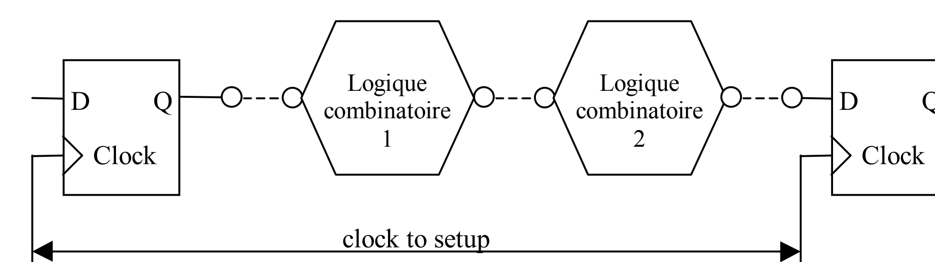
Comment va-t-on déterminer cette contrainte ? On procède de la manière suivante :

1. on implémente une première fois avec une contrainte très facile à réaliser pour voir l'ordre de grandeur de la fréquence obtenue.
2. on place une contrainte proche de la fréquence obtenue précédemment. Si elle correspond à la fréquence souhaitée, on arrête là.
3. Si la fréquence obtenue est proche de la valeur désirée (moins de 10 %), on peut ajuster les propriétés de l'implémentation en augmentant l'effort de placement et de routage.
4. Si on est trop loin (plus de 10 %), il faut changer l'architecture du design en utilisant par exemple la méthode du pipeline.

ATTENTION : en ce qui concerne la contrainte d'horloge, il faut tenir compte du jitter de cette horloge. Il a pour conséquence de faire varier la période rapidement au cours du temps. La contrainte doit donc être exprimée en tenant compte de la période minimale possible. En général, on prend une marge de 5 à 10 % sur la valeur nominale de l'horloge.

7.4.3 Méthode du pipeline

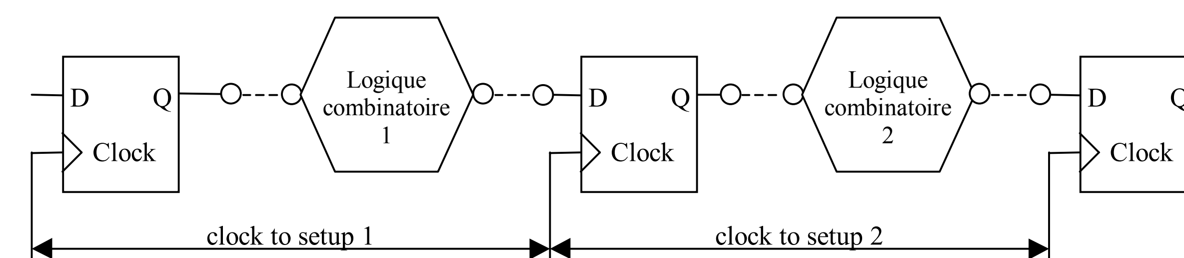
Quand on dispose d'un circuit intégré ayant de nombreuses bascules (FPGA), on a tout intérêt à utiliser une architecture de type pipeline pour augmenter la fréquence de fonctionnement du design. L'objectif de l'architecture en pipeline est de placer des registres entre chaque couche de fonctions combinatoires afin de diminuer le temps critique et par conséquence le temps clock to setup. Dans le schéma suivant, il y a deux couches logiques (deux niveaux de LUT par exemple) entre les deux bascules.



On a donc la relation (avec en gras, le temps critique):

$$t_{\text{clock to setup}} = t_{\text{p clock to Q}} + \mathbf{t_{p\ net}} + \mathbf{t_{p\ logique\ 1}} + \mathbf{t_{p\ net}} + \mathbf{t_{p\ logique\ 2}} + \mathbf{t_{p\ net}} + t_{\text{setup}}$$

En modifiant le design de la manière suivante :



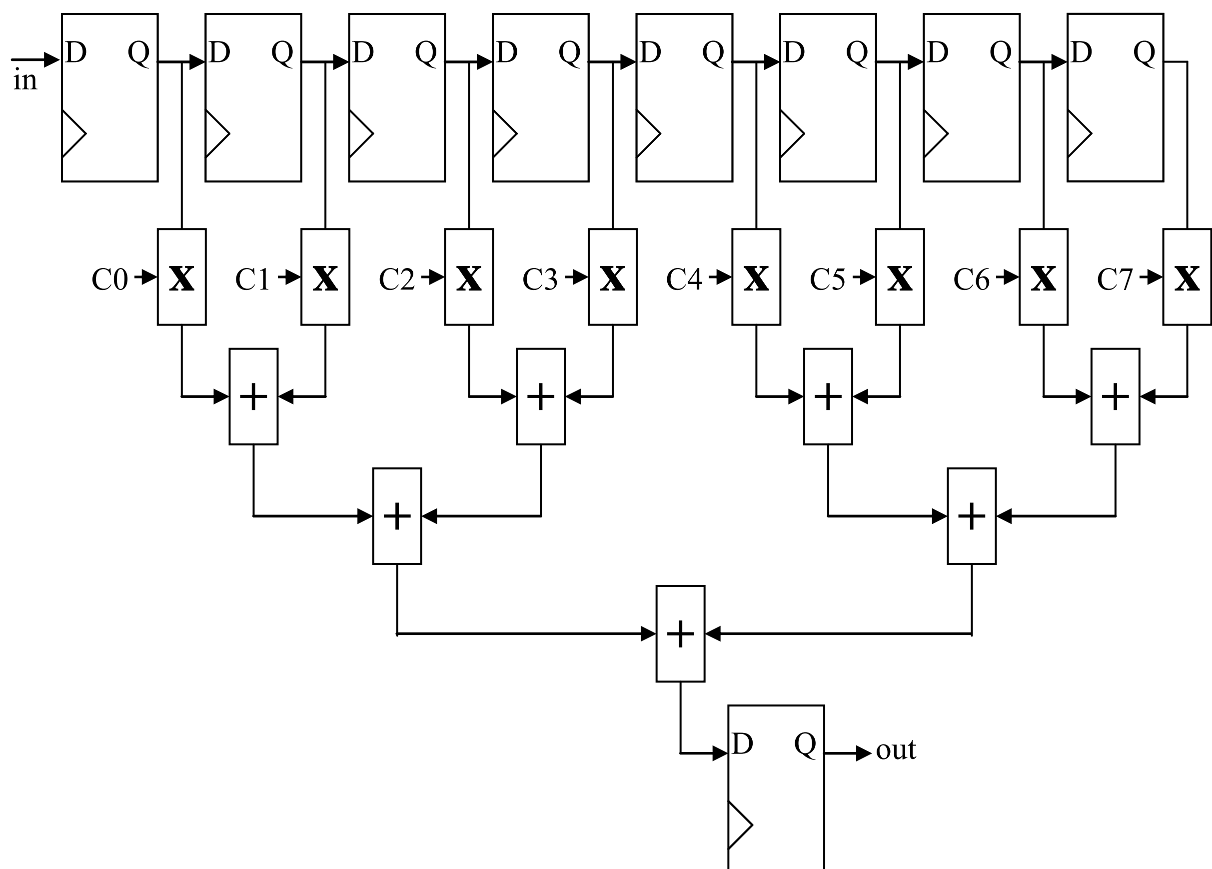
On obtient les relations :

$$t_{\text{clock to setup 1}} = t_{\text{p clock to Q}} + \mathbf{t_{p\ net}} + \mathbf{t_{p\ logique\ 1}} + \mathbf{t_{p\ net}} + t_{\text{setup}}$$

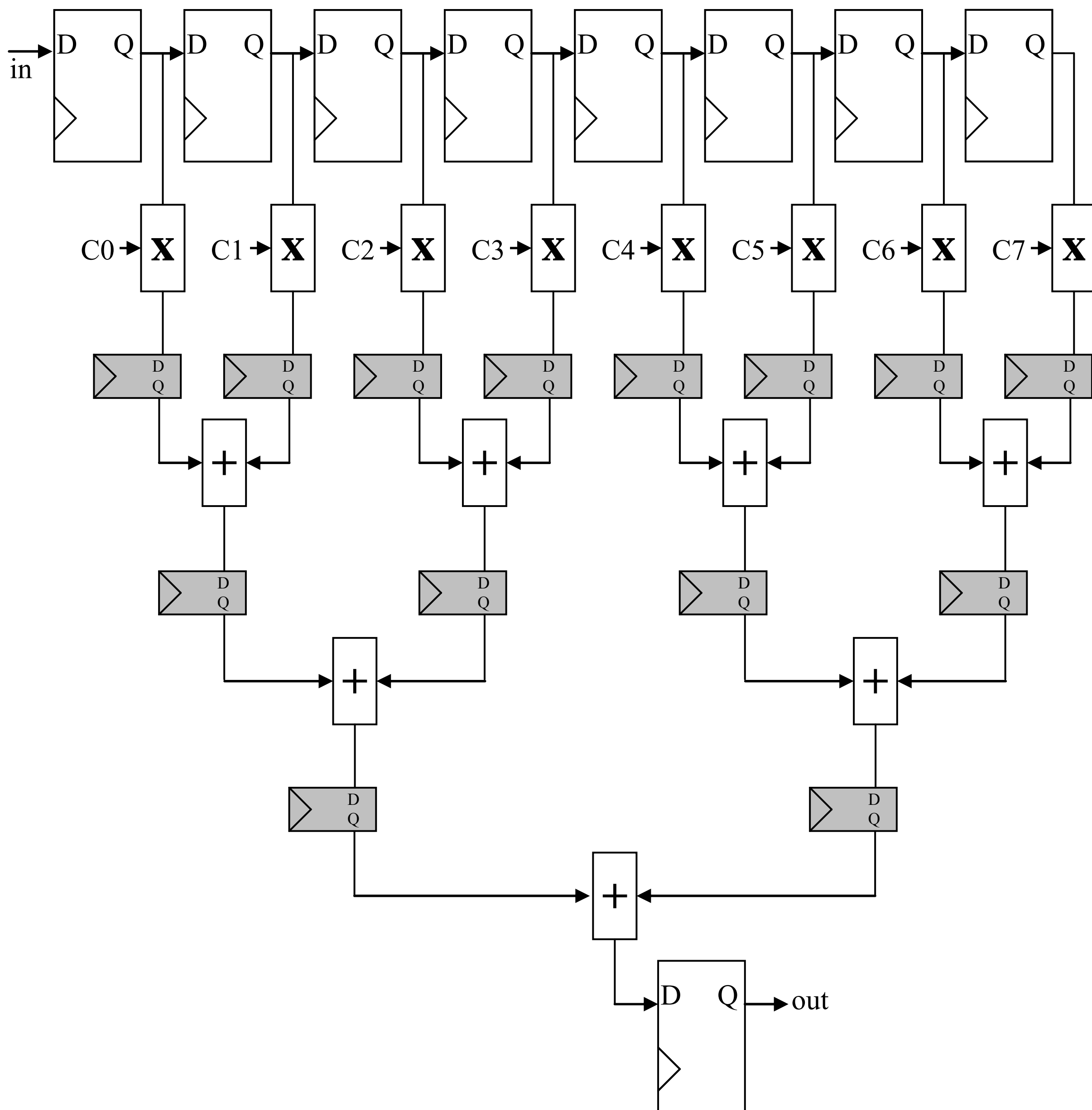
$$t_{\text{clock to setup 2}} = t_{\text{p clock to Q}} + \mathbf{t_{p\ net}} + \mathbf{t_{p\ logique\ 2}} + \mathbf{t_{p\ net}} + t_{\text{setup}}$$

Le plus élevé de ces deux temps ($t_{\text{clock to setup X}}$) correspond à la période minimale de l'horloge. Il est évident que $t_{\text{clock to setup X}}$ est toujours inférieur au $t_{\text{clock to setup}}$ du premier montage, car on a réduit la longueur du chemin critique. Le pipeline a bien augmenté la fréquence de fonctionnement du design. En contrepartie, on a retardé le signal de sortie d'un coup d'horloge. On parle de **latence** ou **temps de latence** pour désigner le retard introduit. Tout ce passe comme si on échangeait une fréquence de fonctionnement plus élevée contre un temps de latence supplémentaire. Ce retard est égal au nombre de couches de bascules inséré multiplié par la période de l'horloge.

Reprenons l'exemple d'un FIR à structure parallèle 8 coefficients :



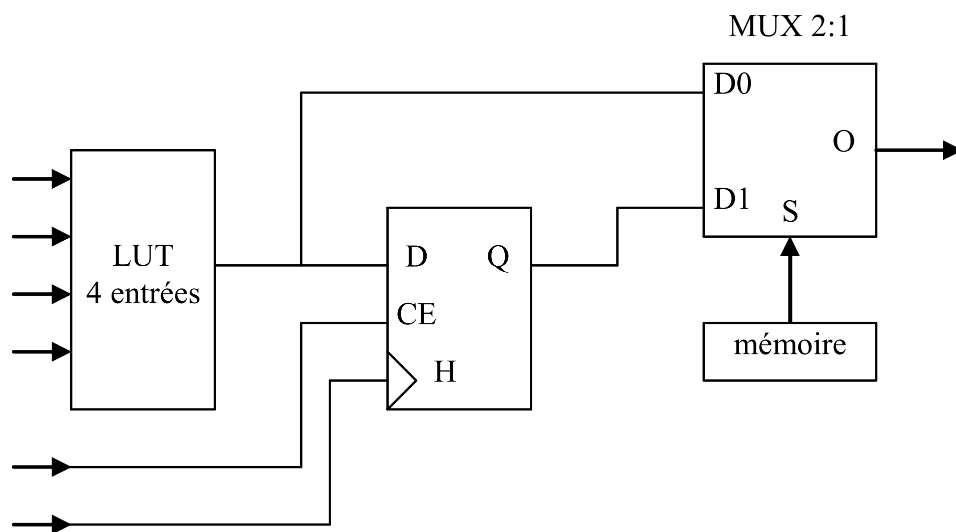
Quel est le chemin critique ? Entre deux fronts actifs d'horloge, les sorties du registre à décalage doivent traverser un multiplieur plus trois couches d'additionneurs avant d'arriver sur le registre de sortie. L'échantillon filtré sera disponible en sortie avec une période d'horloge de retard par rapport à l'entrée. Pour réduire la longueur du chemin critique, nous allons insérer des registres après le multiplieur et chaque additionneur :



Quel est le nouveau chemin critique ? A priori, il doit s'agir du temps de traversé du multiplieur. On a rajouté 3 périodes d'horloge de retard sur l'échantillon de sortie, donc le temps de latence est maintenant de 4 périodes d'horloge. Par contre la fréquence de fonctionnement (hors délai d'interconnections) a augmenté :

f_{max} sans pipeline	f_{max} avec pipeline
$f\ \max = \dfrac{1}{t_{p\,clock\,to\,Q} + t_{p\,mult} + 3.t_{p\,add} + t_{setup}}$	$f\ \max = \dfrac{1}{t_{p\,clock\,to\,Q} + t_{p\,mult} + t_{setup}}$

Cela nous a couté 14 registres supplémentaires. Il faut noter que cela n'est pas si important dans le cas d'un FPGA. En effet, la cellule de base d'un FPGA est constituée d'une LUT 4 entrées associée à une bascule D :



La LUT est conçue pour réaliser un additionneur complet sur 1 bit, la bascule D fournissant le registre associé. La structure interne du bloc logique d'un FPGA est conçue pour le pipeline. Il est souhaitable de maintenir un équilibre entre le nombre de LUT et le nombre de bascules D utilisés. Vous ne payerez pas le circuit moins cher si vous n'utilisez pas les bascules.

En ce qui concerne la description en VHDL, le pipeline est réalisé très simplement. Il suffit de mettre les assignations des signaux dans la branche sensible à l'horloge et le design est pipeliné automatiquement. Reprenons par exemple l'additionneur 4 entrées vu au §3.3.4.3. La version purement combinatoire était la suivante :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity add is
  port(A : in std_logic_vector(3 downto 0);
       B : in std_logic_vector(3 downto 0);
       C : in std_logic_vector(3 downto 0);
       D : in std_logic_vector(3 downto 0);
       S : out std_logic_vector(5 downto 0));
end add;

architecture comporte of add is
     signal S1 : std_logic_vector(5 downto 0);
     signal S2 : std_logic_vector(5 downto 0);
begin
     S1 <= ("00"&A) + ("00"&B);
     S2 <= ("00"&C) + ("00"&D);
     S <= S1 + S2;
end comporte ;
```

La version pipelinée sera simplement :

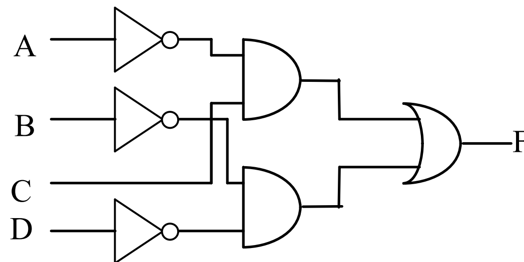
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity add is
  port(A, B, C, D : in std_logic_vector(3 downto 0);
       CLK : in std_logic;
       CLEAR : in std_logic;
       S : out std_logic_vector(5 downto 0));
end add;

architecture comporte of add is
     signal S1 : std_logic_vector(5 downto 0);
     signal S2 : std_logic_vector(5 downto 0);
begin
     process(CLK, CLEAR) begin
          if (CLEAR='1') then
               S1 <= (others => '0');
               S2 <= (others => '0');
               S <= (others => '0');
          elsif (CLK'event and CLK='1') then
               S1 <= ("00"&A) + ("00"&B);
               S2 <= ("00"&C) + ("00"&D);
               S <= S1 + S2;
          end if;
     end process;
end comporte ;
```

8 Corrigés succincts

8.1 Corrigés chapitre 2

Exercice 2.1

1. Voir cours.
2. $Y = A.B.C.D$, $Y = \overline{A + B + C + D}$, $Y = A+B+C+D$, $Y = \overline{A.B.C.D}$, $Y = \overline{A}.B.C.\overline{D}$.
3. $F1 = A + \overline{B}$, $F2 = A.C + B.\overline{C}$, $F3 = \overline{A}.B + A.\overline{B}$, $F4 = 1$.
4. voir cours.
5. voir cours.
6. $\overline{A}.\overline{B} + A.B$, $\overline{A}.B.C + \overline{B}.C.\overline{D} + A.\overline{C}.\overline{D}$, $\overline{A \oplus B \oplus C}$.
7. $S = \overline{A.B}$.
8.



Exercice 2.2

1. $F = S_1.A.B + S_1.S_0.\overline{A}.\overline{B} + C.\overline{S_1}.\overline{A.B}.\overline{\overline{A}.\overline{B}.S_0}$.
2.

C	S_1	S_0	F
0	0	0	0
0	0	1	0
0	1	0	$A.B$
0	1	1	$A.B + \overline{A}.\overline{B}$
1	0	0	$\overline{A.B}$
1	0	1	$A.\overline{B} + \overline{A}.B$
1	1	0	$A.B$
1	1	1	$A.B + \overline{A}.\overline{B}$

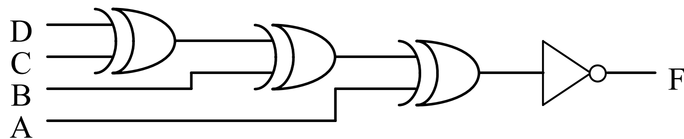
3. F=A.B si S_1S_0 = 10, F=$\overline{A.B}$ si CS_1S_0 = 100, F=$A \oplus B$ si CS_1S_0 = 101, F=$\overline{A \oplus B}$ si S_1S_0 = 11.

Exercice 2.3

1.

D	C	B	A	P
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

2.



Exercice 2.4

1. $Y_1 = \overline{A}$, $Y_2 = \overline{A} + \overline{B}$, $Y_3 = 1$.

Exercice 2.5

$Y_1 = \overline{\overline{\overline{B.C}.\overline{A.\overline{B}}}}$, implantation avec 5 NAND à 2 entrées.

$Y_1 = \overline{\overline{\overline{A+B}+\overline{C+\overline{B}}}}$, implantation avec 5 NOR à 2 entrées.

$Y_2 = \overline{A.\overline{B.\overline{C}}}$, implantation avec 3 NAND à 2 entrées.

$Y_2 = \overline{\overline{\overline{A}+\overline{\overline{B}+C}}}$, implantation avec 5 NOR à 2 entrées.

$Y_3 = \overline{\overline{B.D}.\overline{\overline{B}.\overline{D}}}$, implantation avec 5 NAND à 2 entrées.

$Y_3 = \overline{\overline{\overline{\overline{B}+\overline{D}}+\overline{B+D}}}$, implantation avec 6 NOR à 2 entrées.

$Y_4 = \overline{B.\overline{A.\overline{D}}}$, implantation avec 3 NAND à 2 entrées.

$Y_4 = \overline{\overline{\overline{B} + \overline{\overline{A} + D}}}$, implantation avec 5 NOR à 2 entrées.

$Y_5 = \overline{\overline{\overline{\overline{A}.C}.\overline{B.\overline{C}}}}$, implantation avec 6 NAND à 2 entrées.

$Y_5 = \overline{\overline{A + \overline{C}} + \overline{\overline{B} + C}}$, implantation avec 5 NOR à 2 entrées.

$Y_6 = \overline{\overline{\overline{B}.\overline{C}}}$, implantation avec 4 NAND à 2 entrées.

$Y_6 = \overline{B + C}$, implantation avec 1 NOR à 2 entrées.

$Y_7 = \overline{\overline{\overline{B}.\overline{\overline{A}.C}}}$, implantation avec 5 NAND à 2 entrées.

$Y_7 = \overline{B + \overline{A + \overline{C}}}$, implantation avec 3 NOR à 2 entrées.

Exercice 2.6

1.

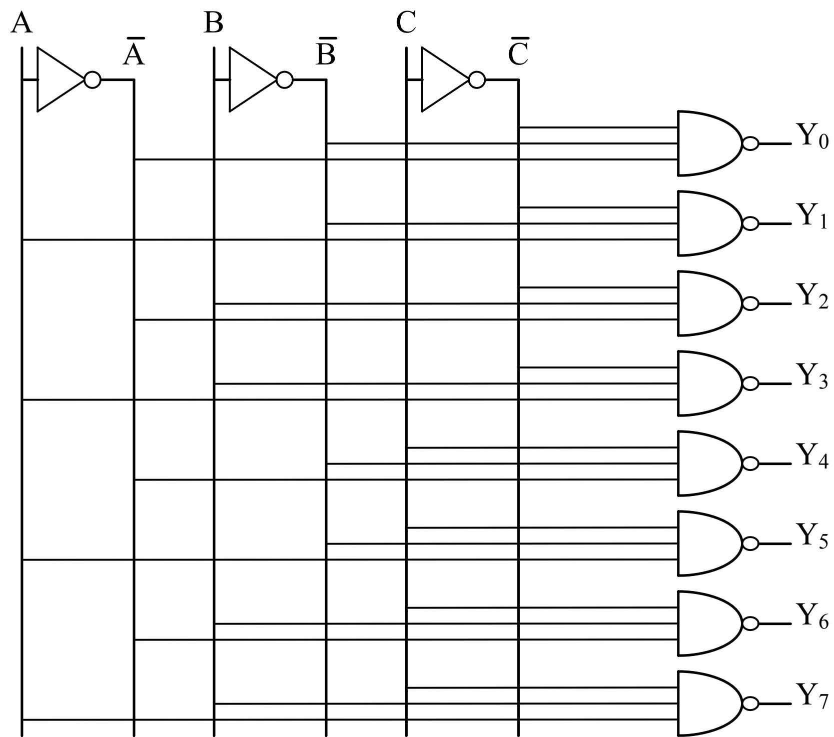
Décimal	C	B	A	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1	1	1	1	1
2	0	1	0	1	1	0	1	1	1	1	1
3	0	1	1	1	1	1	0	1	1	1	1
4	1	0	0	1	1	1	1	0	1	1	1
5	1	0	1	1	1	1	1	1	0	1	1
6	1	1	0	1	1	1	1	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1	0

2. $Y_0 = C + B + A = \overline{\overline{C}.\overline{B}.\overline{A}}$, $Y_1 = C + B + \overline{A} = \overline{\overline{C}.\overline{B}.A}$, $Y_2 = C + \overline{B} + A = \overline{\overline{C}.B.\overline{A}}$,

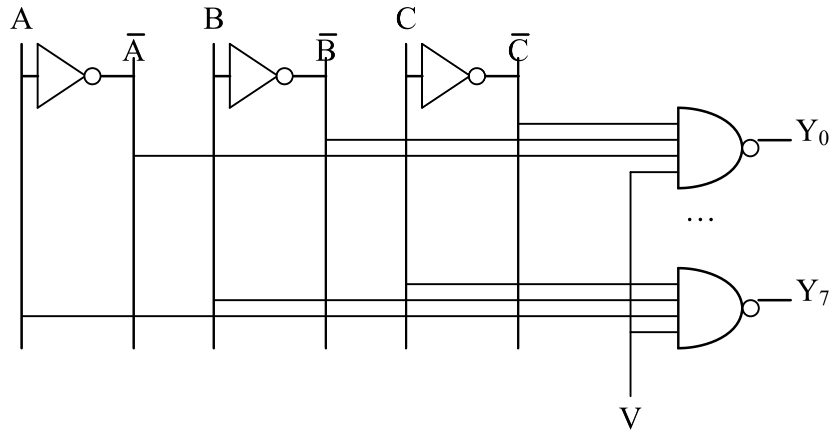
$Y_3 = C + \overline{B} + \overline{A} = \overline{\overline{C}.B.A}$, $Y_4 = \overline{C} + B + A = \overline{C.\overline{B}.\overline{A}}$, $Y_5 = \overline{C} + B + \overline{A} = \overline{C.\overline{B}.A}$,

$Y_6 = \overline{C} + \overline{B} + A = \overline{C.B.\overline{A}}$, $Y_7 = \overline{C} + \overline{B} + \overline{A} = \overline{C.B.A}$.

3.



4.

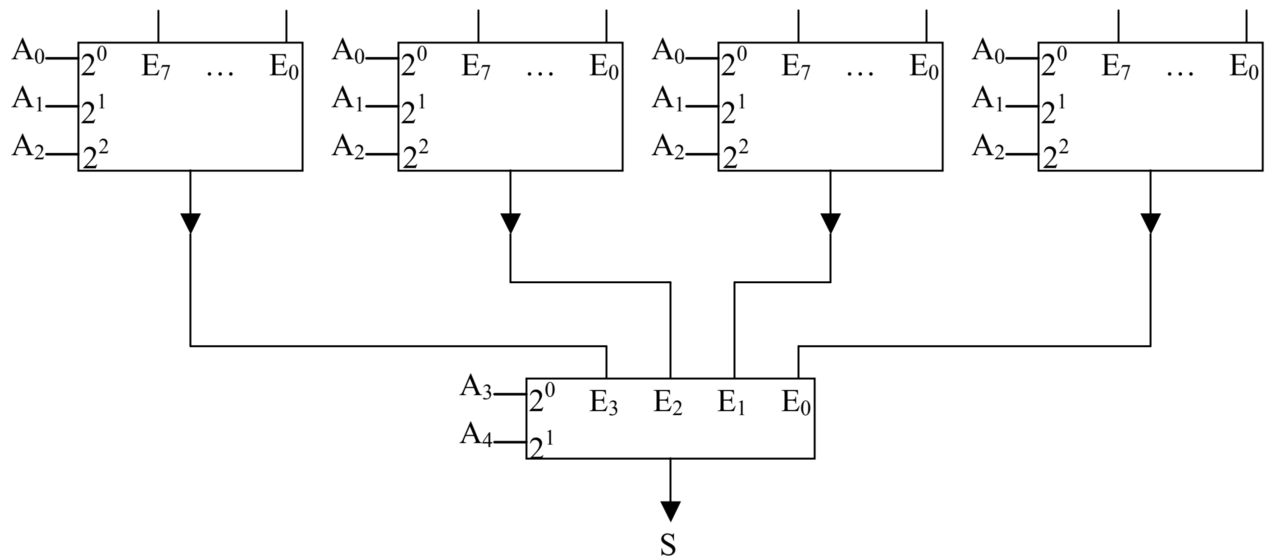


Exercice 2.7

1. $S = (A \oplus B).(E \oplus (\overline{C} + D))$.

Exercice 2.8

1.



Exercice 2.9

1.

e	p	m	c	E	P	M	C
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	1
0	1	1	0	1	0	1	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	1	1	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	0
1	1	1	1	0	1	0	0

2. $M = p.m.\bar{c}$, $C = p.\overline{m}.c$, $P = p.\overline{m}.\bar{c} + p.m.c$, $E = e.\overline{m}.\bar{c} + C + M$.

3. $A_0 = c$, $A_1 = m$, $A_2 = p$, $A_3 = e$, $A_4 = 0$. $D_0 = E$, $D_1 = M$, $D_2 = C$, $D_3 = P$. Contenu PROM = table de vérité.

4. Réalisation avec des NAND des équations $M = p.m.\bar{c}$, $C = p.\overline{m}.c$, $P = p.\overline{m}.\bar{c} + p.m.c$, $E = e.\overline{m}.\bar{c} + C + M$.

Exercice 2.10

1.

nb	E	D	C	B	A	S	T	U
0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0
2	0	0	0	1	0	0	0	0
3	0	0	0	1	1	1	0	0
4	0	0	1	0	0	0	0	0
5	0	0	1	0	1	0	1	0
6	0	0	1	1	0	1	0	0
7	0	0	1	1	1	0	0	1
8	0	1	0	0	0	0	0	0
9	0	1	0	0	1	1	0	0
10	0	1	0	1	0	0	1	0
11	0	1	0	1	1	0	0	0
12	0	1	1	0	0	1	0	0
13	0	1	1	0	1	0	0	0
14	0	1	1	1	0	0	0	1
15	0	1	1	1	1	1	1	0
16	1	0	0	0	0	0	0	0
17	1	0	0	0	1	0	0	0
18	1	0	0	1	0	1	0	0
19	1	0	0	1	1	0	0	0
20	1	0	1	0	0	0	1	0

2. $S = \overline{A}.B.E + A.B.\overline{C}.\overline{D}.\overline{E} + A.\overline{B}.\overline{C}.D + \overline{A}.\overline{B}.C.D + A.B.C.D + \overline{A}.B.C.\overline{D}$,

   $T = A.\overline{B}.C.\overline{D} + A.B.C.D + \overline{A}.B.\overline{C}.D + C.E$, $U = A.B.C.\overline{D} + \overline{A}.B.C.D$.

3. Voir : théorème de « De Morgan » + formules précédentes.

4. $A_0 = A$, $A_1 = B$, $A_2 = C$, $A_3 = D$, $A_4 = E$. $D_0 = S$, $D_1 = T$, $D_2 = U$. Contenu PROM = table de vérité.

Exercice 2.11

1.

A B C D EQ NE LT GT

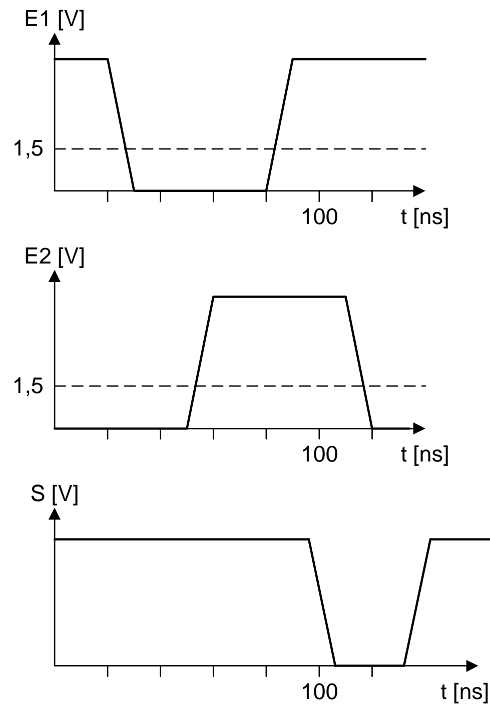
0	0	0	0	1	0	0	0
0	0	0	1	0	1	1	0
0	0	1	0	0	1	1	0
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	1	0	0	0
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	0
1	0	0	0	0	1	0	1
1	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	1	0	1
1	1	0	1	0	1	0	1
1	1	1	0	0	1	0	1
1	1	1	1	1	0	0	0

2. $EQ = \overline{A}.\overline{B}.\overline{C}.\overline{D} + \overline{A}.B.\overline{C}.D + A.B.C.D + A.\overline{B}.C.\overline{D}$, $NE = A.\overline{C} + \overline{A}.C + \overline{B}.D + B.\overline{D}$,

   $LT = \overline{A}.C + \overline{A}.\overline{B}.D + \overline{B}.C.D$, $GT = A.\overline{C} + A.B.\overline{D} + B.\overline{C}.\overline{D}$.
3. Voir : théorème de « De Morgan » + formules précédentes.
4. $A_0 = D$, $A_1 = C$, $A_2 = B$, $A_3 = A$, $A_4 = 0$. $D_0 = EQ$, $D_1 = NE$, $D_2 = LT$, $D_3 = GT$. Contenu PROM = table de vérité.

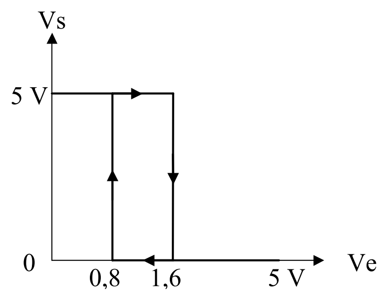
Exercice 2.12

temps de propagation = 15 ns.



Exercice 2.13

1.



2. hystérésis = V_{T+} - V_{T-} = 0,8 V.

8.2 Corrigés chapitre 3

Exercice 3.1

1. 4 bits : 0 → 15, -8 → 7 ; 8 bits : 0 → 255, -128 → 127 ; 16 bits : 0 → 65535, -32768 → 32767 ; 32 bits : 0 → 4294967295, -2147483648 → 2147483647 ; N bits : 0 → 2^{N-1}, -2^{N-1} → 2^{N-1}-1.
2. $(1101101)_2$=$(109)_{10}$.
3. $(19)_{10}$ = $(10011)_2$, $(45)_{10}$ = $(101101)_2$, $(63)_{10}$ = $(111111)_2$.

4. $(1CA57)_{16}$.
5. $(10A4)_{16} = (4260)_{10} = (1000010100100)_2$, $(CF8E)_{16} = (53134)_{10} = (1100111110001110)_2$, $(9742)_{16} = (38722)_{10} = (1001011101000010)_2$.

Exercice 3.2

1. 55h = 85d, C6h = 198d, 12h = 18d, CBh = 203d
2. 55h = 85d, C6h = -58d, 12h = 18d, CBh = -53d
3. 0055h, 00C6h, 0012h, 00CBh
4. 0055h, FFC6h, 0012h, FFCBh

Exercice 3.3

1. Resultat = 8506, correct sur 4 chiffres (retenue = 0). Resultat = 1110, faux sur 4 chiffres (retenue $\neq$ 0), vrai sur 5 chiffres. Resultat = 5509, correct sur 4 chiffres (retenue = 0). Resultat = 4491, faux sur 4 chiffres (retenue $\neq$ 0), vrai avec complément à 10000 et signe – (-(10000-4491)=-5509).
2. Resultat = 82h, C6 = 1, C7 = 0, V = 1, résultat faux. Resultat = 2Ah, C6 = 1, C7 = 1, V = 0, résultat juste. Resultat = D6h, C6 = 0, C7 = 0, V = 0, résultat juste. Resultat = 68h, C6 = 0, C7 = 1, V = 1, résultat faux. Resultat = 98h, C6 = 1, C7 = 0, V = 1, résultat faux.

Exercice 3.4

1.

		b_4	b_3	b_2	b_1	b_0		a_3	a_2	a_1	a_0
0		0	0	0	1	1		0	0	0	0
1		1	1	0	0	0		0	0	0	1
2		1	0	1	0	0		0	0	1	0
3		0	1	1	0	0		0	0	1	1
4		1	0	0	1	0		0	1	0	0
5		0	1	0	1	0		0	1	0	1
6		0	0	1	1	0		0	1	1	0
7		1	0	0	0	1		0	1	1	1
8		0	1	0	0	1		1	0	0	0
9		0	0	1	0	1		1	0	0	1

2.

3. $b0 = a3 + \overline{a2}.\overline{a1}.\overline{a0} + a2.a1.a0$, $b1 = \overline{a1}.a2 + \overline{a3}.\overline{a1}.\overline{a0} + a2.a1.a0$,

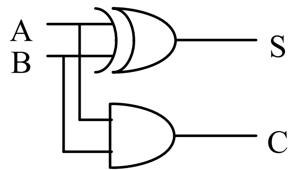
   $b2 = a1.\overline{a0} + a3.a0 + \overline{a2}.a1.a0$, $b3 = a3.\overline{a0} + \overline{a3}.\overline{a1}.a0 + \overline{a2}.a1.a0$,

   $b4 = \overline{a3}.\overline{a2}.a1.a0 + a2.\overline{a1}.\overline{a0} + a2.a1.a0 + \overline{a2}.a1.\overline{a0}$.

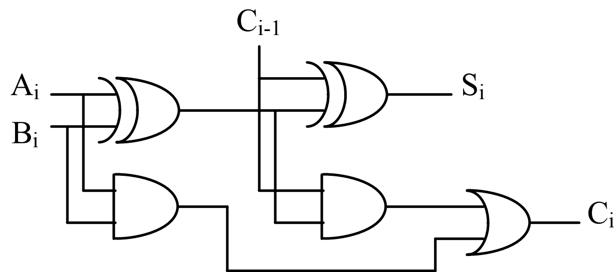
4. $b0 = \overline{\overline{a3}.\overline{\overline{a2}.\overline{a1}.\overline{a0}}.\overline{a2.a1.a0}}$ implantation avec 4 NAND à 2 entrées et 3 NAND à 3 entrées.

Exercice 3.5

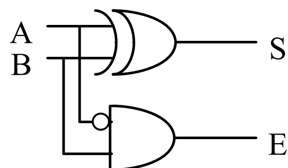
1. $S = A \oplus B$, C=A.B.



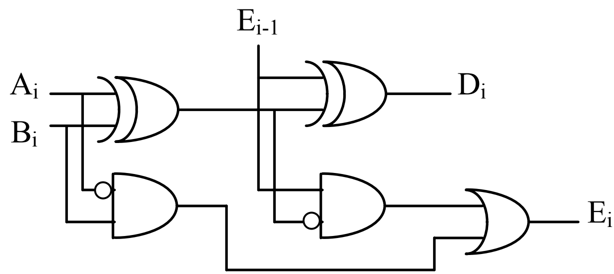
2. $S_i = C_{i-1} \oplus A_i \oplus B_i$, $C_i = C_{i-1}.(A_i \oplus B_i) + A_i.B_i$.



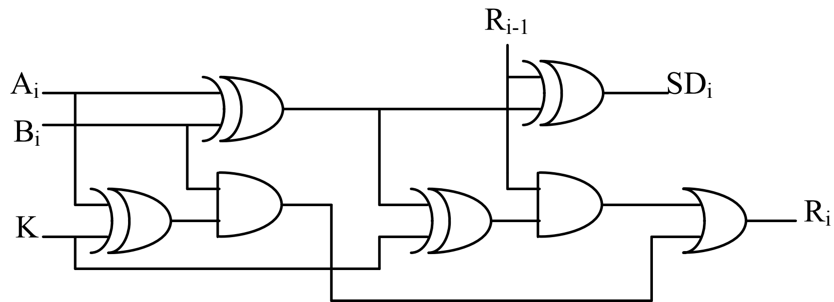
3. $D = A \oplus B$, $E = \overline{A}.B$.



4. $D_i = E_{i-1} \oplus A_i \oplus B_i$, $E_i = E_{i-1}.(\overline{A_i \oplus B_i}) + \overline{A_i}.B_i$.



5. K = 0, addition ; K = 1, soustraction.

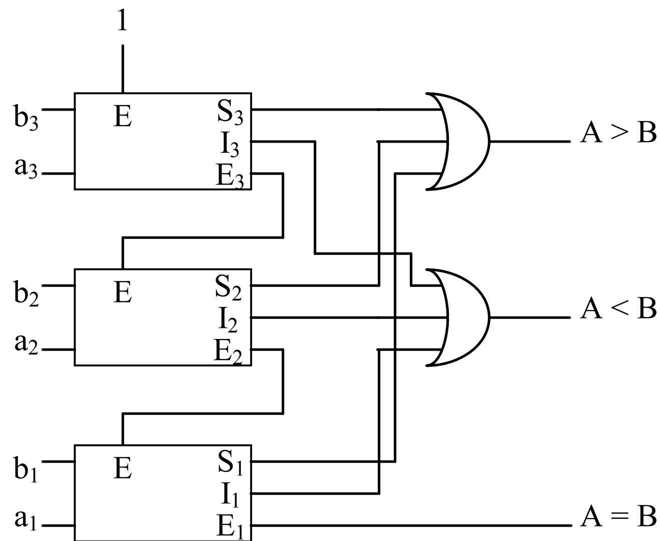


Exercice 3.6

1. C'est un comparateur d'égalité de deux nombres sur 4 bits. S = 1 si a3 = b3 et a2 = b2 et a1 = b1 et a0 = b0.

Exercice 3.7

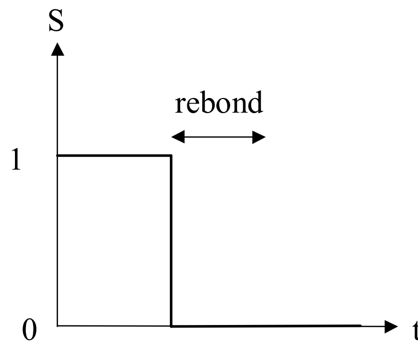
1. $S_i = E.a_i.\overline{b_i}$, $E_i = E.(\overline{S_i + I_i})$, $I_i = E.\overline{a_i}.b_i$.
2.



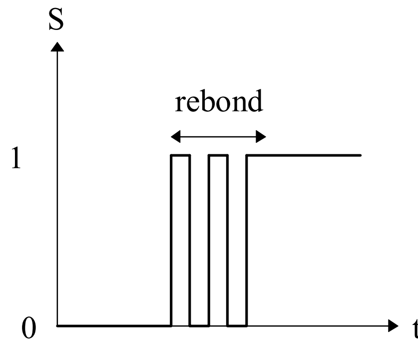
8.3 Corrigés chapitre 4

Exercice 4.1

1.



2.



3. On a maintenant une bascule SR. 2 $\rightarrow$ 1 : pendant le rebond, on passe de 'mise à 0' à 'mémoire' $\Rightarrow$ S reste à 0. 1 $\rightarrow$ 2 : pendant le rebond, on passe de 'mise à 1' à 'mémoire' $\Rightarrow$ S reste à 1.

Exercice 4.2

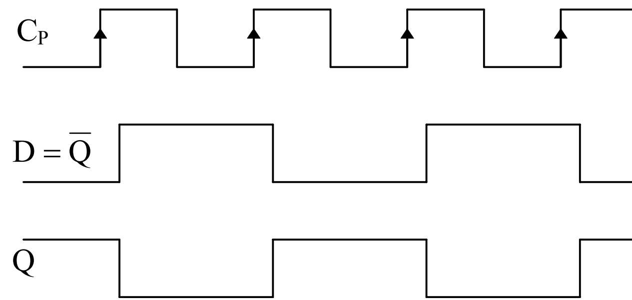
1.

H	A	B	R	S	Q^+
1	D	$\overline{D}$	0	0	Q
$\downarrow$	D	$\overline{D}$	$\overline{D}$	D	D
0	D	$\overline{D}$ ou 0 si D change	$\overline{D}$	D	Q
$\uparrow$	D	$\overline{D}$	0	0	Q

2. bascule D synchrone sur front descendant.

Exercice 4.3

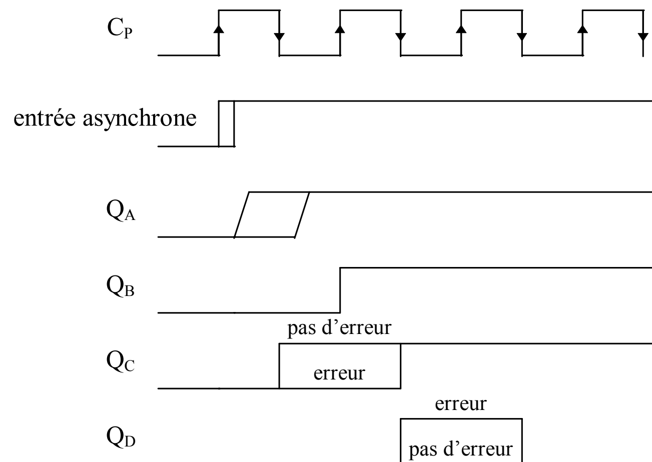
1.



2. Nand : t_P = 15 ns. Bascule : t_{PHLmax} = 40 ns, t_{smin} = 20 ns, t_{hmin} = 5 ns.
3. fmax = 11,1 MHz.

Exercice 4.4

1. si $t_{PD \to QA} < T/2$, alors pas d'erreur. si $T/2 < t_{PD \to QA} < T$, alors erreur (métastabilité).

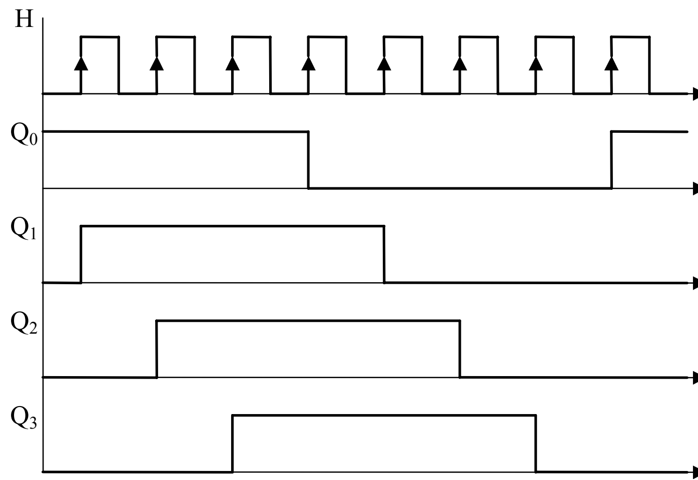


2.

t [ns]	0	0,5	1	1,5	2
MTBF	0.001 s	16,3 s	74 h	138 ans	$2{,}25.10^6$ ans

Exercice 4.5

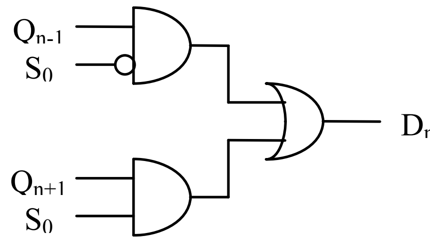
1.



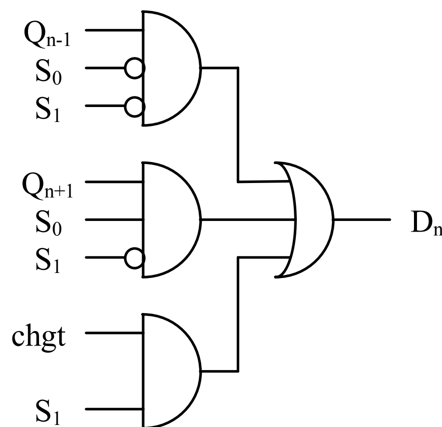
2. C'est un compteur Johnson. Une seule sortie change à chaque coup d'horloge. On peut donc réaliser des combinaisons de sorties garanties sans glitches. Autre application, les horloges décalées en phase.

Exercice 4.6

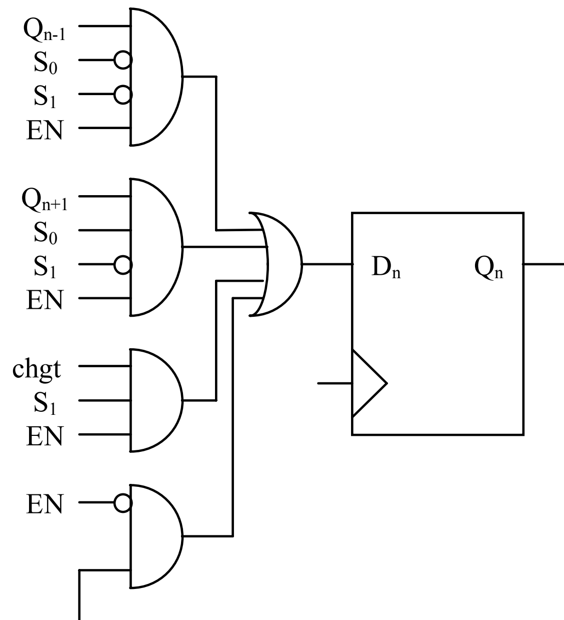
1. Registre à décalage à droite.
2. Registre à décalage à gauche.
3. $S_0 = 0$, décalage à droite. $S_0 = 1$, décalage à gauche.



4. $S_1 = 0$, décalage. $S_1 = 1$, chargement.

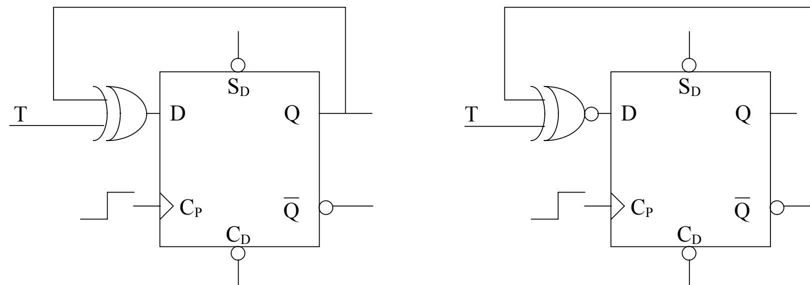


5. EN = 1, fonctionnement précédent. EN = 0, Qn = Dn.

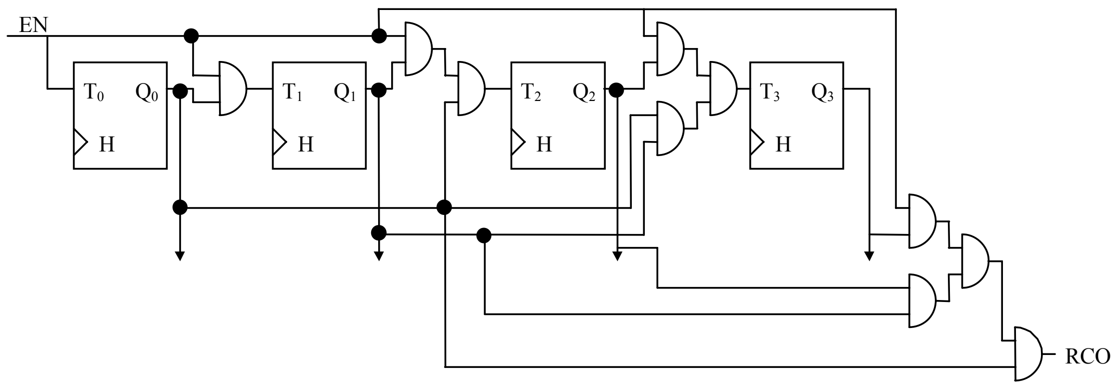


Exercice 4.7

1. fmax = 13,3 MHz.

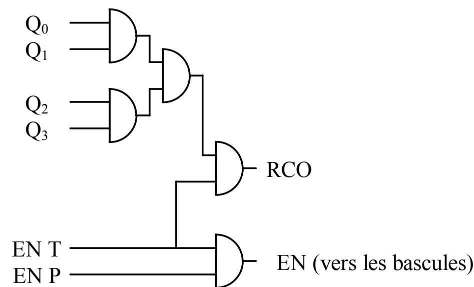


2. $T_0 = 1$, $T_1 = Q_0$, fmax = 13,3 MHz.
3. 3 bits : $T_0 = 1$, $T_1 = Q_0$, $T_2 = Q_0.Q_1$, fmax = 11,1 MHz. 4 bits : $T_0 = 1$, $T_1 = Q_0$, $T_2 = Q_0.Q_1$, $T_3 = Q_0.Q_1.Q_2$, fmax = 9,5 MHz. La taille du AND augmente avec le nombre de bits. On ne peut pas dépasser une taille limite ⇒ on fait des compteurs 4 bits et on les associe en cascade.
4. On ajoute un signal EN (validation) et un signal RCO (Ripple Carry Output vaut 1 pour Qn = 1111).



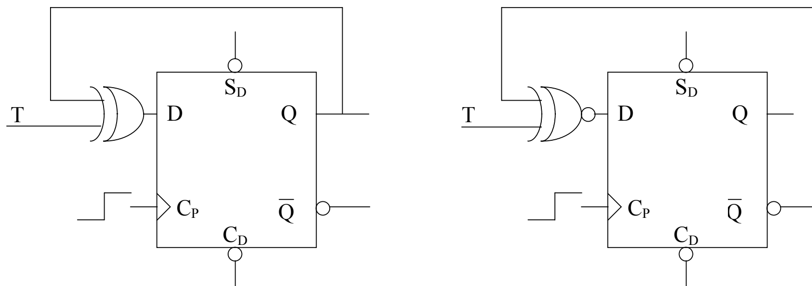
16 bits : fmax = 5,5 MHz, 32 bits : fmax = 4,2 MHz.

5. fmax = 6,1 MHz.



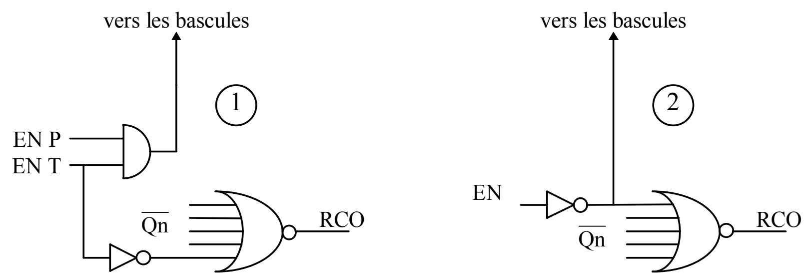
Exercice 4.8

1.



2. C'est un compteur 4 bits : $T_0 = 1$, $T_1 = Q_0$, $T_2 = Q_0.Q_1$, $T_3 = Q_0.Q_1.Q_2$. La taille du AND augmente avec le nombre de bits. On ne peut pas dépasser une taille limite ⇒ on fait des compteurs 4 bits et on les associe en cascade.
3. les Dn valent 0 ⇒ les Qn passent à 0 sur le front suivant de l'horloge.
4. $D_0 = A$, $D_1 = B$, $D_2 = C$, $D_3 = D$ ⇒ les Qn changent sur le front suivant de l'horloge.
5. RCO = 0, Tn = 0 ⇒ $Q^+n = Qn$ (effet mémoire).

6. On passe du montage 1 au montage 2.



16 bits : fmax = 6 MHz, 32 bits : fmax = 4,4 MHz.

7. On distribue parallèlement EN P. La fmax est indépendante du nombre de compteurs associés : 6,7 MHz.

9 Travaux pratiques

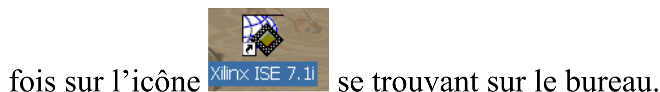
9.1 Travail pratique N°1

Ce premier TP a pour but de permettre la prise en main des outils de CAO Xilinx sur un exemple simple : le design « portes combinatoires » du §2.4.2 du cours. Les entrées seront reliées sur les interrupteurs de la maquette FPGA et les sorties sur les leds. Nous allons passer en revue toutes les phases du développement d'un design en VHDL. Ce texte a pour but de vous indiquer l'enchaînement des tâches nécessaires, les explications détaillées concernant la finalité des commandes vous seront données oralement au fur et à mesure du déroulement des travaux pratiques.

9.1.1 Ouverture de session

Appuyer sur control+Alt+Sup pour ouvrir une session sur l'ordinateur. Le nom de l'utilisateur est fpga, le mot de passe est fpga.

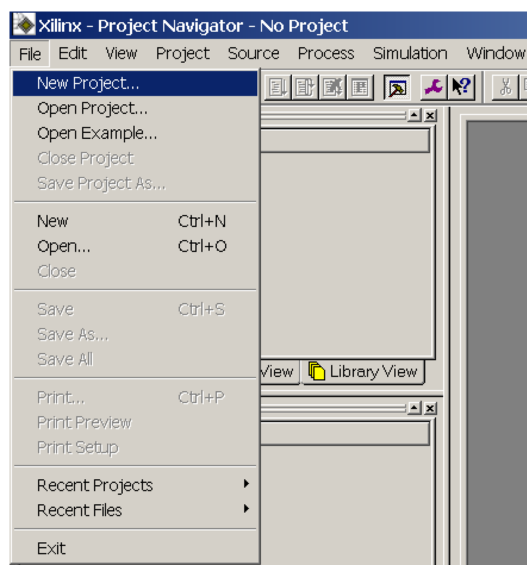
9.1.2 Lancement de « Project Navigator »

Le lancement de l'application principale « Project Navigator » s'effectue en cliquant deux fois sur l'icône 

se trouvant sur le bureau.

9.1.3 Création du projet

Après un long moment (30 secondes la première fois), la fenêtre de « Project Navigator » s'ouvre. Cliquez sur le menu « File » et le sous-menu « New Project… ».



La fenêtre de création du projet apparaît.

Dans l'ordre suivant :

1. Tapez le répertoire du projet c:\users\fpga dans le champ « Project Location »,
2. Tapez le nom du projet tp1 dans le champ « Project Name »,
3. Sélectionnez le type HDL pour le design principal.

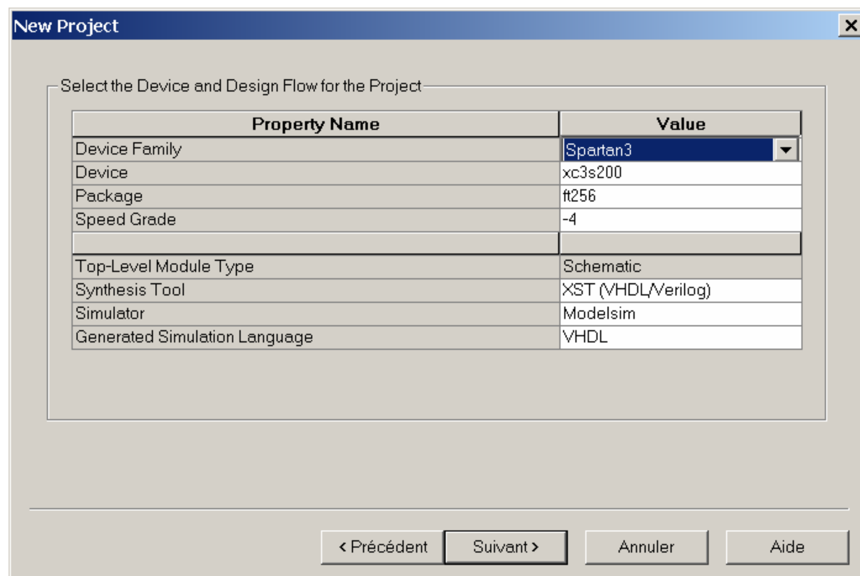
Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Suivant » :

Dans la fenêtre qui s'ouvre, sélectionnez :

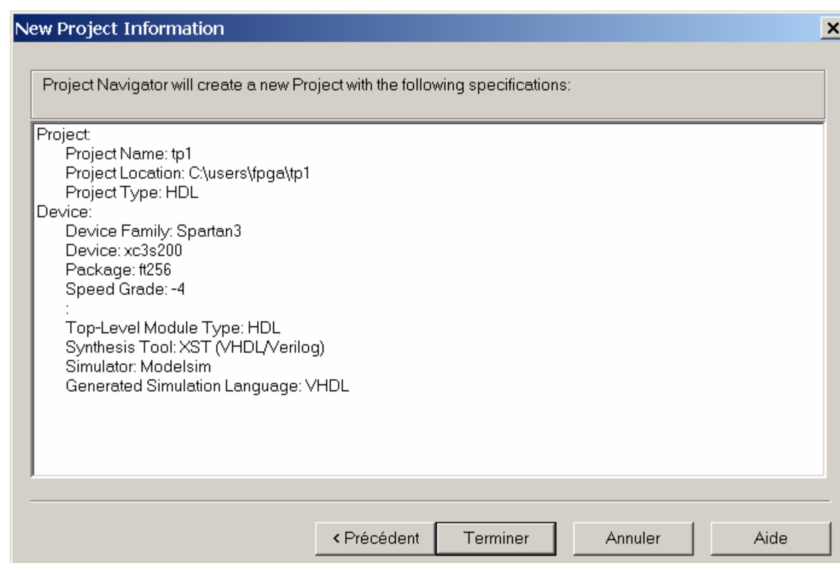
- la famille de FPGA utilisée (Spartan3 dans le champ « Device Family »),

- le circuit utilisé (xc3s200 dans le champ « Device »),
- le boîtier (ft256 dans le champ « Package »),
- la vitesse (-4 dans le champ « Speed Grade »),
- les outils du flot de développement (synthétiseur : XST, simulateur : modelsim, langage VHDL).

Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Suivant » :

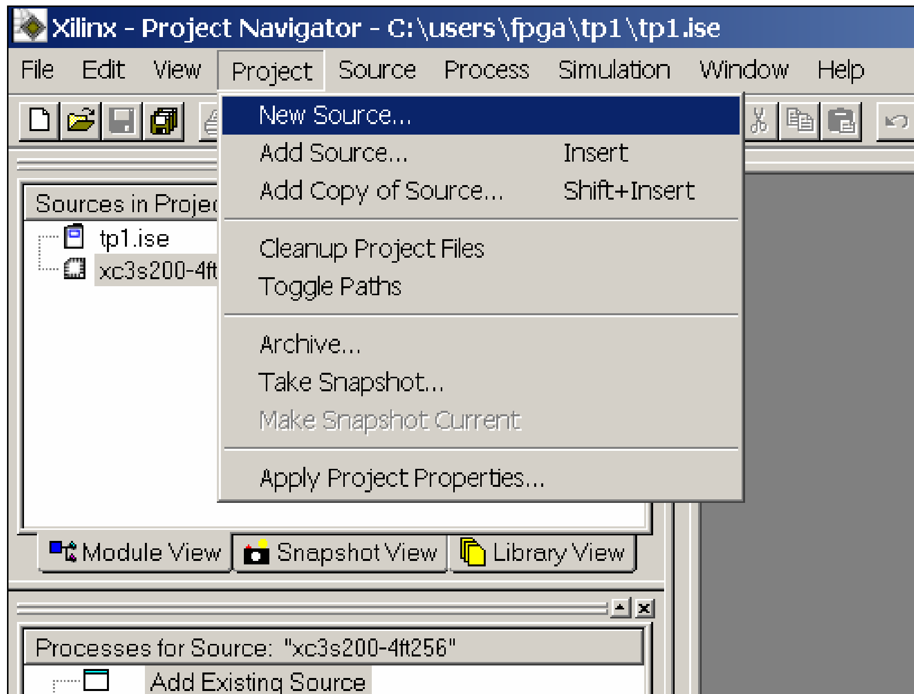


Cliquez sur « Suivant » dans les deux fenêtres suivantes, puis sur « Terminer » dans la fenêtre finale :

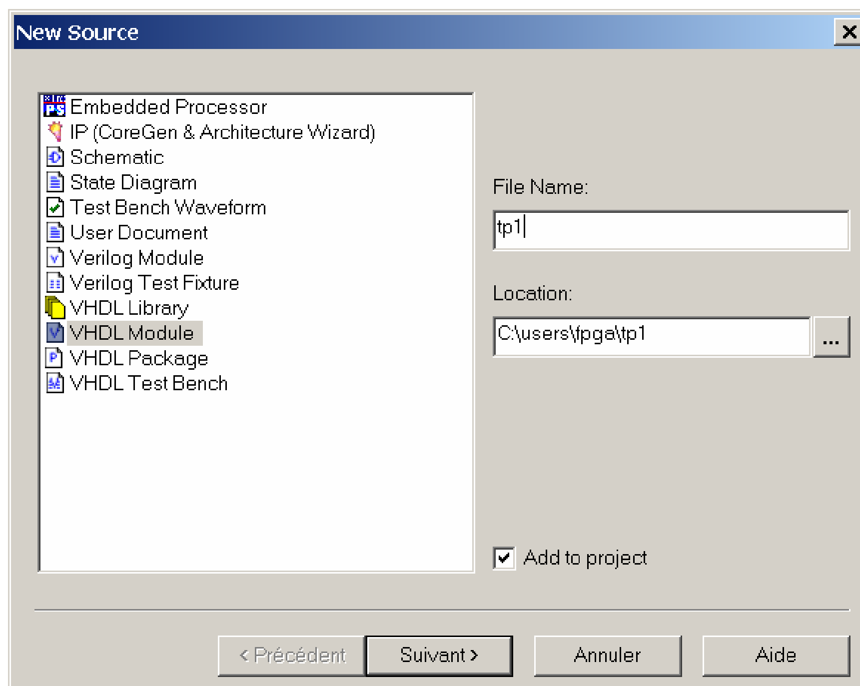


9.1.4 Création du design VHDL

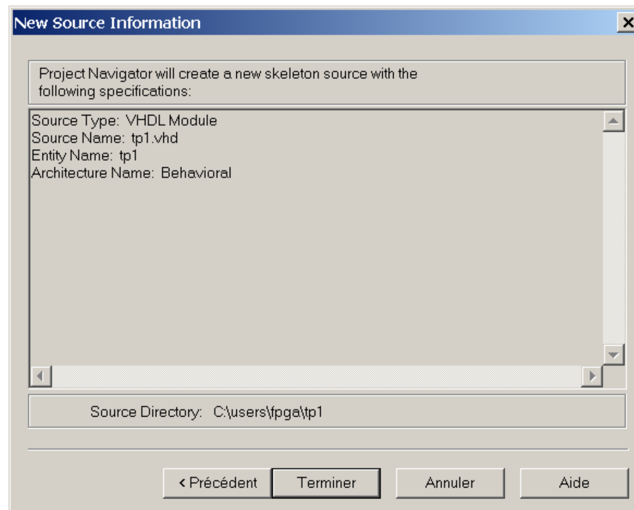
La saisie du design s'effectue à l'aide de l'éditeur intégré. Pour le lancer, cliquez sur le menu « Project » puis sur le sous-menu « New Source… ».



La fenêtre suivante s'ouvre. Sélectionnez « VHDL Module », tapez le nom du design tp1 dans le champ « File Name » puis cliquez sur « Suivant ».



Cliquez sur « Suivant » dans la fenêtre suivante, puis sur « Terminer » dans la fenêtre d'information :



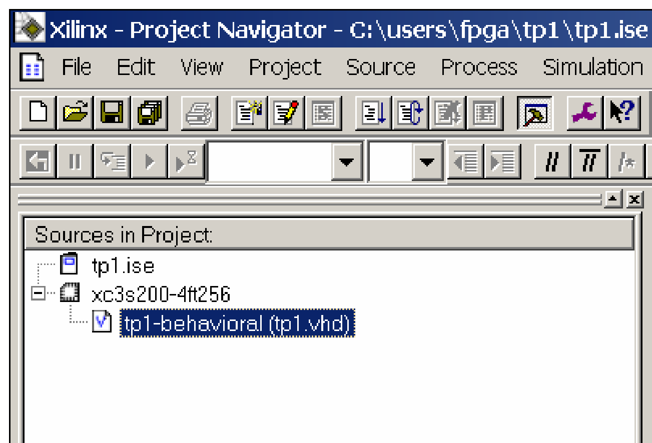
ISE crée automatiquement un fichier VHDL qui contient l'appel de certaines librairies et la déclaration de l'entité et de l'architecture. Ce fichier est ouvert automatiquement dès sa création :

```
1  --------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    16:58:44 09/25/06
6  -- Design Name:
7  -- Module Name:    tp1 - Behavioral
8  -- Project Name:
9  -- Target Device:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 --------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_ARITH.ALL;
23 use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25 ---- Uncomment the following library declaration if instantiating
26 ---- any Xilinx primitives in this code.
27 --library UNISIM;
28 --use UNISIM.VComponents.all;
29
30 entity tp1 is
31 end tp1;
32
33 architecture Behavioral of tp1 is
34
35 begin
36
37
38 end Behavioral;
39
```

Il reste à le compléter pour obtenir :

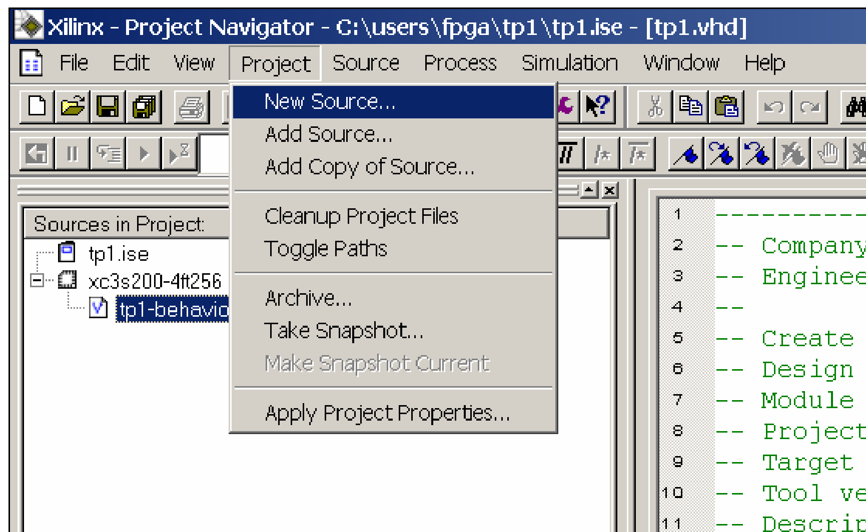
```vhdl
-- Module Name:    tp1 - Behavioral
-- Project Name:
-- Target Device:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tp1 is
  port(D1, D2, D3 : in  std_logic;
       Y1, Y2, Y3, Y4, Y5 : out std_logic);
end tp1;

architecture Behavioral of tp1 is
   signal tmp : std_logic;
begin
  Y1 <= D1 nor D2;
  Y2 <= not (D1 or D2 or D3);
  Y3 <= D1 and D2 and not D3;
  Y4 <= D1 xor (D2 xor D3);
  tmp <= D1 xor D2;
  Y5 <= tmp nand D3;
end Behavioral;
```

Sauvez votre design en cliquant sur [icône] dans la barre d'outil. Vous devez normalement obtenir la fenêtre suivante dans les sources du navigateur de projet.

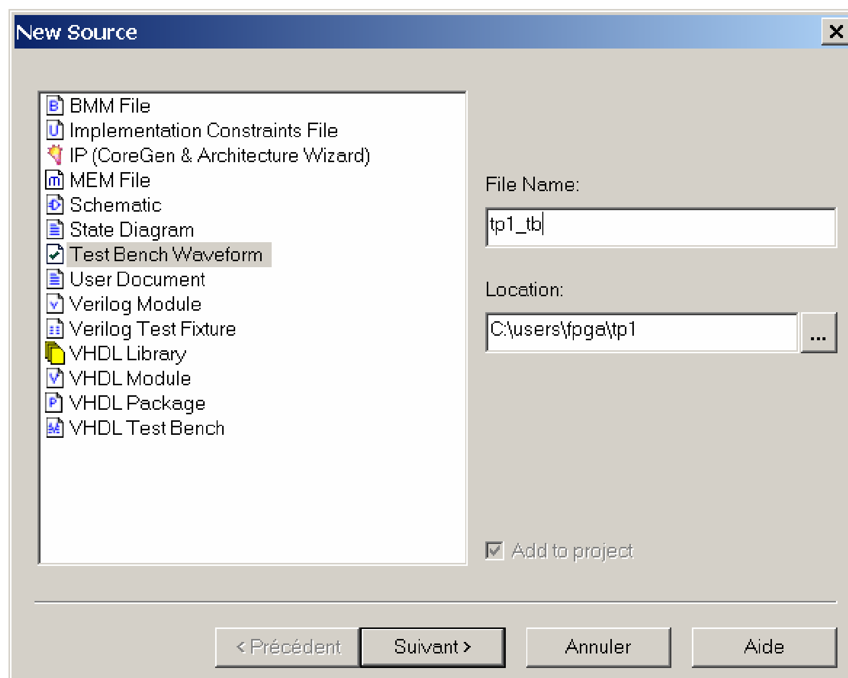


9.1.5 Génération du fichier de stimuli VHDL

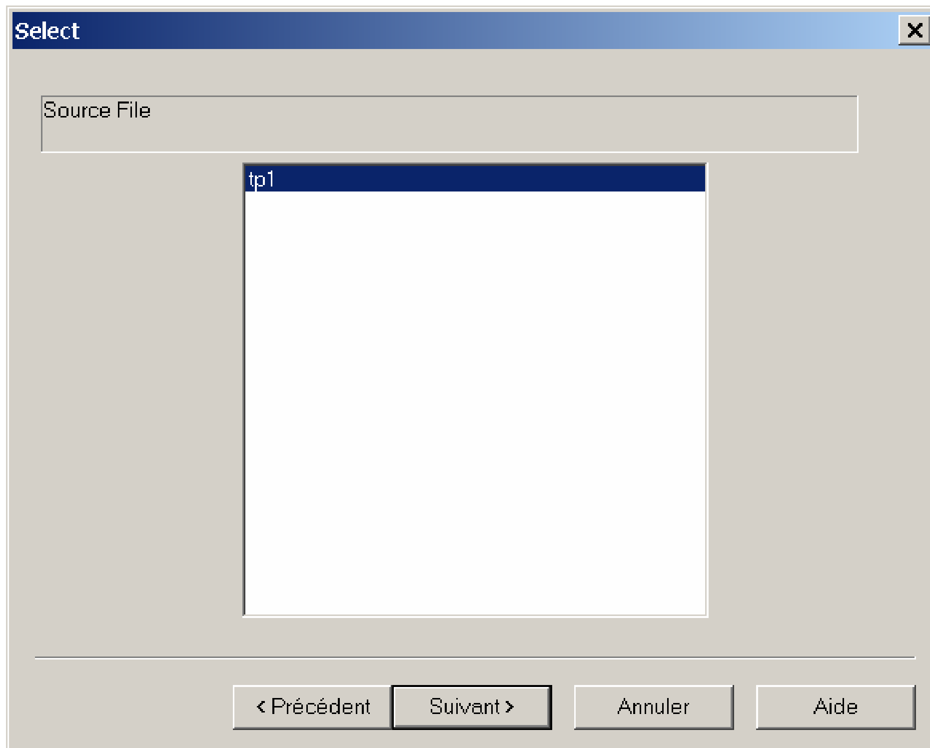
Nous avons maintenant un design prêt à être simulé et nous devons écrire un fichier en langage VHDL décrivant les signaux d'entrées. Ce fichier s'appelle un fichier de stimuli (un testbench en VHDL) qui contient des vecteurs de test. Il y a, parmi les outils Xilinx, un outil graphique pour définir les stimuli : Waveform Editor. Pour lancer cette application, cliquez sur le menu « Project » puis sur le sous-menu « New Source… ».



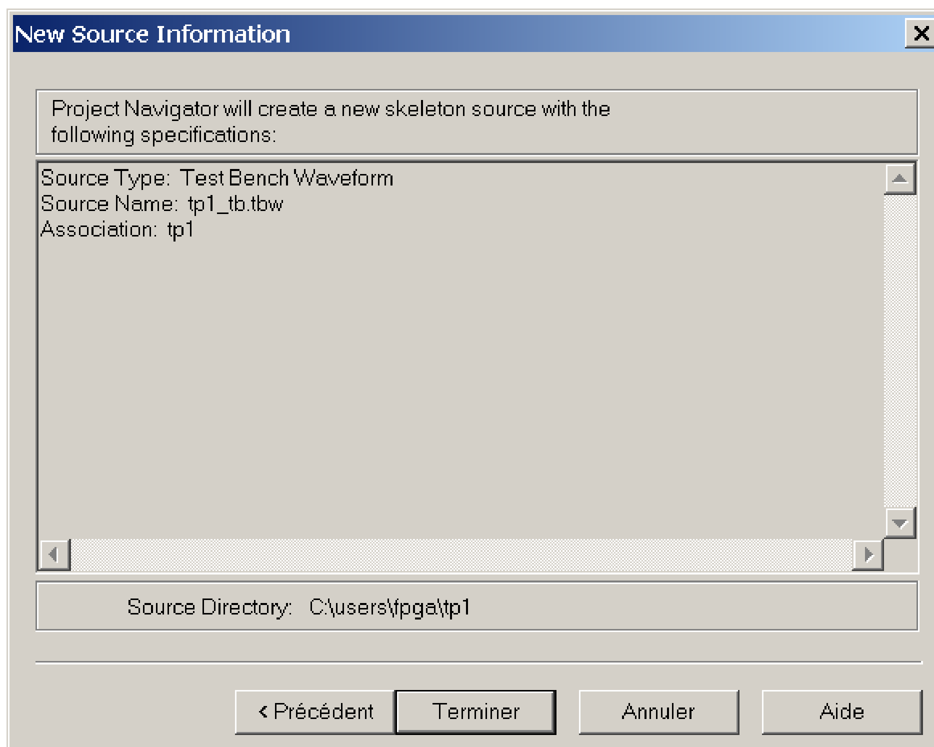
La fenêtre suivante apparaît alors à l'écran. Sélectionner le type « Testbench waveform », tapez tp1_tb comme nom de fichier pour notre testbench puis cliquez sur « Suivant » :



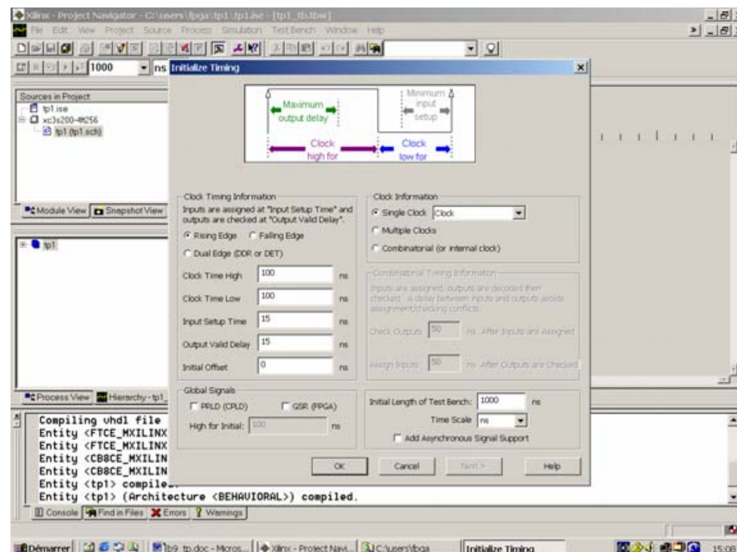
Cliquez sur suivant pour associer le testbench avec le design tp1 :



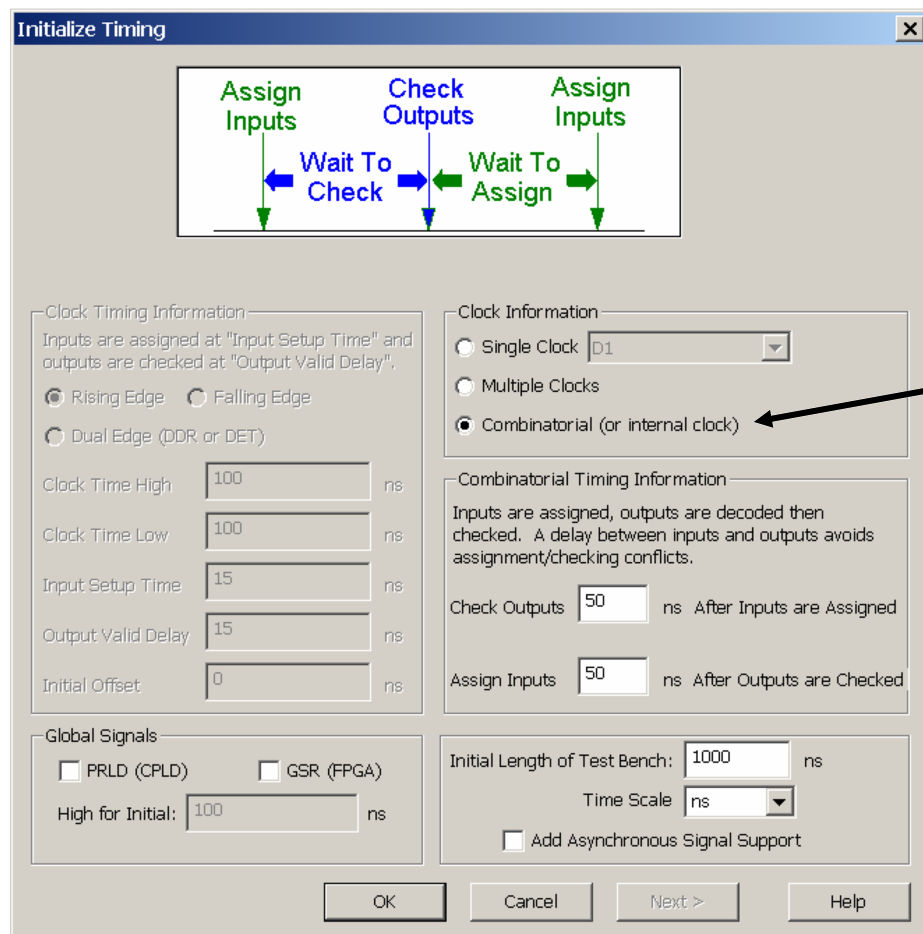
puis sur terminer pour lancer l'éditeur de stimuli :



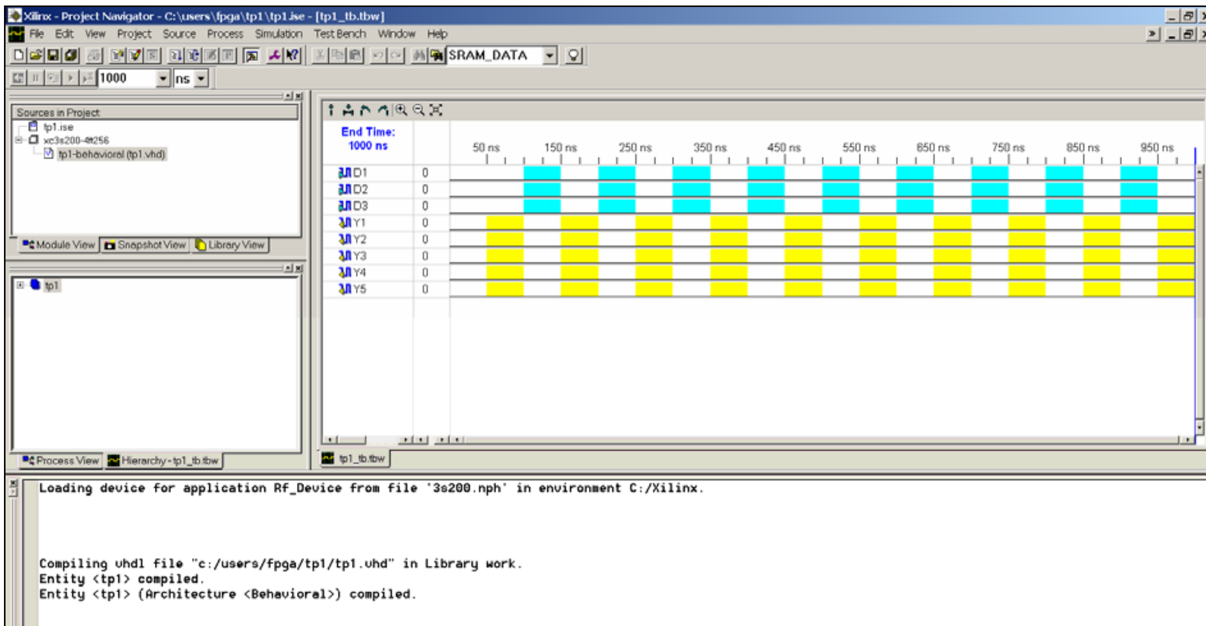
L'application démarre.



Notre design est purement combinatoire et ne possède donc pas d'horloge. Remplissez les différents champs comme sur la fenêtre suivante avant de cliquez sur « OK » :

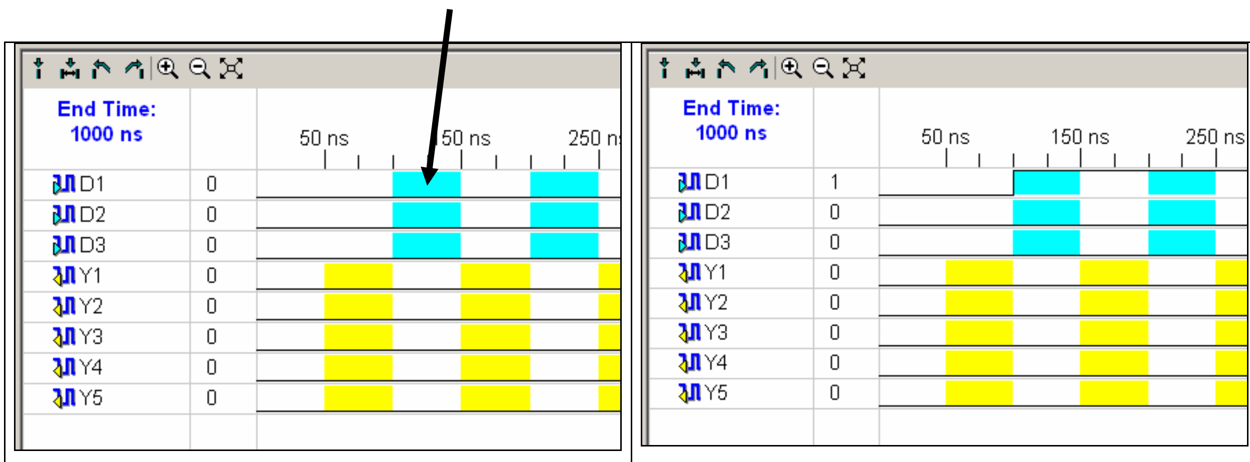


Un chronogramme apparaît dans la fenêtre de droite du navigateur de projet :

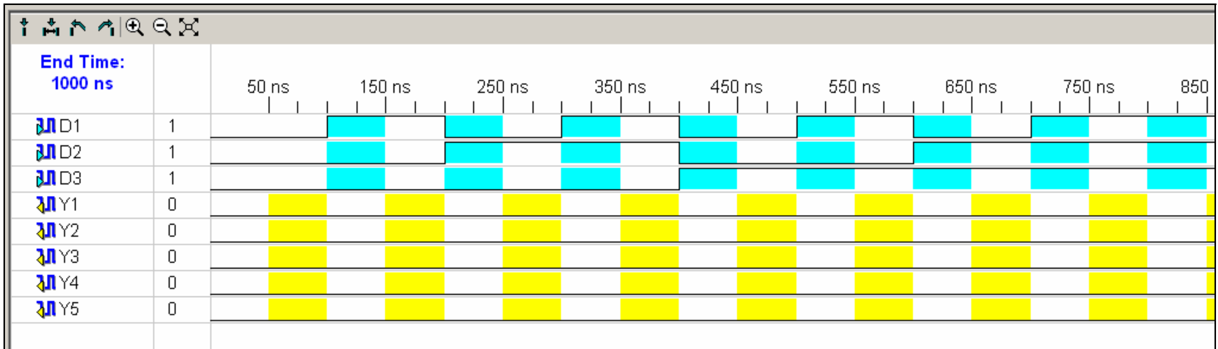


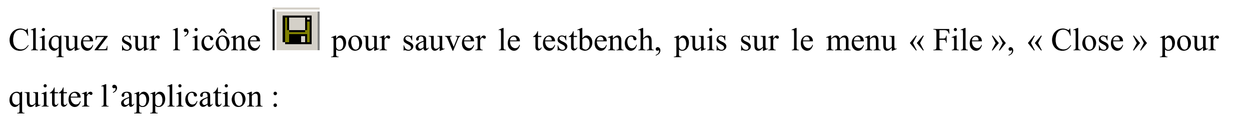
Nous pouvons maintenant spécifier les stimuli sous forme graphique.

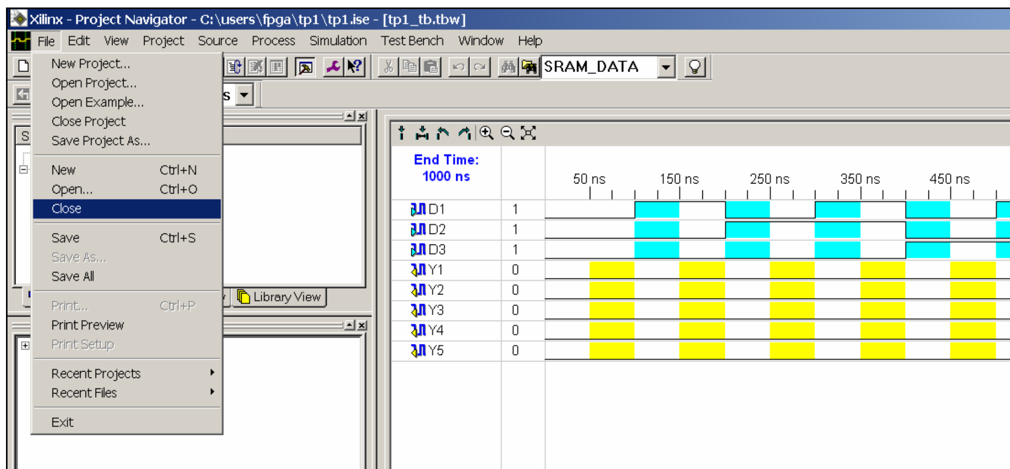
Vous devez voir vos trois entrées (D1, D2 et D3) et 5 sorties (Y1 à Y5). Si tel n'est pas le cas, vous avez sûrement oublié un signal dans l'entité. Fermez la fenêtre sans la sauvegarder puis, dans le navigateur, sélectionnez tp1, cliquez avec le bouton droit de la souris puis sur « Open ». La fenêtre de l'éditeur s'ouvre à nouveau. Faites les modifications nécessaires, sauvez votre fichier puis relancez Waveform Editor. Renouvelez ces opérations jusqu'à ce que vous ayez les bonnes entrées-sorties. Une fois la bonne fenêtre obtenue, cliquez sur la première zone bleue de D1 pour le faire passer à 1 :



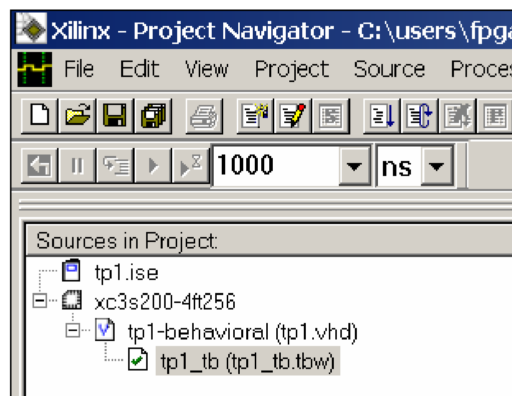
Cliquez sur les autres zones bleues jusqu'à obtenir les chronogrammes suivants :



Cliquez sur l'icône  pour sauver le testbench, puis sur le menu « File », « Close » pour quitter l'application :



Le testbench apparaît maintenant dans les sources du navigateur de projet. Vous pouvez à tout moment relancer Waveform Editor en double cliquant sur tp1_tb.tbw.

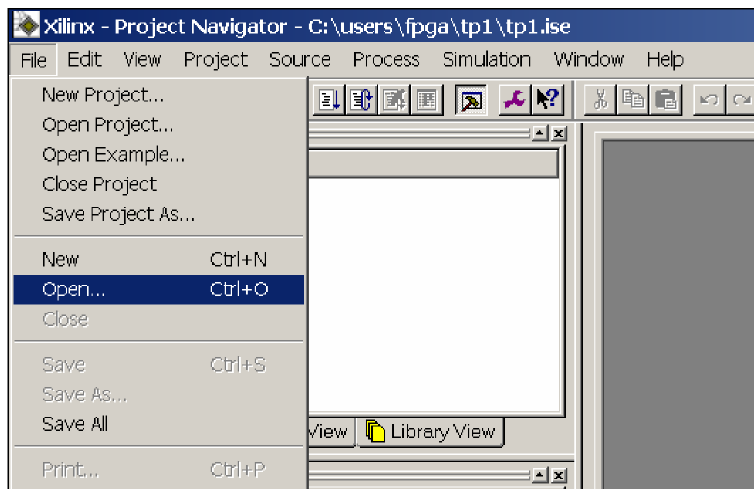


9.1.6 Simulation fonctionnelle

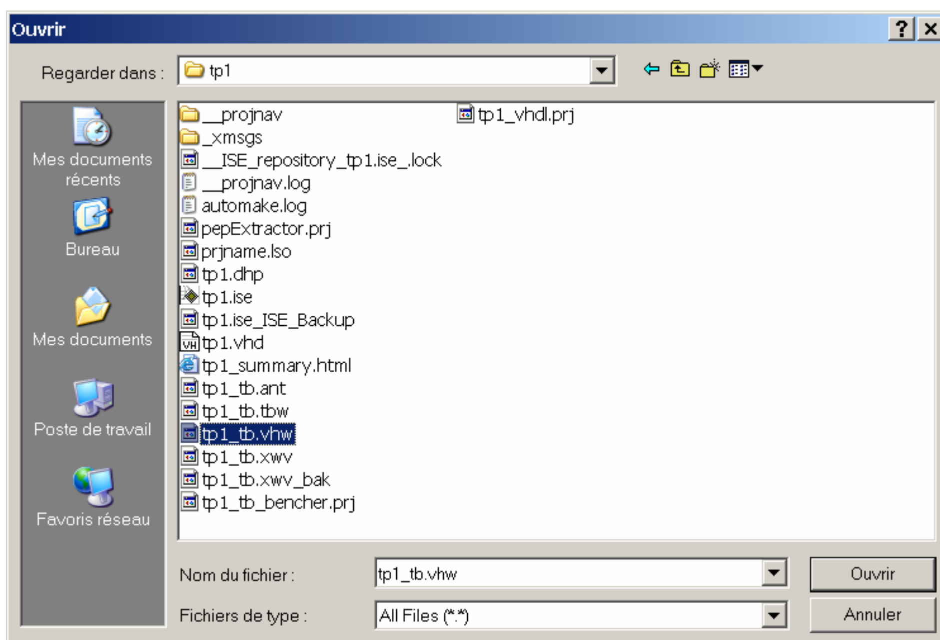
Nous sommes maintenant en possession de deux fichiers :

1. un fichier VHDL représentant le design à réaliser (tp1.vhd).
2. un fichier VHDL représentant les signaux que l'on souhaite appliquer sur ses entrées (tp1_tb.vhw). Ce fichier, généré par Waveform Editor, est la représentation en VHDL des stimuli créés précédemment sous forme graphique.

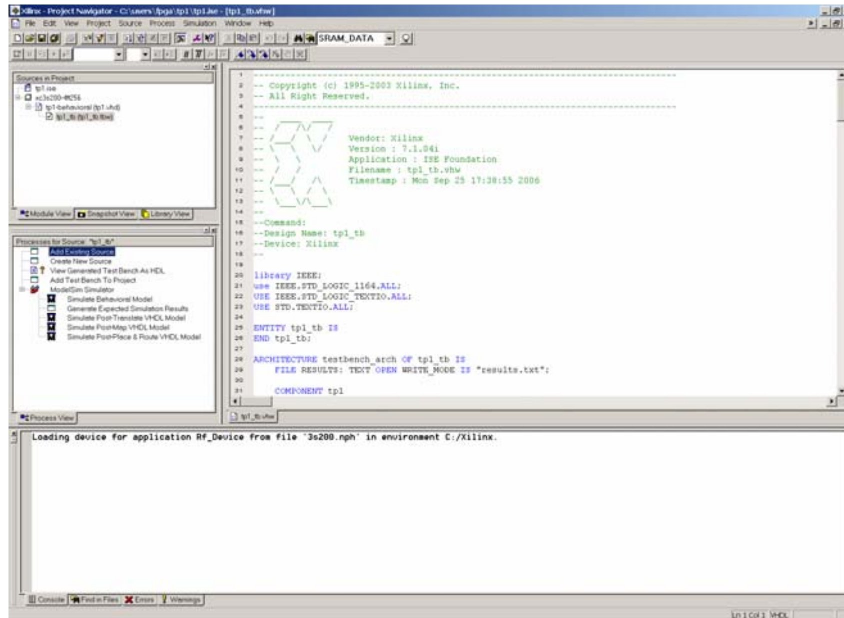
Vous pouvez visualiser ce fichier à l'aide du navigateur de projet en cliquant sur le menu File puis sur le sous-menu Open :



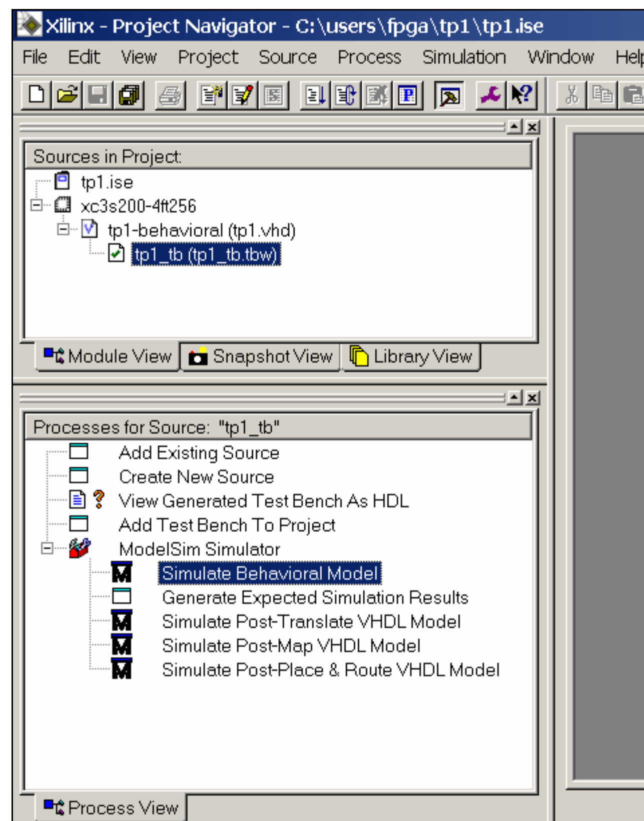
Sélectionnez le type de fichier « All Files » puis le fichier tp1_tb.vhw. Cliquez alors sur Ouvrir.



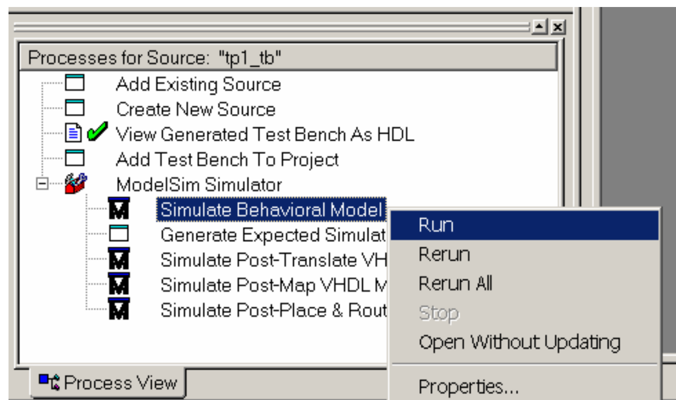
Le contenu du fichier apparaît dans la fenêtre d'édition du navigateur :



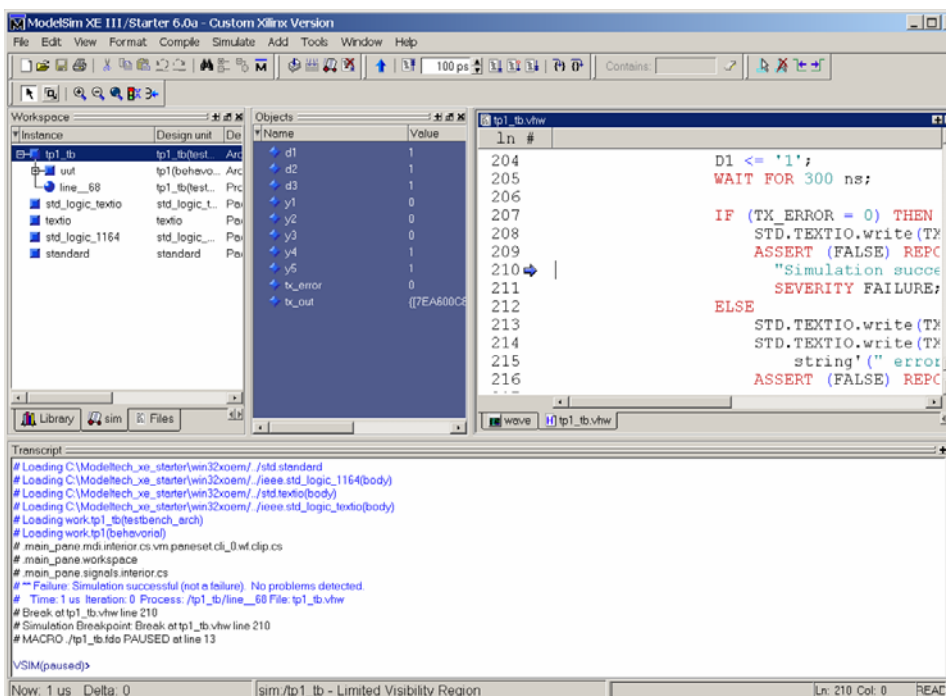
Fermez la fenêtre d'édition. Nous allons maintenant simuler le design à l'aide du simulateur VHDL ModelSim. Pour cela, sélectionnez le fichier tp1_tb.tbw dans la fenêtre « Sources In Project ». Les actions qui peuvent être effectuées avec ce fichier apparaissent dans la fenêtre inférieure « Processes for Source ».



Sélectionnez le process « Simulate Behavorial Model » puis cliquez avec le bouton droit de la souris sur le menu Run :



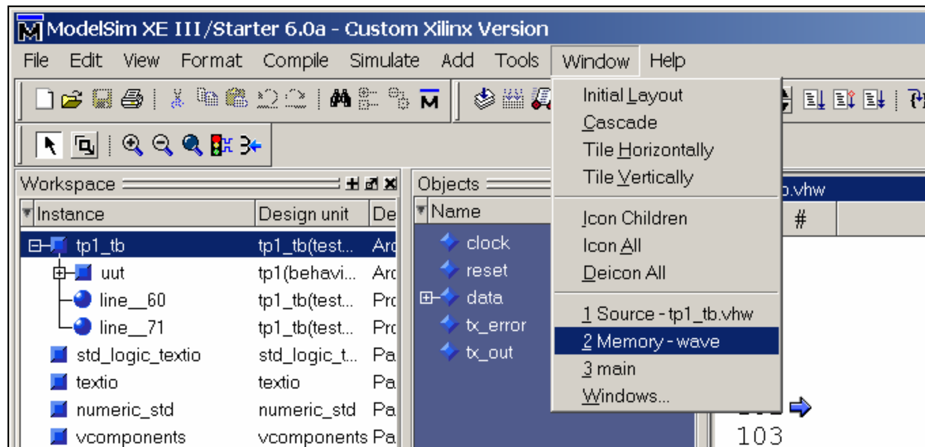
Le simulateur VHDL ModelSim démarre et la fenêtre suivante s'ouvre à l'écran. Si tel n'est pas le cas, c'est parce que ModelSim s'exécute sous le navigateur de projet. Il faut alors cliquer sur son icône dans la barre de tache de Windows pour le mettre au premier plan.

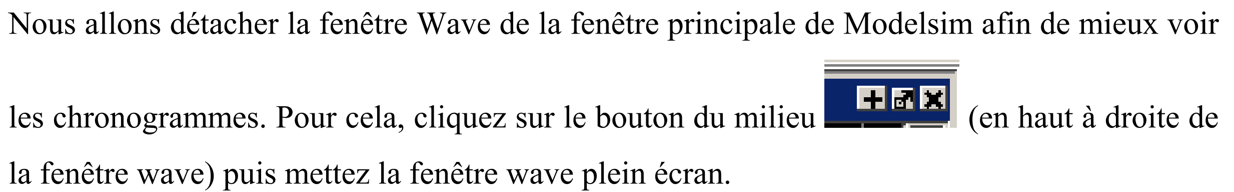


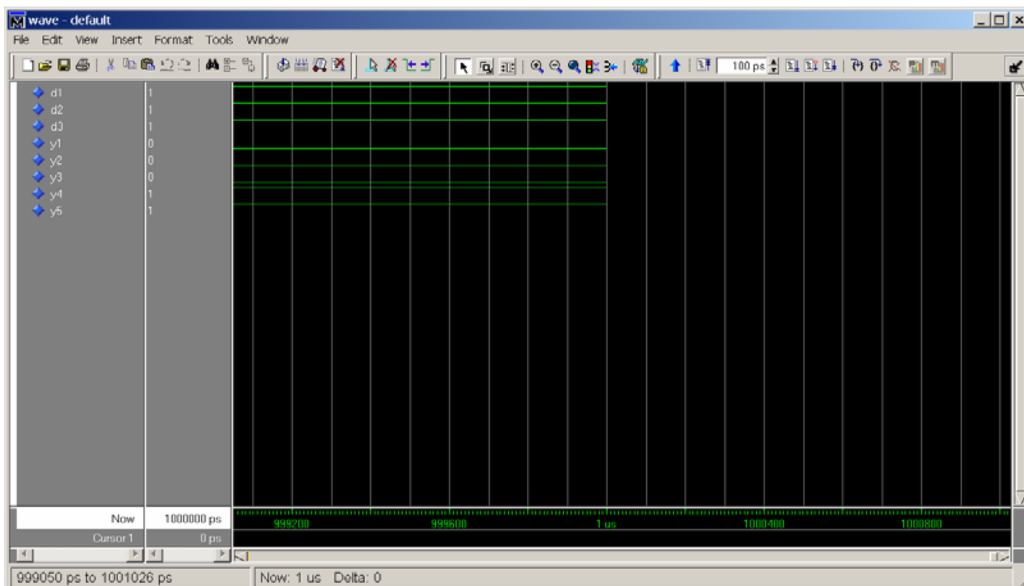
ModelSim va enchaîner automatiquement :

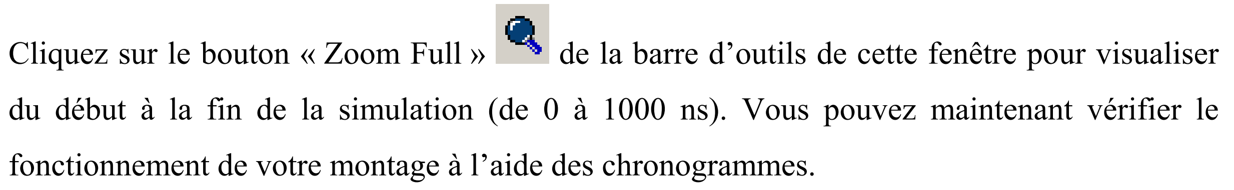
- la compilation du design et des stimuli,
- le lancement du simulateur,
- la simulation jusqu'à l'arrêt automatique du testbench.

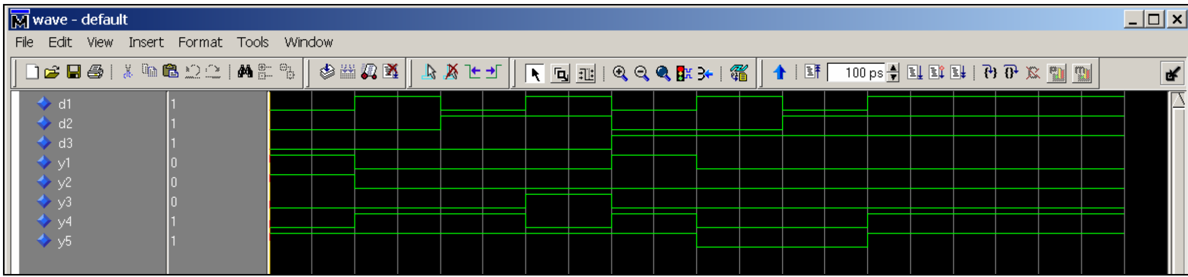
Sélectionnez la fenêtre Wave (là où se trouvent les chronogrammes) en cliquant sur le menu Windows, Memory-wave :



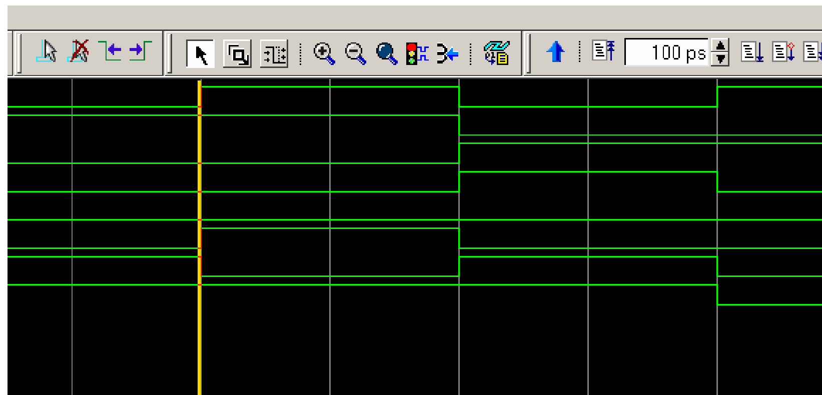
Nous allons détacher la fenêtre Wave de la fenêtre principale de Modelsim afin de mieux voir les chronogrammes. Pour cela, cliquez sur le bouton du milieu  (en haut à droite de la fenêtre wave) puis mettez la fenêtre wave plein écran.

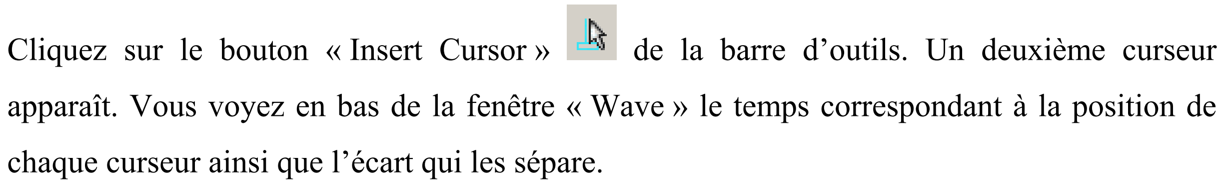


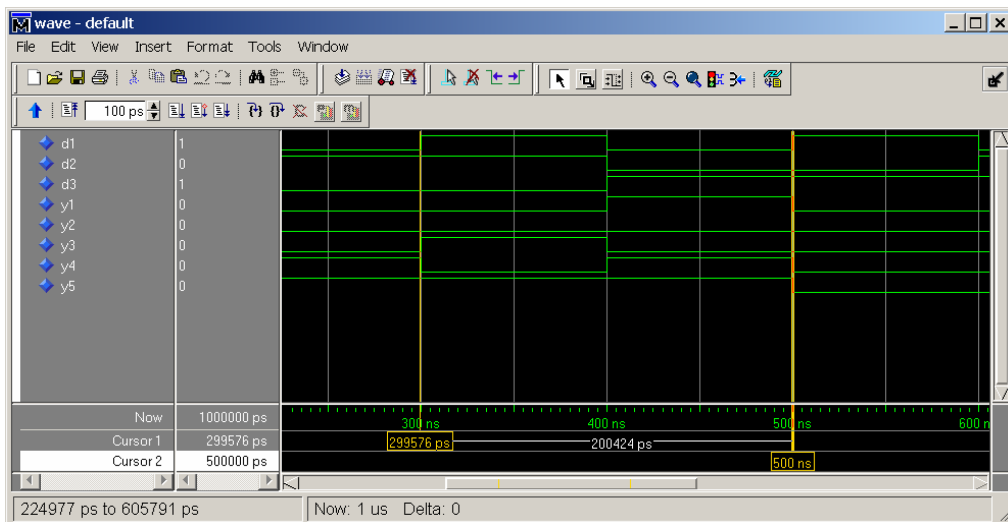
Cliquez sur le bouton « Zoom Full »  de la barre d'outils de cette fenêtre pour visualiser du début à la fin de la simulation (de 0 à 1000 ns). Vous pouvez maintenant vérifier le fonctionnement de votre montage à l'aide des chronogrammes.



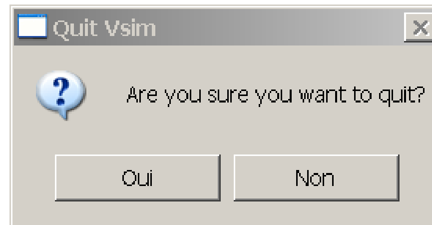
Faîtes un zoom sur le chronogramme en cliquant en haut et à gauche de la zone à agrandir avec le bouton du milieu de la souris. Maintenez cliqué et déplacez la souris en bas et à droite de la zone (un rectangle bleu apparaît). Relâchez le bouton du milieu. Le zoom s'exécute. Cliquez n'importe où sur le chronogramme. Un curseur jaune apparaît :



Cliquez sur le bouton « Insert Cursor »  de la barre d'outils. Un deuxième curseur apparaît. Vous voyez en bas de la fenêtre « Wave » le temps correspondant à la position de chaque curseur ainsi que l'écart qui les sépare.

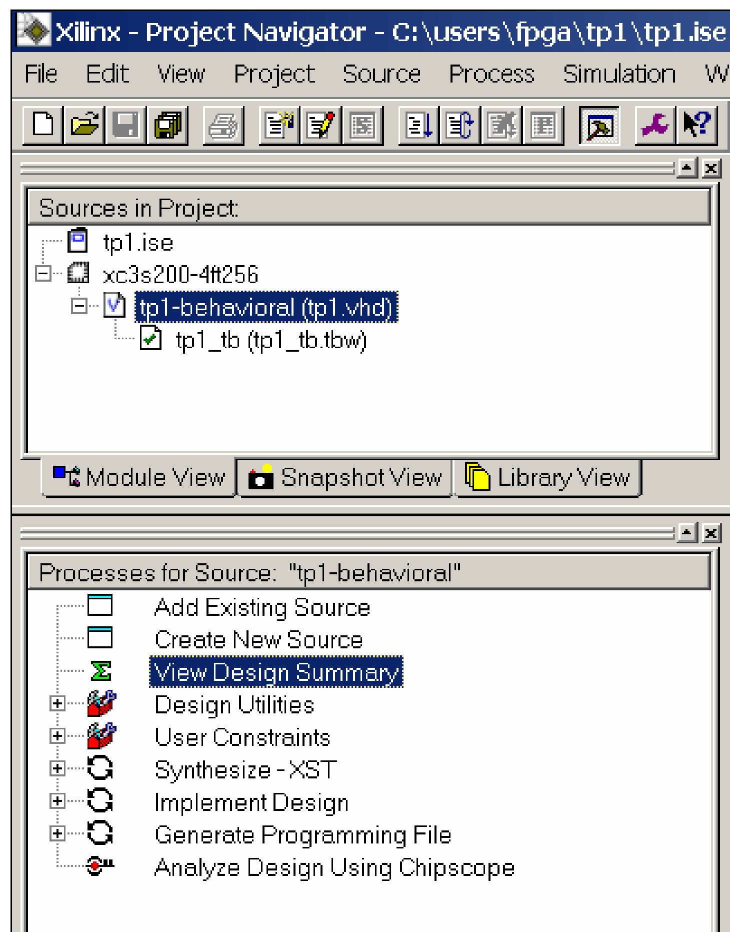


Quand la vérification est terminée, faites apparaître la fenêtre de ModelSim, puis cliquez sur le menu « File » dans la fenêtre principale, puis sur « Quit ». Quand le message suivant apparaît à l'écran, cliquez sur Oui.

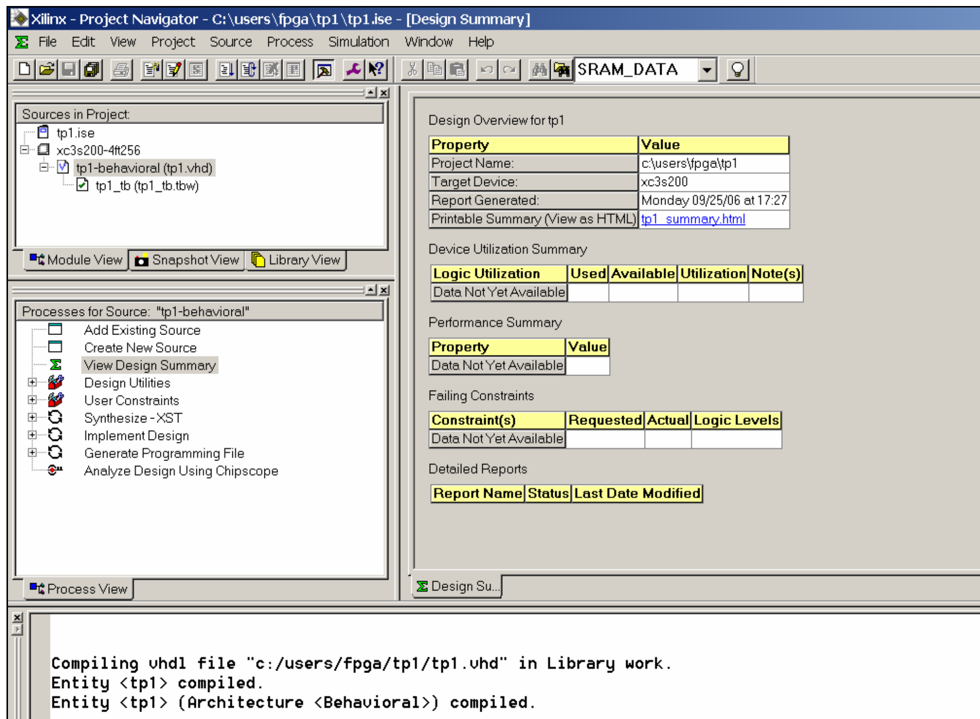


9.1.7 Synthèse

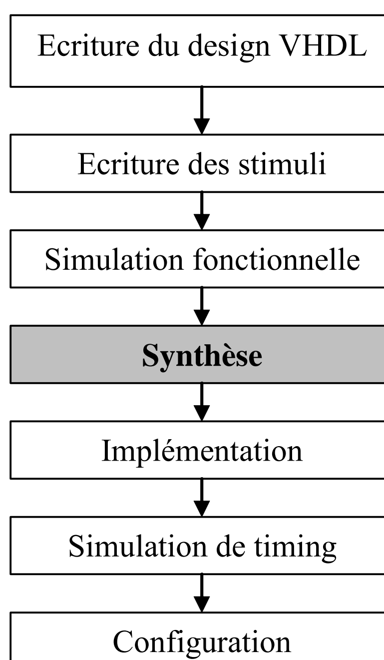
Nous avons fini la première phase de création et de vérification du design. Vous pouvez voir à tout moment le résumé des différentes étapes du design en sélectionnant tp1, puis en double-cliquant sur « View Design Summary » :



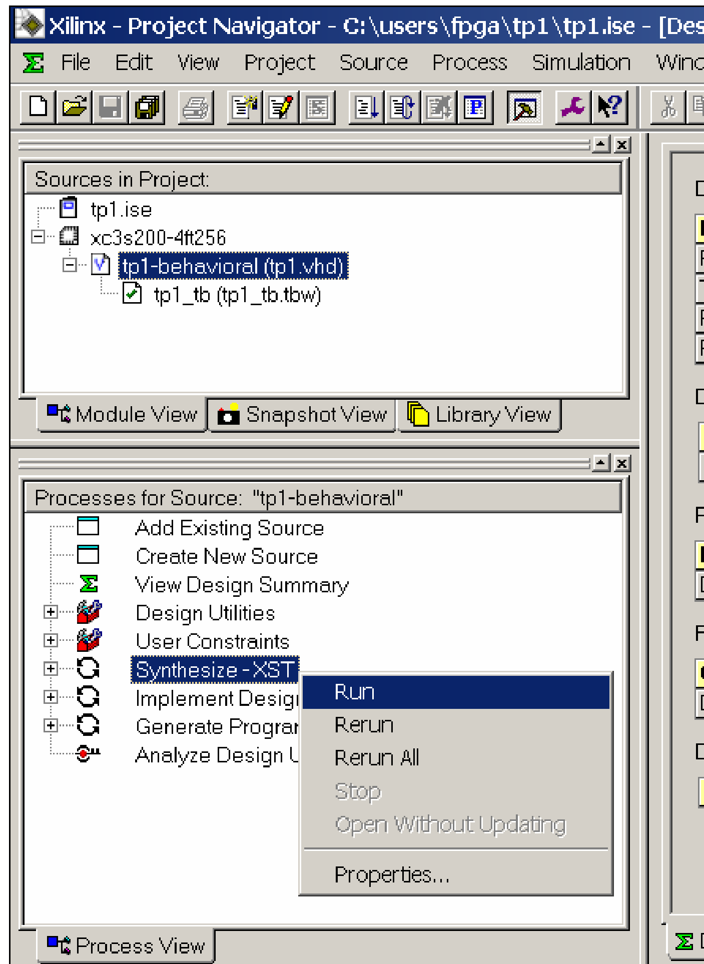
La page de résumé s'affiche à droite du navigateur de projet :



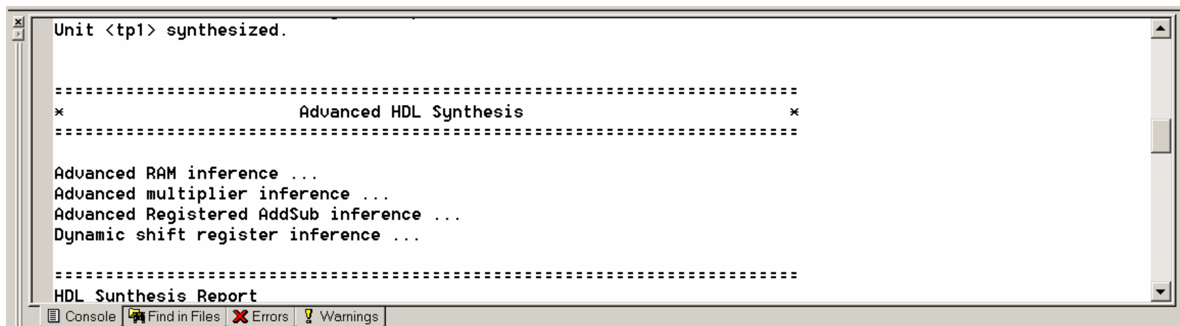
Elle contient pour l'instant peu d'informations, mais elle va s'enrichir au fur et à mesure de l'avancement du projet. La description VHDL tp1.vhd est assez générale et ne prend en compte aucun circuit cible en particulier. Le rôle du synthétiseur XST est de comprendre et d'interpréter cette description générale afin de générer un fichier NGC compréhensible par les outils d'implémentation, c'est-à-dire une netlist NGC composée de primitives simples. Cette netlist est spécifique à notre FPGA.



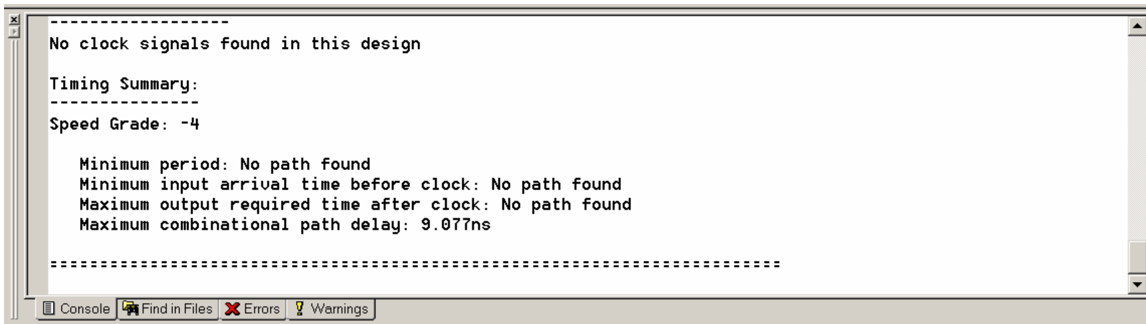
Sélectionnez le design tp1.vhd dans la fenêtre « Sources » puis « Synthesize » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



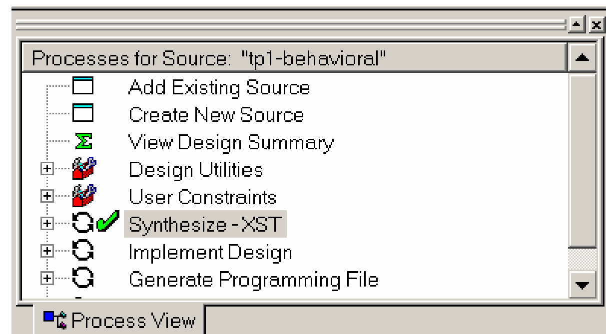
La synthèse démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant son déroulement apparaît :



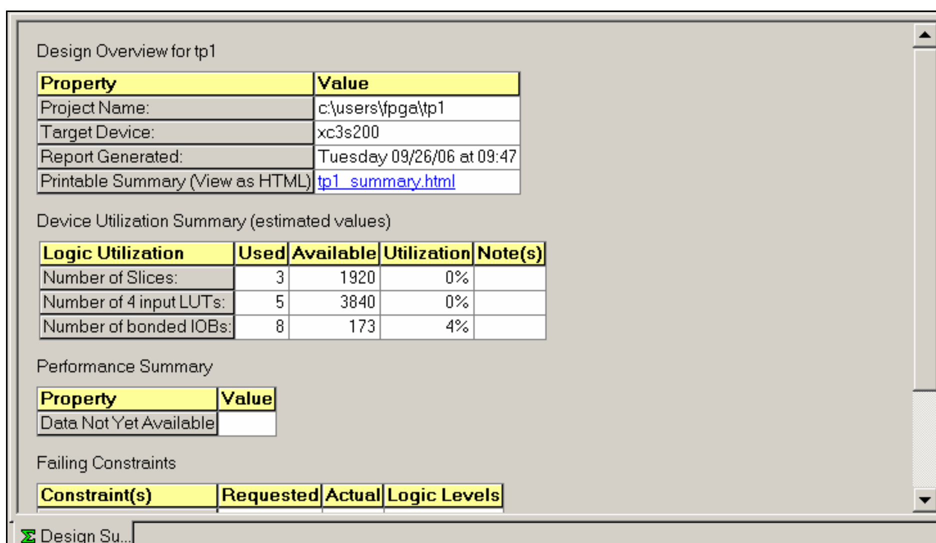
Lorsque la synthèse est finie, vous devez voir une estimation des timings du design, ce qui indique que la synthèse s'est bien terminée.



Une marque verte apparaît dans la fenêtre Processes. Elle indique que la synthèse s'est bien déroulée. Un point d'exclamation jaune signalerait un message d'avertissement (un Warning) et une croix rouge indiquerait une erreur. Tous ces messages apparaissent dans le rapport de synthèse.



Vous ne pouvez pas visualiser le fichier NGC issu de la synthèse car il s'agit d'un format de netlist binaire qui ne peut donc être visualisé avec un éditeur ASCII. La fenêtre de résumé a évoluée et reflète maintenant les informations de synthèse :



Property	Value
Project Name:	c:\users\fpga\tp1
Target Device:	xc3s200
Report Generated:	Tuesday 09/26/06 at 09:47
Printable Summary (View as HTML)	tp1_summary.html

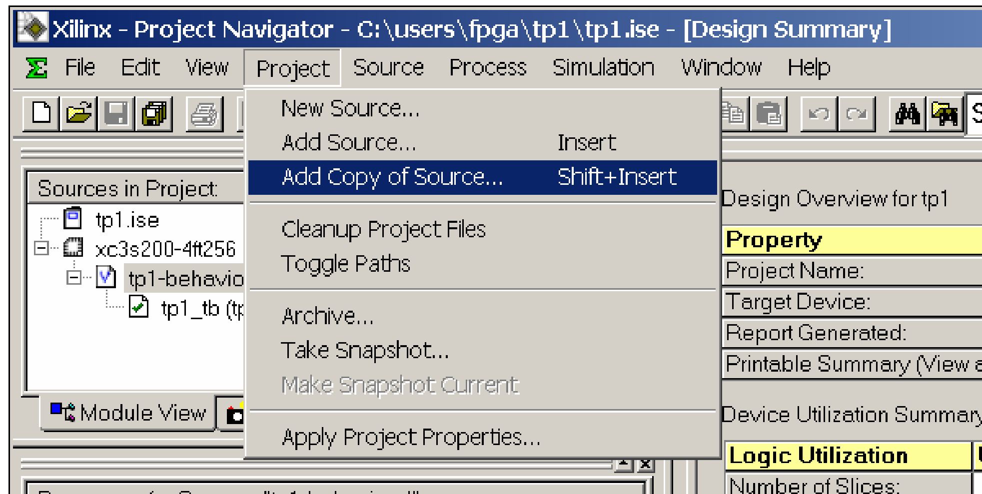
Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	3	1920	0%	
Number of 4 input LUTs:	5	3840	0%	
Number of bonded IOBs:	8	173	4%	

Property	Value
Data Not Yet Available	

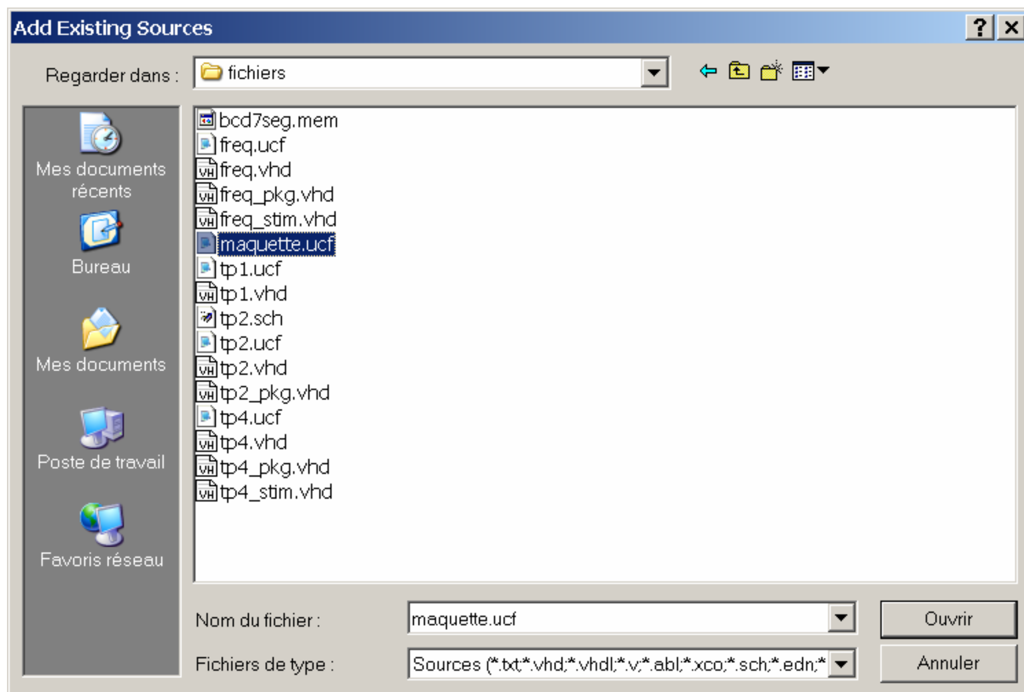
Constraint(s)	Requested	Actual	Logic Levels

9.1.8 Implémentation

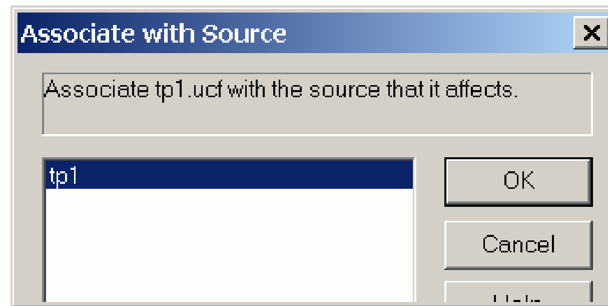
Nous pouvons maintenant travailler sur le circuit FPGA lui-même. C'est le rôle des outils d'implémentation. Nous allons commencer par affecter les broches D1 à D3 et Y1 à Y5 aux broches du FPGA selon le câblage de la maquette. Vous utiliserez pour cela un fichier de contraintes utilisateur déjà écrit qui se nomme maquette.ucf (User Constraint File). Pour l'ajouter au projet tp1, sélectionnez le menu Project, Add Copy of Source :



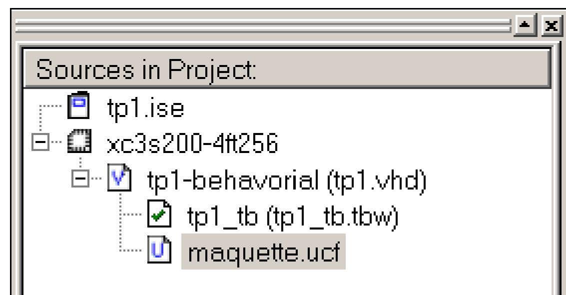
Dans la fenêtre qui s'ouvre, sélectionnez le répertoire c:\users\fpga\fichiers, cliquez sur le fichier maquette.ucf puis sur « Ouvrir » :



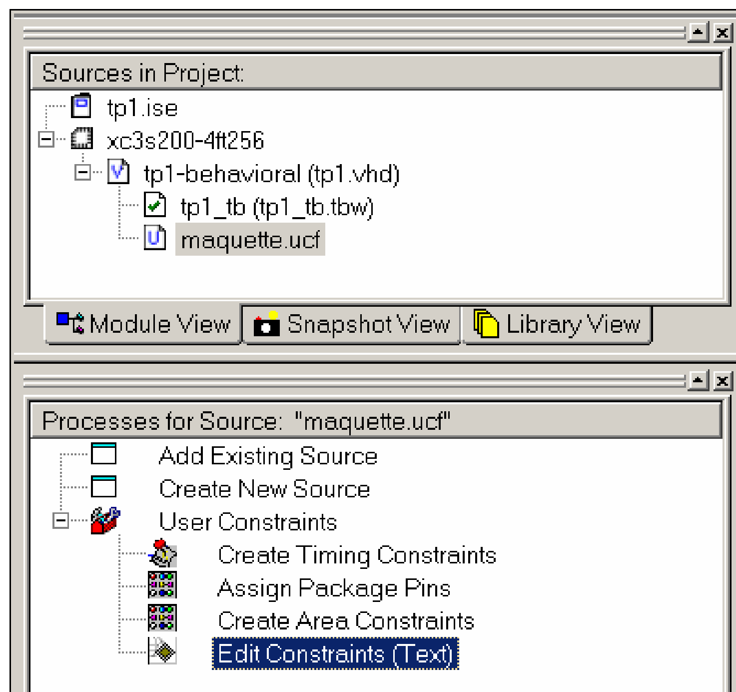
Cliquez sur le bouton « OK » dans la petite fenêtre qui s'ouvre alors :



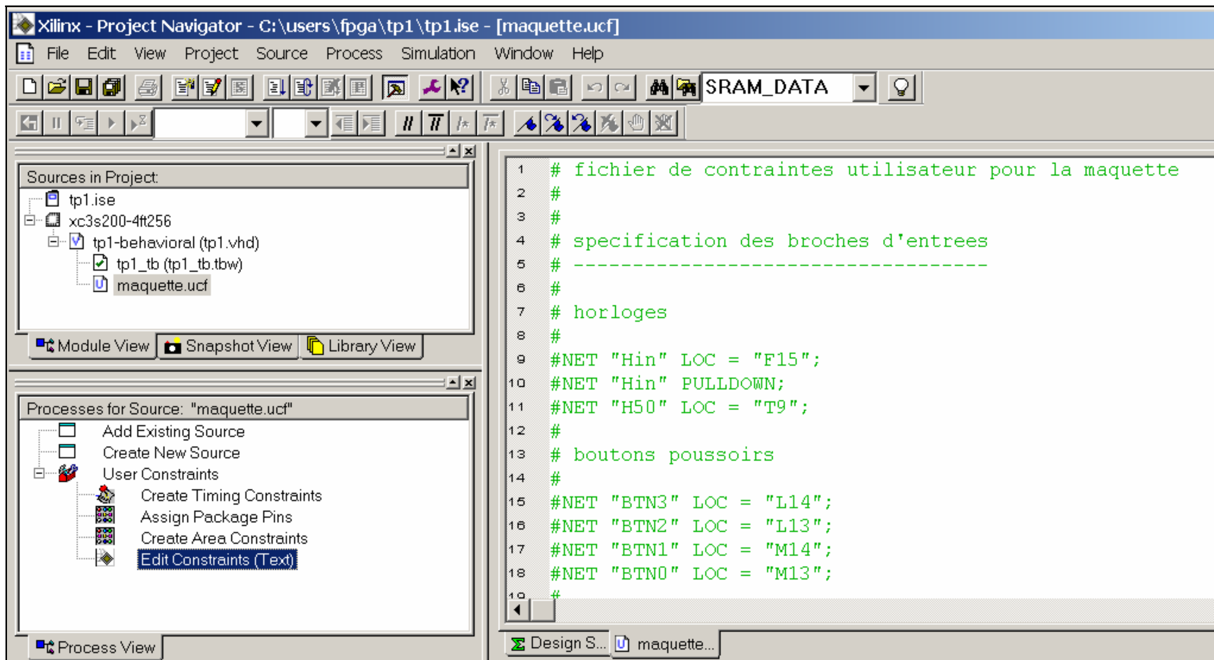
Le fichier est maintenant inclus dans le projet :



Pour voir le contenu de ce fichier, il suffit de le sélectionner dans la fenêtre « Sources », puis de double-cliquer dans la fenêtre Processes sur « Edit Constraints (Text) » :



Le contenu du fichier apparaît alors dans l'éditeur :



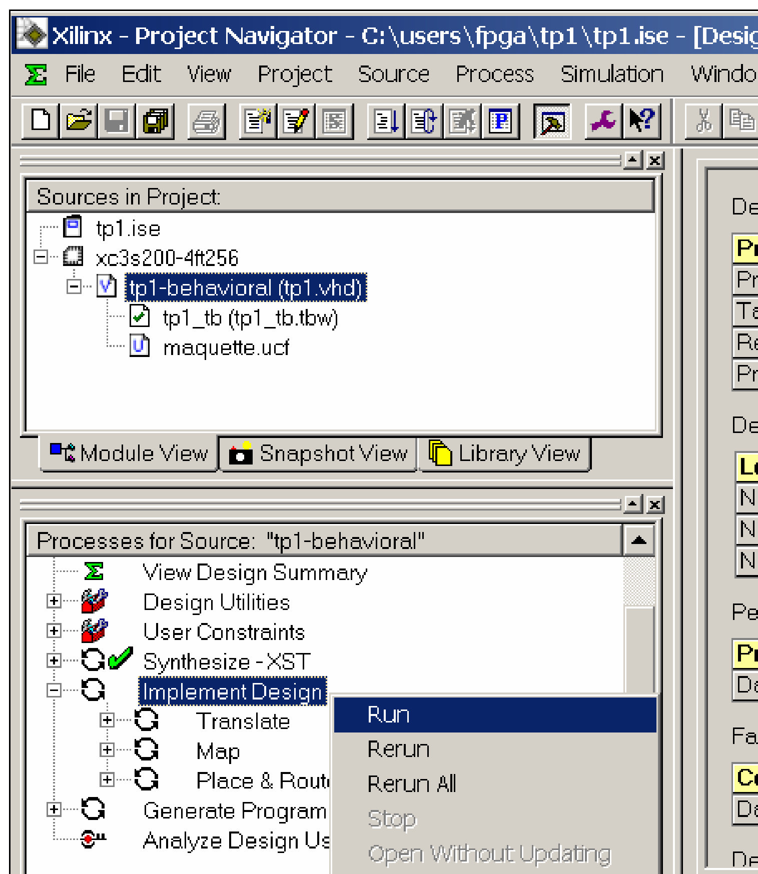
Toutes les lignes de ce fichier sont commentées avec un #. Nous allons affecter D1, D2 et D3 sur les dip switch SW0, SW1 et SW2. Pour cela, il suffit de décommenter les 3 lignes et de changer les noms des signaux :

```
17  #NET "BTN1" LOC = "M14";
18  #NET "BTN0" LOC = "M13";
19  #
20  # interrupteurs DIP
21  #
22  #NET "SW7" LOC = "K13";
23  #NET "SW6" LOC = "K14";
24  #NET "SW5" LOC = "J13";
25  #NET "SW4" LOC = "J14";
26  #NET "SW3" LOC = "H13";
27  NET "D3" LOC = "H14";
28  NET "D2" LOC = "G12";
29  NET "D1" LOC = "F12";
30  #
31  # specification des broches de sorties
32  # ------------------------------------
33  #
```

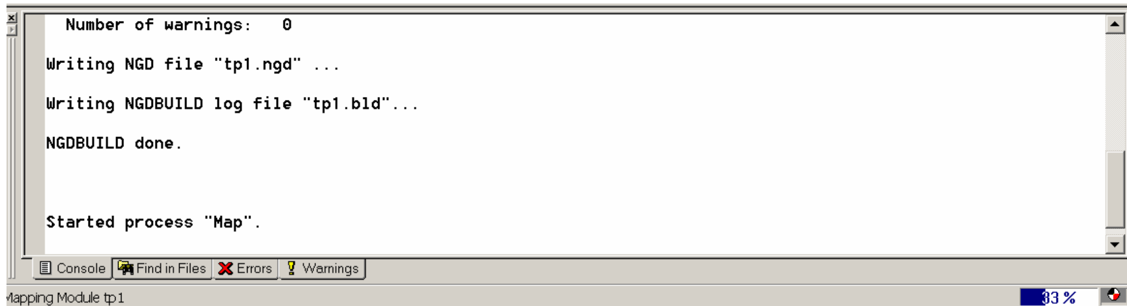
Nous allons ensuite affecter Y1 à Y5 sur les leds led0 à led4. Pour cela, il suffit de décommenter les 5 lignes et de changer les noms des signaux :

```
27  NET "D3" LOC = "H14";
28  NET "D2" LOC = "G12";
29  NET "D1" LOC = "F12";
30  #
31  # specification des broches de sorties
32  # ------------------------------------
33  #
34  # leds
35  #
36  #NET "Led7" LOC = "P11";
37  #NET "Led6" LOC = "P12";
38  #NET "Led5" LOC = "N12";
39  NET "Y5" LOC = "P13";
40  NET "Y4" LOC = "N14";
41  NET "Y3" LOC = "L12";
42  NET "Y2" LOC = "P14";
43  NET "Y1" LOC = "K12";
44  #
45  # afficheur 7 segments
```

Vous pouvez maintenant sauver le fichier puis fermer l'éditeur. Nous pouvons lancer l'implantation du FPGA (implementation en anglais). Pour cela, sélectionnez le design tp1.vhd dans la fenêtre « Sources » puis « Implement Design » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



L'implémentation démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant les différentes opérations apparaît :



Lorsque l'implémentation est finie, vous devez voir le message : PAR done ! qui indique que l'implémentation s'est bien terminée. Des points d'exclamation jaunes peuvent apparaître dans la fenêtre Processes. Ils indiquent que des messages d'avertissement (Warnings) ont été émis pendant l'implémentation. Vous pouvez voir ces Warnings en faisant défiler le rapport dans la Console à l'aide de la barre de défilement de droite. Il ne doit pas y en avoir dans notre exemple. Un certain nombre de rapports concernant les différentes étapes de l'implémentation sont maintenant disponibles dans la fenêtre Design Summary :

Printable Summary (View as HTML) tp1_summary.html

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs:	5	3,840	1%	
Logic Distribution:				
Number of occupied Slices:	3	1,920	1%	
Number of Slices containing only related logic:	3	3	100%	
Number of Slices containing unrelated logic:	0	3	0%	
Total Number of 4 input LUTs:	5	3,840	1%	
Number of bonded IOBs:	8	173	4%	

Performance Summary

Property	Value
Number of Unrouted Signals:	All signals are completely routed.
Number of Failing Constraints:	0

Failing Constraints

Constraint(s)	Requested	Actual	Logic Levels
No Constraints Found			

Detailed Reports

Report Name	Status	Last Date Modified
Synthesis Report	Current	Tuesday 09/26/06 at 09:47
Translation Report	Current	Tuesday 09/26/06 at 10:20
Map Report	Current	Tuesday 09/26/06 at 10:20
Pad Report	Current	Tuesday 09/26/06 at 10:20
Place and Route Report	Current	Tuesday 09/26/06 at 10:20
Post Place and Route Static Timing Report	Current	Tuesday 09/26/06 at 10:20

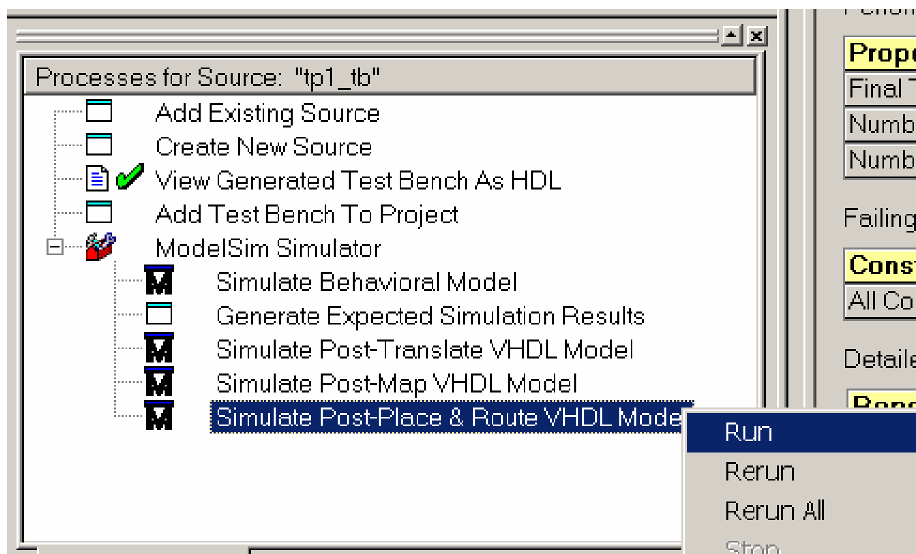
Le tableau suivant vous indique le rôle des différentes étapes de l'implémentation ainsi que les rapports associés :

Etape	Rapport	Signification
Translation	Translation report	Création d'un fichier de design unique
Mapping	Map report	Découpage du design en primitives
Placement-routage	Place & Route report Post Place & Route static timing report Pad report	Placement et routage des primitives Respect des contraintes temporelles et fréquence max de fonctionnement Assignation des broches du FPGA

9.1.9 Simulation de timing

Les outils de placement-routage peuvent fournir un nouveau modèle VHDL (tp1_timesim.vhd) appelé **<u>modèle VITAL</u>** qui correspond au modèle de simulation réel du circuit ainsi qu'un fichier contenant tous les timings du FPGA (tp1_timesim.sdf) appelé fichier **SDF**. A l'aide de ces deux fichiers et du fichier de stimuli tp1_tb.timesim_vhw généré en même temps, nous allons pouvoir vérifier le fonctionnement réel de notre design. C'est la simulation de timing (ou simulation Post-Place&Route ou encore simulation Post-layout ou simulation Post-Implementation).

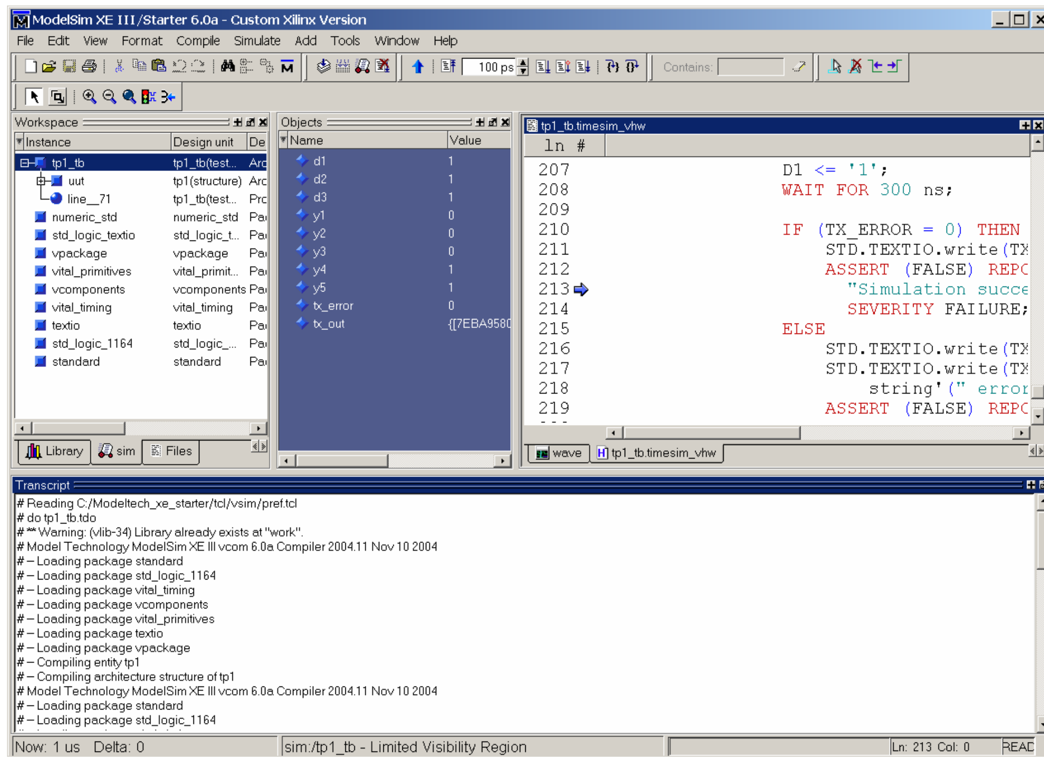
Pour lancer cette simulation, sélectionnez le fichier tp1_tb.tbw dans la fenêtre « Sources In Project ». Sélectionnez le processus « Simulate Post-Place&Route VHDL Model » puis cliquez avec le bouton droit de la souris sur le menu Run :

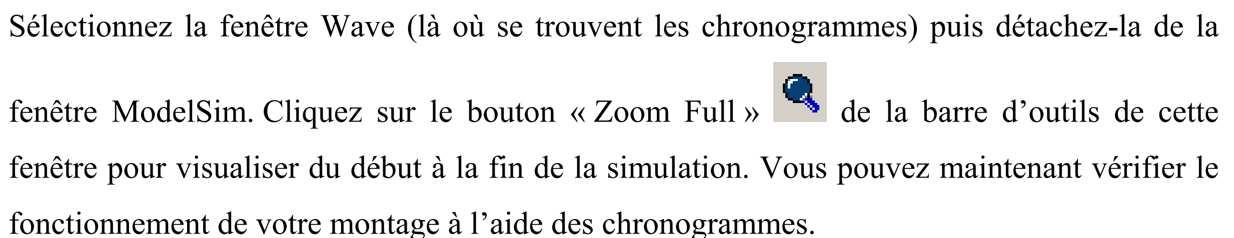


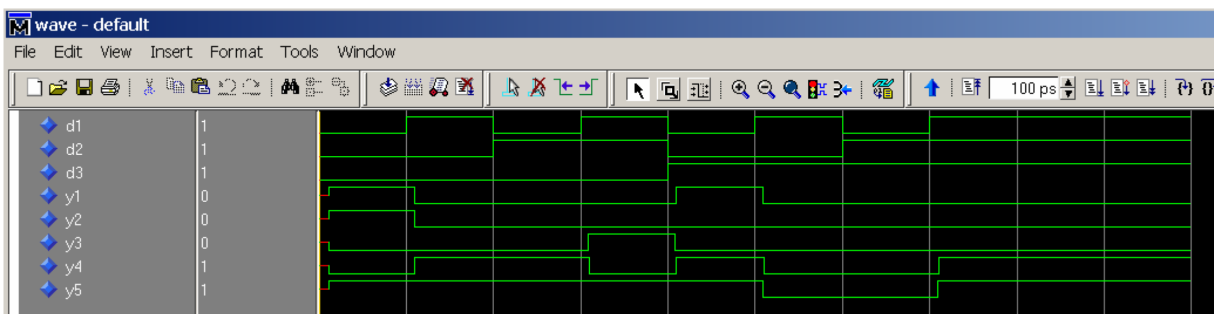
Le simulateur VHDL ModelSim démarre. Il va enchaîner automatiquement :

- la compilation du design et des stimuli,
- le lancement du simulateur,
- la simulation jusqu'à l'arrêt automatique du testbench.

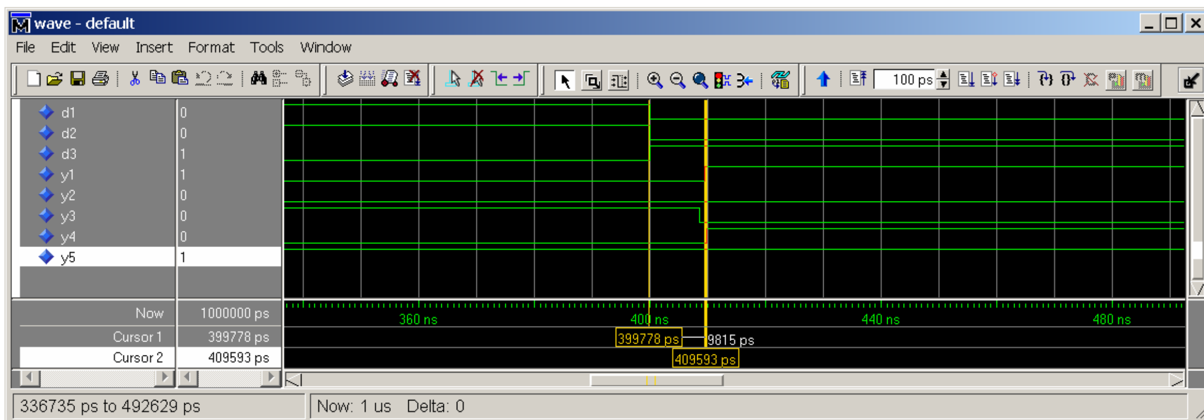
A la fin, la fenêtre Modelsim est la suivante :



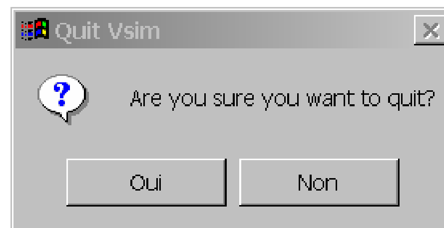
Sélectionnez la fenêtre Wave (là où se trouvent les chronogrammes) puis détachez-la de la fenêtre ModelSim. Cliquez sur le bouton « Zoom Full »  de la barre d'outils de cette fenêtre pour visualiser du début à la fin de la simulation. Vous pouvez maintenant vérifier le fonctionnement de votre montage à l'aide des chronogrammes.



Faîtes un zoom sur le chronogramme en cliquant en haut et à gauche de la zone à agrandir avec le bouton du milieu de la souris. Maintenez cliqué et déplacez la souris en bas et à droite de la zone (un rectangle bleu apparaît). Relâchez le bouton du milieu. Le zoom s'exécute. Sur l'exemple de chronogramme suivant, on voit clairement apparaître le décalage entre les entrées et les sorties. Utilisez les curseurs temporels pour mesurer ce délai.

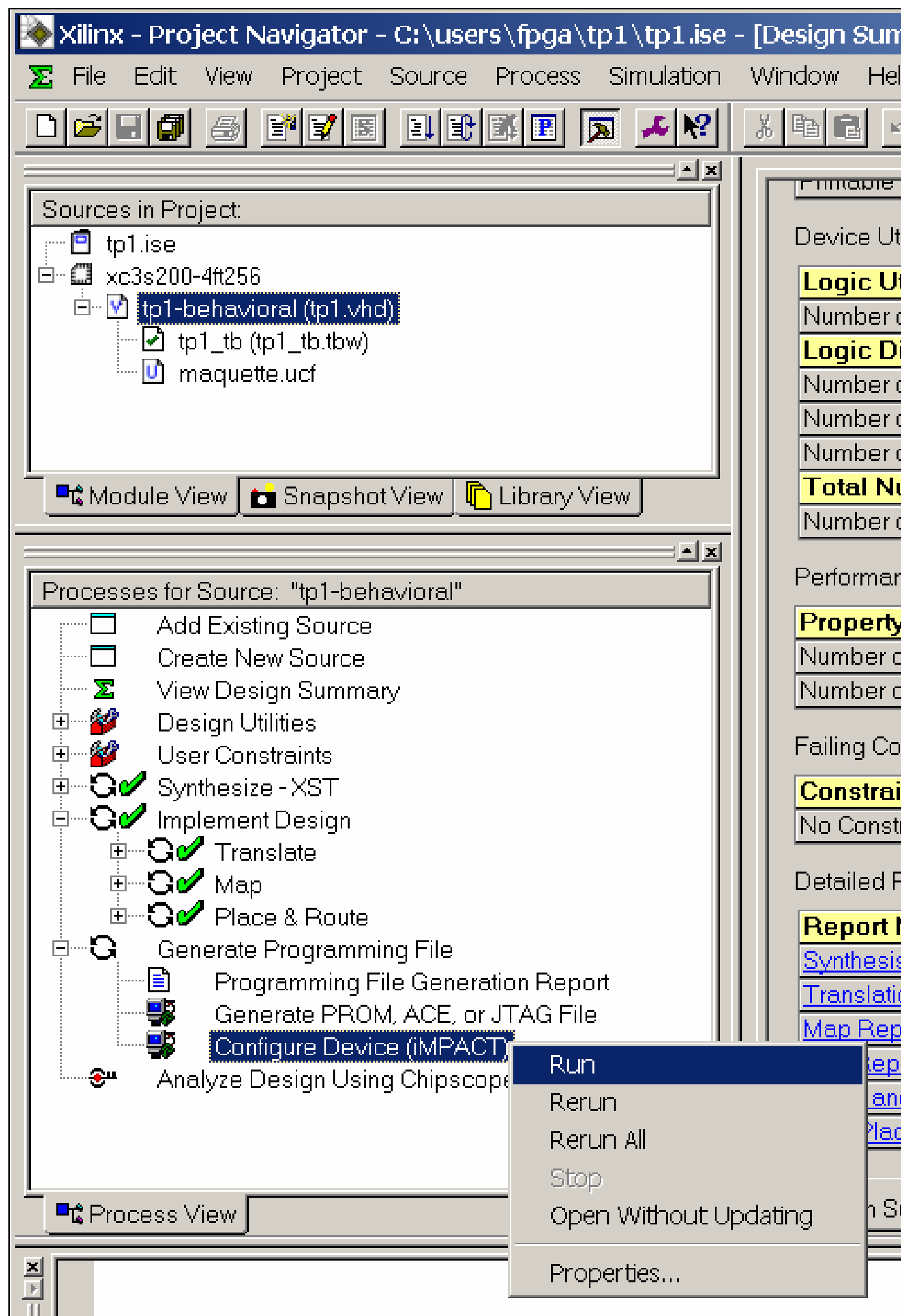


Quand la vérification est terminée, cliquez sur le menu « File » dans la fenêtre ModelSim, puis sur « Quit». Quand le message ci-dessous apparaît à l'écran, cliquez sur Oui.

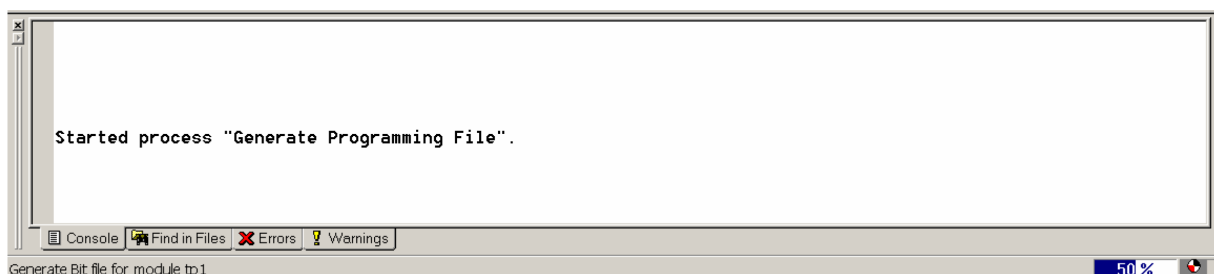


9.1.10 Configuration de la maquette

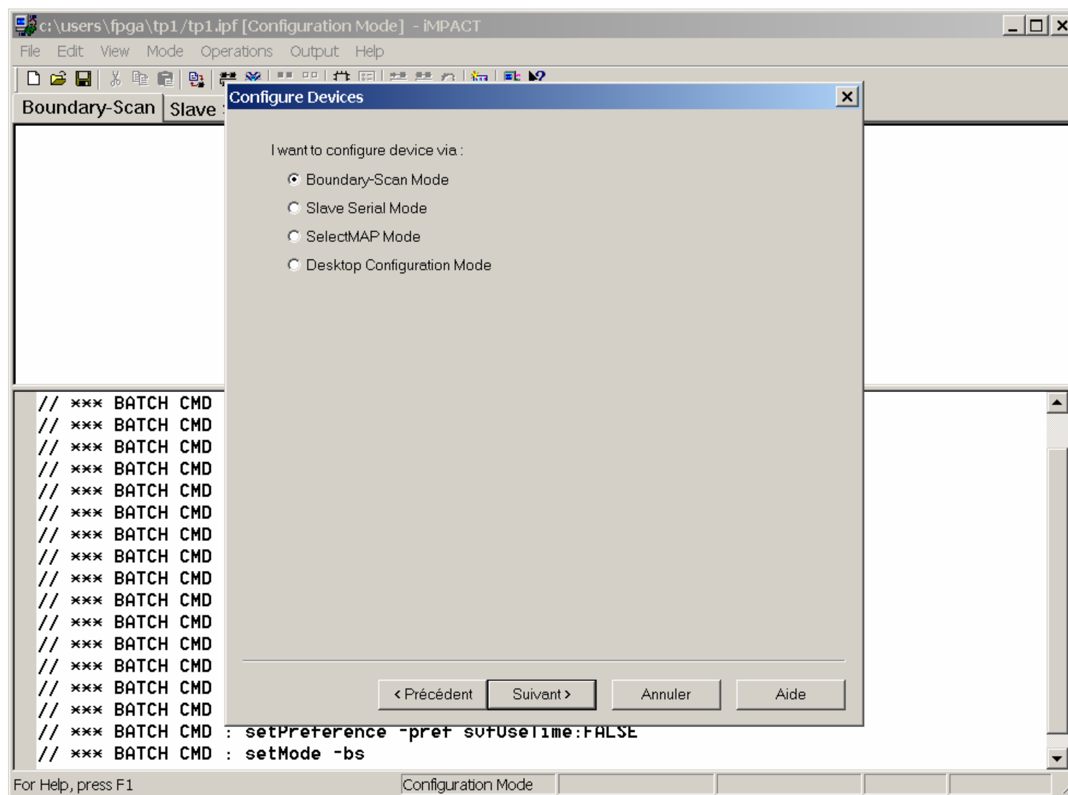
A ce point du TP, le design est entièrement vérifié. Nous pouvons maintenant le télécharger dans le FPGA. Sélectionnez le design tp1.vhd dans la fenêtre « Sources » puis « Configure Device » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



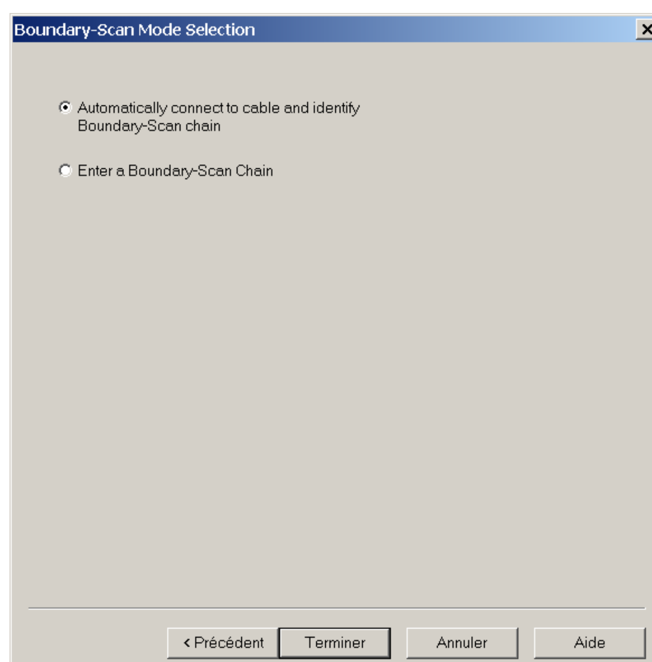
Le processus démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant les différentes opérations apparaît :



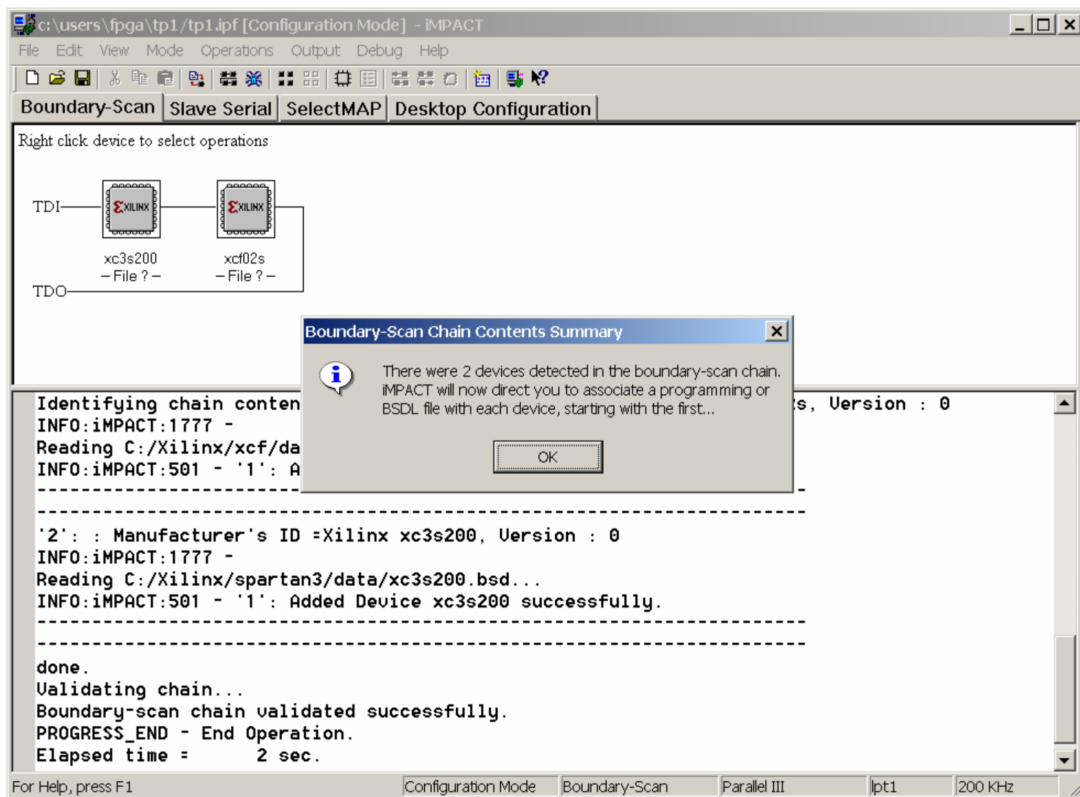
Puis la fenêtre de l'application Impact apparaît à l'écran. Nous allons configurer le FPGA en utilisant le mode « Boundary-scan » (JTAG). Cliquez sur le bouton « Suivant » :



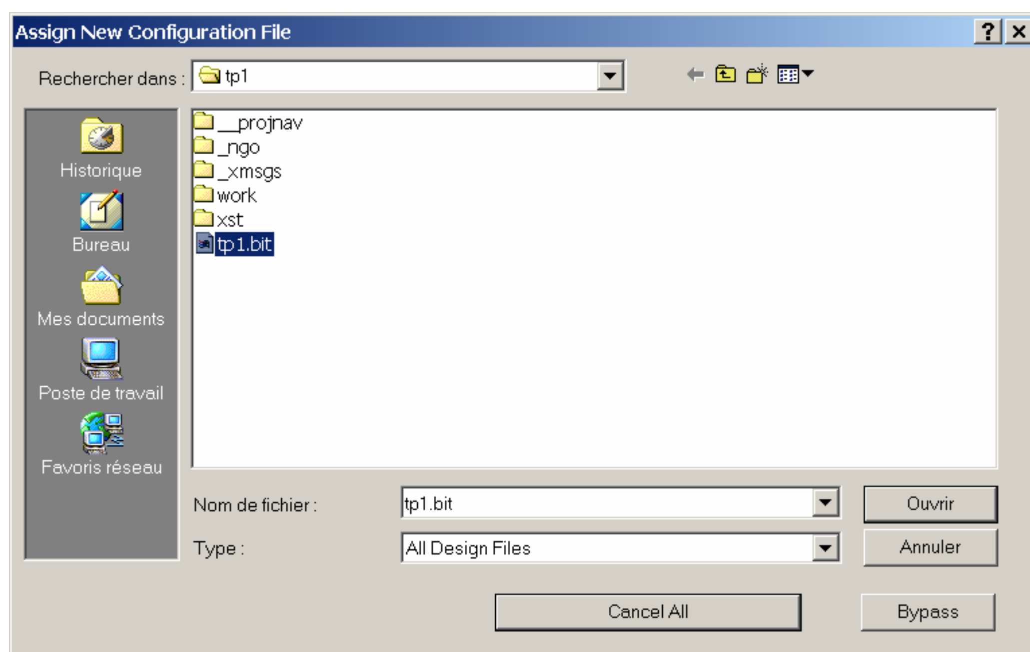
Laissons Impact découvrir automatiquement les circuits connectés sur la chaîne JTAG. Cliquez sur le bouton « Terminer » :



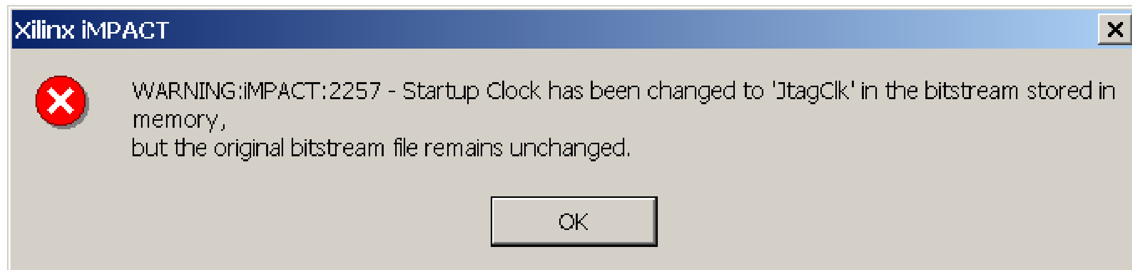
Impact a trouvé 2 circuits dans la chaîne JTAG, le FPGA et une mémoire Flash série que nous n'allons pas utiliser maintenant. Cliquez sur le bouton « OK » :



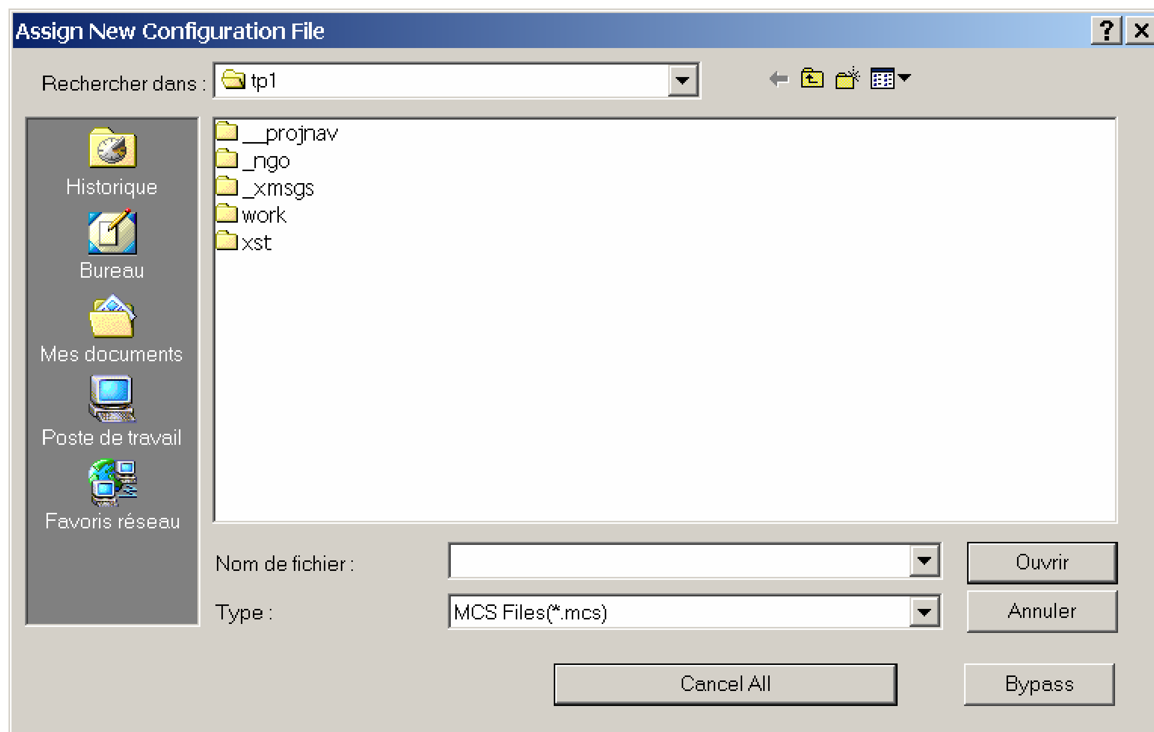
La question suivante concerne le nom du fichier de configuration à associer au FPGA. Dans notre exemple, il s'agit de tp1.bit. Sélectionnez-le, puis cliquez sur « Ouvrir » :



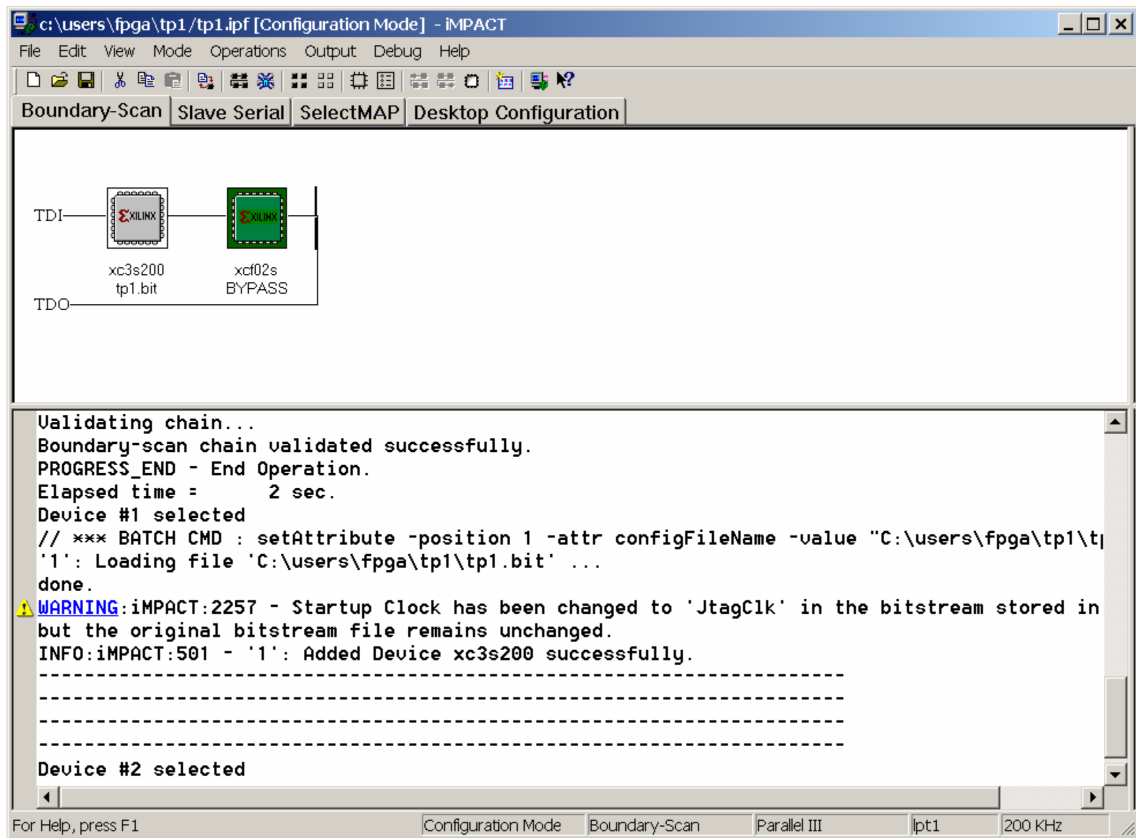
Par défaut, le FPGA est en mode maître, c'est à dire qu'il génère l'horloge CCLK de lecture de la configuration. Comme en JTAG, le FPGA est esclave, la fenêtre suivante vous avertit que l'horloge dans le fichier de configuration tp1.bit a été passée en mode JTAG (sur la copie en mémoire seulement). Cliquez sur « OK » pour poursuivre :



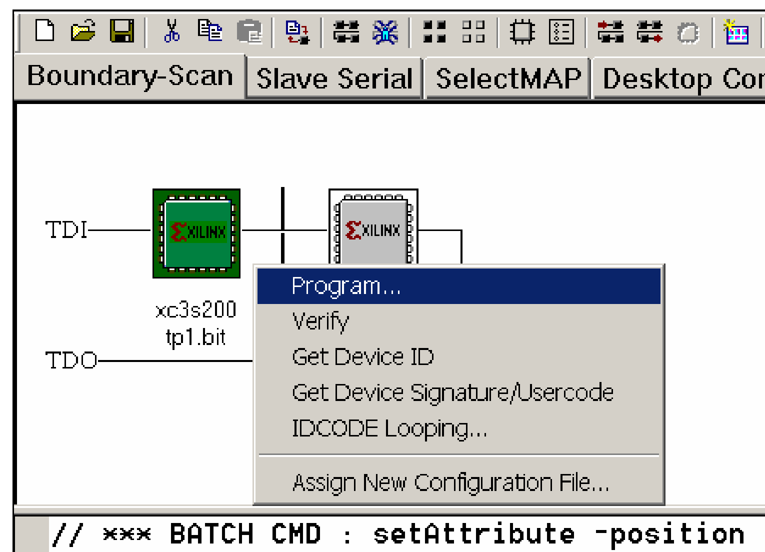
Impact demande ensuite quel fichier va être associé avec la mémoire Flash série. Comme nous n'avons pas créé le fichier pour la PROM (format MCS-86), cliquez sur « Bypass » pour sauter cette étape :



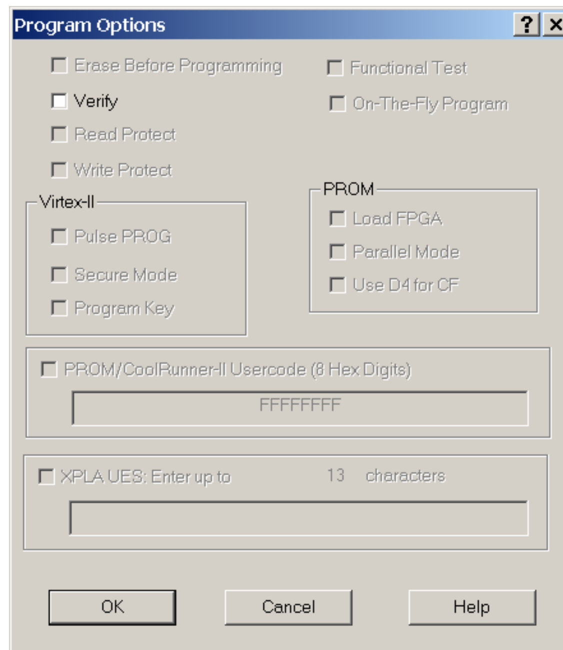
Finalement, on obtient la fenêtre suivante :



Pour configurer le FPGA, il suffit de le sélectionner en cliquant dessus, puis de cliquer avec le bouton droit de la souris puis sur « Program… » :



Dans la fenêtre qui s'ouvre, cliquez sur « OK » :



En moins de 10 secondes, le téléchargement est terminé. Fermez la fenêtre Impact sans sauver le projet. Essayez les différentes combinaisons de SW0, SW1 et SW2 et vérifiez à l'aide des leds LD0 à LD4 le bon fonctionnement du montage.

9.1.11 La suite du TP

En utilisant le même flot de développement, nous allons pouvoir tester les différents designs vus en cours, à savoir :

- Multiplexeurs (voir §2.4.3),
- Démultiplexeur version compacte (voir §2.4.5),
- Encodeur de priorité (voir §2.4.6),
- Mémoire ROM (voir §2.4.7).

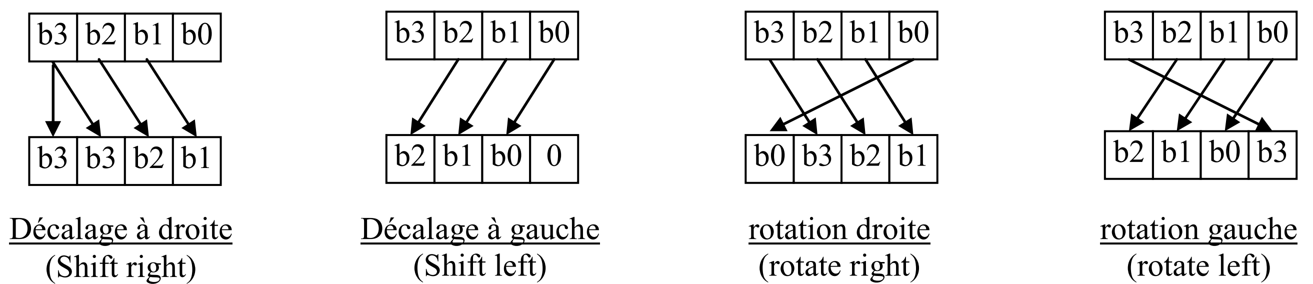
Vous appellerez un enseignant pour valider chaque design **que vous aurez simulé au préalable**. Vous connecterez les entrées sur les dip switches SW0 à SW7 et les sorties sur les leds LD0 à LD7 en partant des poids faibles. N'oubliez pas de modifier le fichier UCF pour chaque design.

9.2 Travail pratique N°2

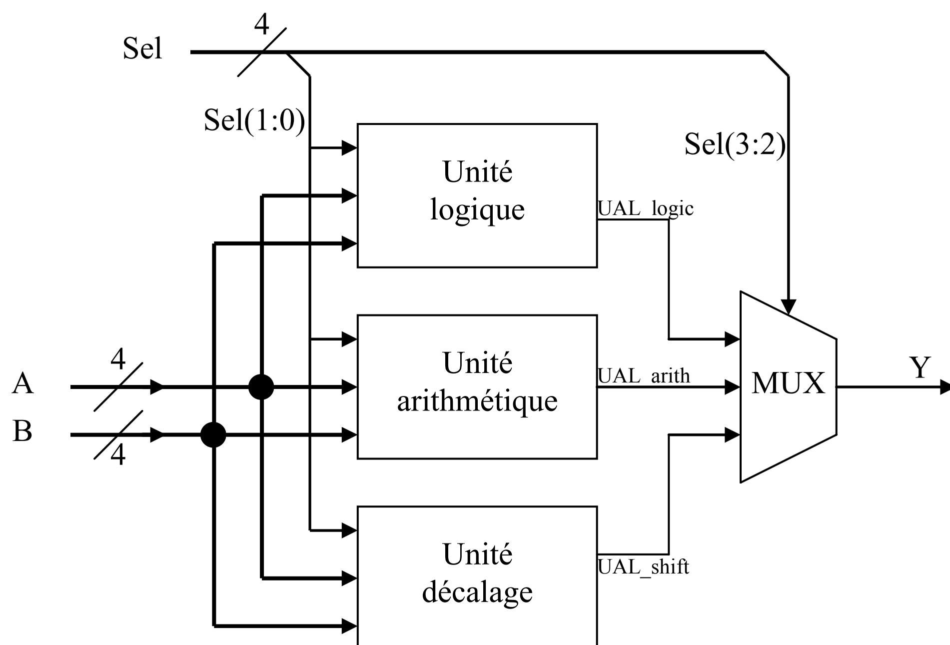
9.2.1 Cahier des charges

Le but de ce TP est de développer une Unité Arithmétique et Logique (UAL ou ALU en anglais) qui va réaliser :

- Des opérations arithmétiques (addition, soustraction, incrémentation, décrémentation),
- Des opérations logiques (and, or, xor et not),
- Des décalages ou des rotations (à gauche et à droite) symbolisés par la figure suivante. Vous noterez que le décalage à droite est réalisé avec extension de signe.



Le synoptique général est le suivant :



On a deux opérandes A et B sur 4 bits en entrée et une sortie sur 4 bits, Y (tous non signés). Le signal Sel, sur 4 bits, sert à sélectionner l'opération à réaliser, selon le tableau suivant :

Sel(3)	Sel(2)	Sel(1)	Sel(0)	opération	
0	0	0	0	Y = A + B	arithmétique
0	0	0	1	Y = A - B	
0	0	1	0	Y = A + 1	
0	0	1	1	Y = A - 1	
0	1	0	0	Y = A and B	logique
0	1	0	1	Y = A or B	
0	1	1	0	Y = A xor B	
0	1	1	1	Y = not B	
1	0	0	0	Y = shift left A	décalage, rotation
1	0	0	1	Y = shift right A	
1	0	1	0	Y = rotate left A	
1	0	1	1	Y = rotate right A	

Les 2 bits de poids fort de Sel servent à sélectionner l'unité de traitement. Les 2 bits de poids faible de Sel servent à sélectionner une opération dans l'unité en cours d'utilisation.

9.2.2 Réalisation pratique

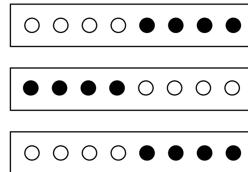
Vous allez créer un nouveau projet TP2 et saisir votre design. Vous écrirez un testbench pour faire la simulation fonctionnelle (**pas de synthèse sans simulation**), puis vous implémenterez votre design dans le FPGA. Vous connecterez Sel sur les boutons poussoirs de la carte (BTN0 à BTN 3), A et B sur les dip switches SW0 à SW7 et Y sur les leds LD0 à LD3. Vous vous aiderez de la simulation fonctionnelle pour valider l'UAL sur la maquette.

9.3 Travail pratique N°3

9.3.1 Cahier des charges

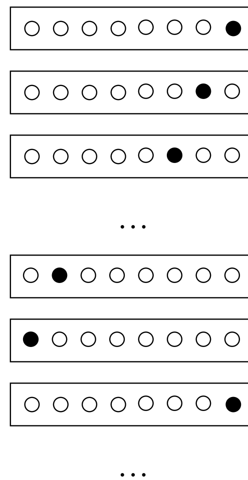
Le but de ce TP est de développer un chenillard qui va réaliser trois séquences de clignotement différentes sur les 8 leds avec une horloge externe basse fréquence (entrée Hin sur la maquette, fréquence 2 Hz) :

- Séquence 1 (registre),

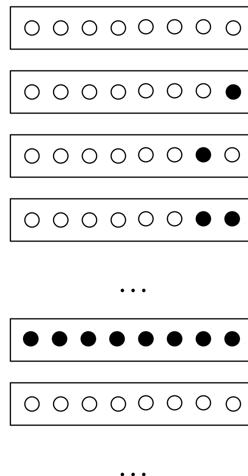


…

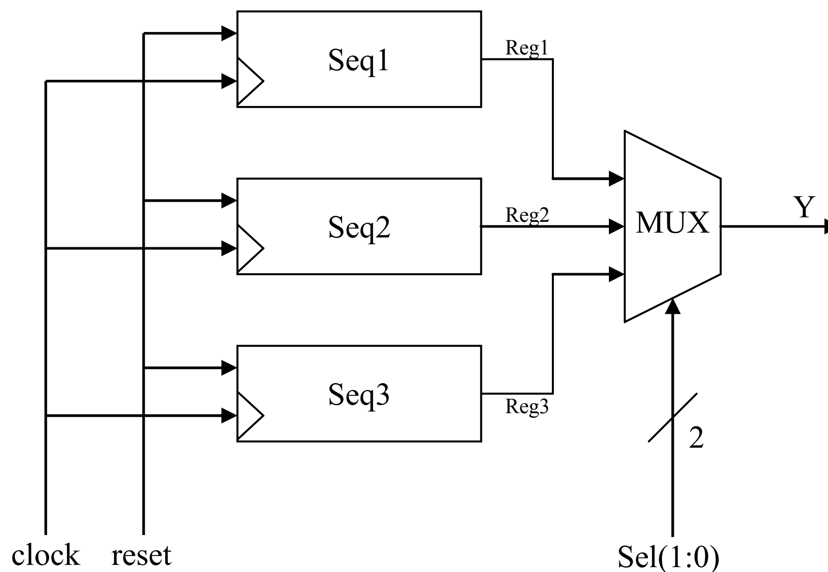
- Séquence 2 (décalage),



- Séquence 3 (comptage),



Le synoptique général sera le suivant :



Le signal Sel, sur 2 bits, sert à sélectionner la séquence, selon le tableau suivant :

Sel(1)	Sel(0)	
0	0	Séquence 1
0	1	Séquence 2
1	0	Séquence 3
1	1	Toutes les leds éteintes

La logique fonctionnera sur front montant avec un reset asynchrone.

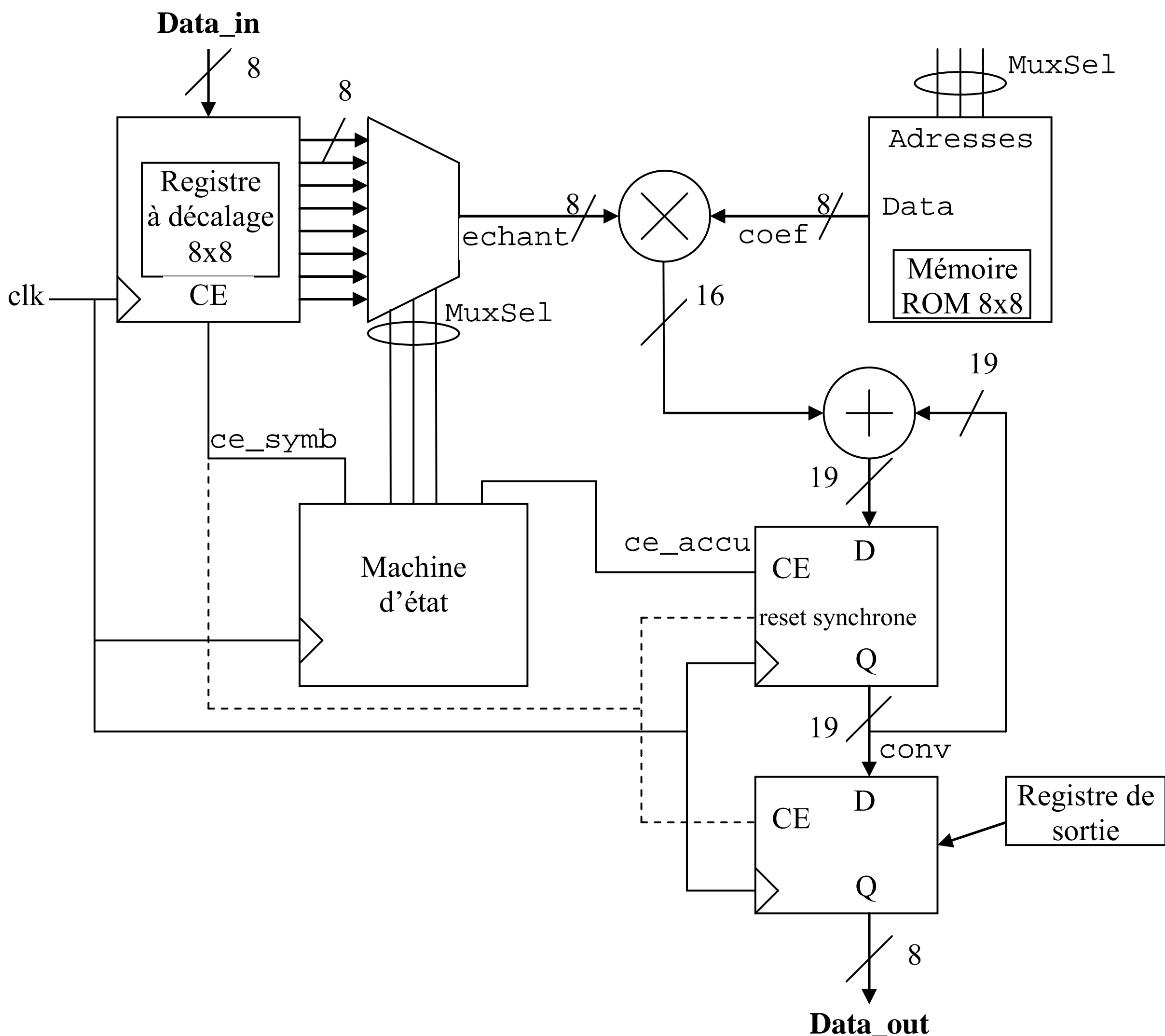
9.3.2 Réalisation pratique

Vous allez créer un nouveau projet TP3 et saisir votre design. Vous écrirez un testbench pour faire la simulation fonctionnelle (**pas de synthèse sans simulation**), puis vous implémenterez votre design dans le FPGA. Vous connecterez Sel sur les dip switches SW0 et SW1, l'horloge sur Hin, le Reset sur le bouton poussoir BTN3 et Y sur les leds LD0 à LD7.

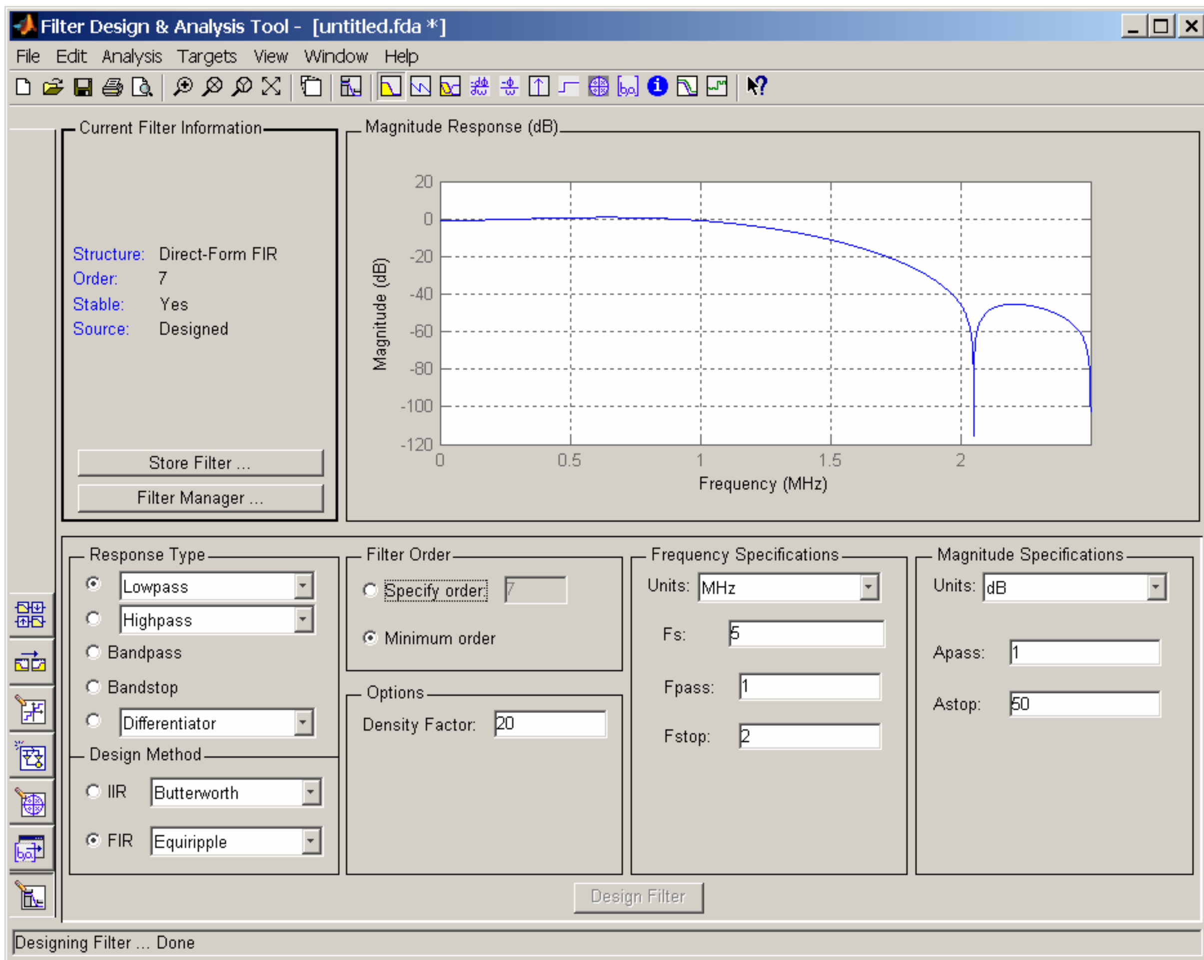
9.4 *Travail pratique N°4*

9.4.1 Cahier des charges

Le but de ce TP est de développer un filtre à réponse impulsionnelle finie (FIR) à structure MAC qui s'inspirera du calcul de la moyenne mobile vu en cours (§5.3.4). Le synoptique général sera le suivant :



La mémoire contiendra les 8 coefficients du filtre. Comme nous l'avons vu avec la moyenne mobile, 10 périodes d'horloge sont nécessaires pour calculer un échantillon. L'horloge de la carte étant égale à 50 MHz, la fréquence d'échantillonnage vaudra 5 Ms/s. Avec Matlab (fdatool), nous allons calculer les coefficients du filtre passe-bas :



Les 8 coefficients obtenus forment une réponse impulsionnelle symétrique exprimée avec des nombres réels.

```
Command Window

  To get started, select MATLAB Help or Demos from the Help menu.

>> fdatool
>> Num

Num =

   -0.0389   -0.0893    0.1062    0.4736    0.4736    0.1062   -0.0893   -0.0389

>> |
```

Il faut les convertir en entiers codés sur 8 bits CA2. Deux options sont possibles :

- On normalise la somme des coefficients à 1 (128 en CA2) pour que le gain du filtre soit unitaire.

- On convertit de telle sorte que les coefficients les plus grands valent 1, c'est-à-dire pour avoir la dynamique maximale en sortie afin d'exploiter tous les bits utilisés pour le calcul.

Dans les deux cas, il faudra choisir judicieusement les 8 bits à extraire des 19 bits en sortie de l'accumulateur de produit. Nous allons utiliser la première solution : le gain unitaire.

```
Command Window
>> Num

Num =

   -0.0389   -0.0893    0.1062    0.4736    0.4736    0.1062   -0.0893   -0.0389

>> b=Num*127/sum(Num)

b =

   -5.4743  -12.5502   14.9339   66.5906   66.5906   14.9339  -12.5502   -5.4743

>> sum(b)

ans =

  127.0000

>>
```

Il ne reste plus qu'à arrondir :

```
>> round(b)

ans =

    -5   -13    15    67    67    15   -13    -5

>>
```

Ce sont ces valeurs qu'il faudra mettre dans la mémoire. Au lieu d'utiliser un tableau de `std_logic_vector` pour constituer la mémoire ROM comme nous l'avons vu au §2.4.7, nous prendrons un tableau d'`integer range -128 to 127`. Cela facilitera la saisie des valeurs des coefficients.

Les convertisseurs de la maquette fonctionnent en binaire naturel. Or, le calcul de la convolution va se faire en CA2. Il va donc falloir convertir data_in en CA2 à l'entrée (par inversion du bit de poids fort) :

```
data_i <= not data_in(7)&data_in(6 downto 0);
```

et convertir `conv` en binaire naturel en sortie (toujours en inversant le bit de poids fort) :

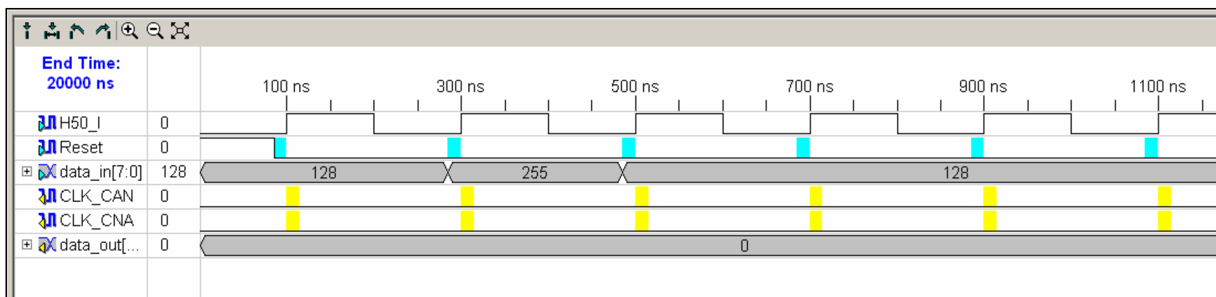
```
data_out <= not Conv(14)&Conv(13 downto 7);
```

Ces conversions sont obligatoires sinon les calculs seront justes en simulation, mais complètement faux sur la maquette. En réalité, l'inversion du bit de poids fort de data_in ne réalise pas une conversion en CA2, mais un changement d'échelle. Prenons un exemple sur 3 bits et visualisons l'échelle des valeurs en binaire naturel avec et sans inversion du bit de poids fort :

Echelle en binaire naturel		Echelle avec inversion du bit de poids fort	
Valeur décimale	Valeur binaire	Valeur décimale en CA2	Valeur binaire
7	1 1 1	3	0 1 1
6	1 1 0	2	0 1 0
5	1 0 1	1	0 0 1
4	1 0 0	0	0 0 0
3	0 1 1	-1	1 1 1
2	0 1 0	-2	1 1 0
1	0 0 1	-3	1 0 1
0	0 0 0	-4	1 0 0

L'inversion du bit de poids fort permet seulement de faire passer l'échelle de [0 : 7] à [-4 : 3], donc d'une échelle en binaire naturel à une échelle en CA2. Il ne s'agit donc pas d'une conversion en CA2, mais d'un glissement d'échelle. L'opération inverse permet le passage d'une échelle CA2 à une échelle en binaire naturel.

Le test du filtre ne pose pas de problème particulier. Nous allons simuler une réponse impulsionnelle avec un échantillon à 1 (255 en non signé 8 bits sur l'entrée, 127 dans le design après inversion du bit de poids fort) suivi d'échantillons à 0 (128 en non signé 8 bits sur l'entrée, 0 dans le design après inversion du bit de poids fort) pendant 20 µs (il nous faut au moins 8x10 périodes d'horloge pour voir apparaitre la réponse du filtre).



Il reste un point à traiter : nous devons extraire 8 bits (pour former data_out) parmi les 19 disponibles en sortie du MAC (signal `conv`). Lesquels devons-nous choisir ?

Il n'y a pas de réponse unique à ce problème :

1. Solution la plus simple et la plus sure ; on prend les 8 bits de poids fort. L'avantage, c'est que le débordement est impossible. L'inconvénient, c'est que l'amplitude du signal de sortie à toutes les chances d'être trop faible pour que l'on puisse l'exploiter.

2. Solution la plus compliquée ; on effectue le calcul en virgule fixe. Il faut alors calculer la position de la virgule en tout point du montage. Les entrées sont en Q7/8, la sortie du multiplieur est en Q14/16 et la sortie de l'accumulateur est en Q14/19. Pour repasser en Q7/8, il faut ignorer les 4 bits de poids fort en sortie du MAC et écrire (en inversant le poids fort pour tenir compte de la conversion CA2 → binaire naturel) :

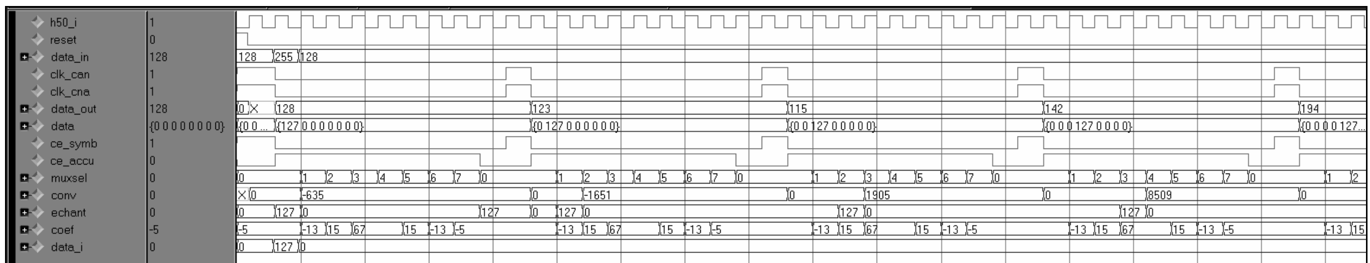
   ```
   data_out <= not Conv(14)&Conv(13 downto 7);
   ```

   Cette méthode n'a aucun inconvénient excepté la complexité du calcul. Nous l'utiliserons dans ce TP.

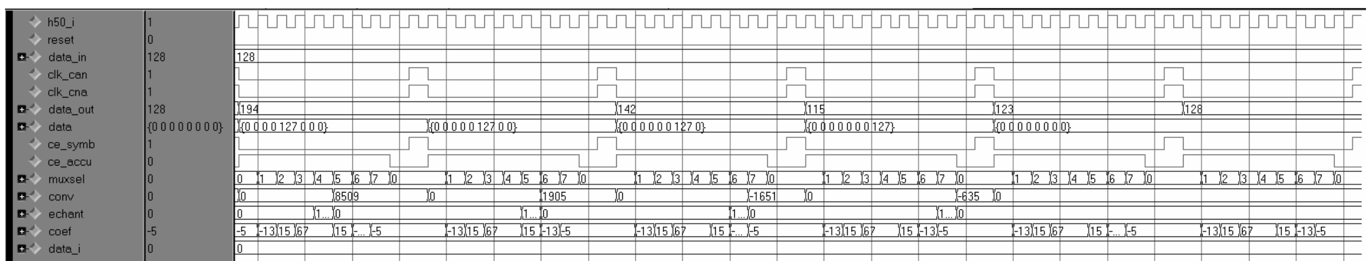
3. Solution efficace sans calcul ; on simule le montage avec un signal réaliste à l'entrée et on observe les bits utiles en sortie. Cette méthode peut aussi être utilisée pour confirmer les résultats de la méthode précédente. Cette solution nécessite un savoir faire au niveau de l'écriture du testbench que nous verrons plus tard. Il faut aussi que le signal d'entrée servant pour la simulation soit représentatif du cas réel. Sinon, le débordement est inéluctable.

La réponse du montage doit être similaire à celle obtenue sur les chronogrammes suivants. Vous noterez que l'on voit apparaitre sur data_out les coefficients du filtre (il faut ajouter 128 à cause de l'inversion du bit de poids fort). C'est une méthode commode pour vérifier le filtre. En réalité, il y a parfois une erreur sur la valeur du coefficient car 127 en CA2 n'est pas égal à 1.0 mais à 127/128.

<u>Début de la simulation :</u>



<u>Fin de la simulation :</u>



La logique du montage doit fonctionner sur le front montant de l'horloge avec un reset asynchrone pour tous les composants sauf bien sûr pour le MAC qui est remis à 0 au début de chaque cycle de calcul.

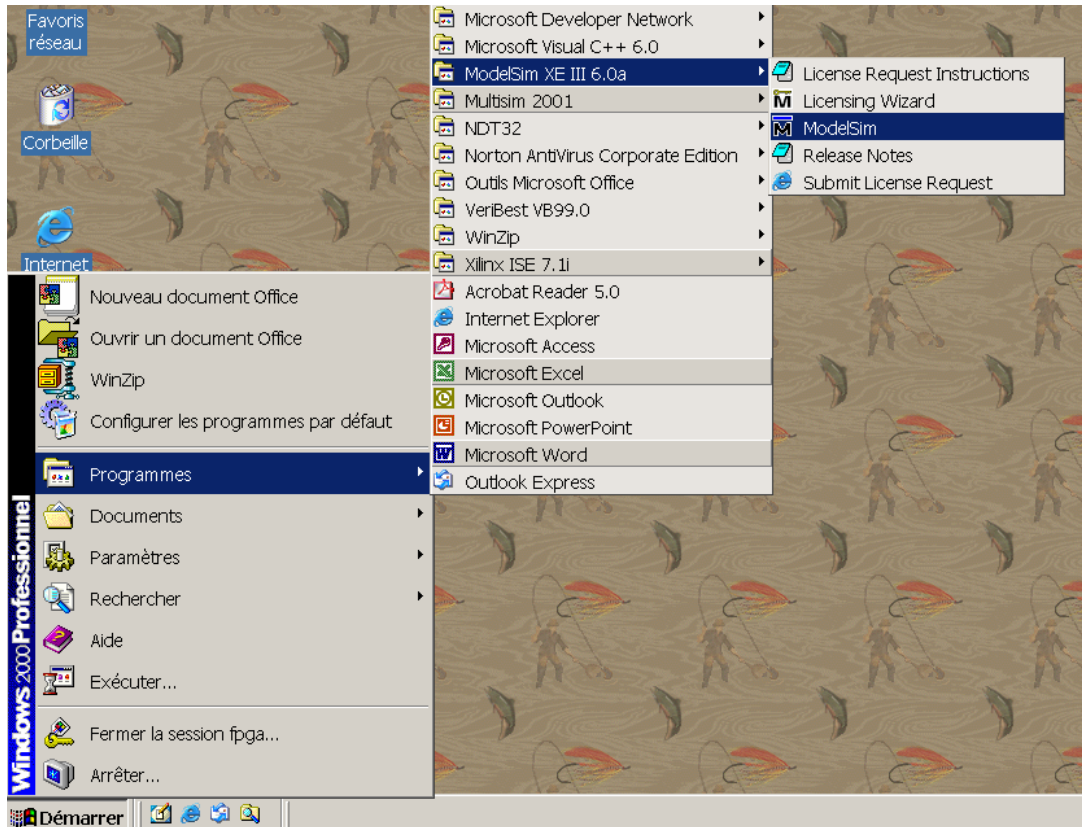
9.4.2 Réalisation pratique

Vous allez créer un nouveau projet TP4. Deux fichiers vous seront fournis : fir.vhd et fir_pkg.vhd. Ils contiennent le top level design et le package du design « calcul de moyenne mobile » vu au §5.3.4. Vous les modifierez pour réaliser le FIR MAC : vous nommerez **`fir_mac`** l'entité du top level design et **`fir_mac_pkg`** le package. Vous écrirez ensuite un testbench pour faire la simulation fonctionnelle (**pas de synthèse sans simulation**), puis vous implémenterez votre design dans le FPGA. Vous connecterez le Reset sur le bouton poussoir BTN3, data_in sur le CAN, data_out sur le CNA, sans oublier les horloges CLK_CAN et CLK_CNA. Toutes les sorties seront mises en mode FAST. Vous connecterez un signal sinusoïdal d'amplitude 1 Vcàc (sur 50 Ω) sur l'entrée Din et vous observerez Dout à l'aide d'un oscilloscope. L'objectif est de mesurer la réponse en fréquence du filtre entre 500 kHz et 2,5 MHz.

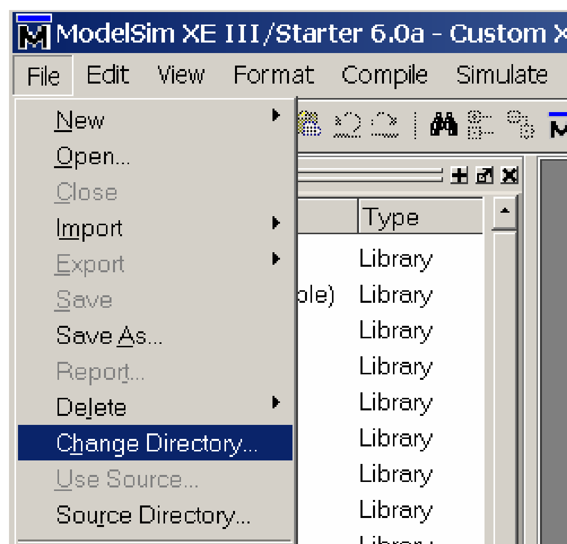
Question supplémentaire facultative : étendez le montage à 16 coefficients. Vous modifierez la machine d'états pour que sa description ne soit pas interminable (en clair, il faudra utiliser un compteur pour le calcul de la convolution).

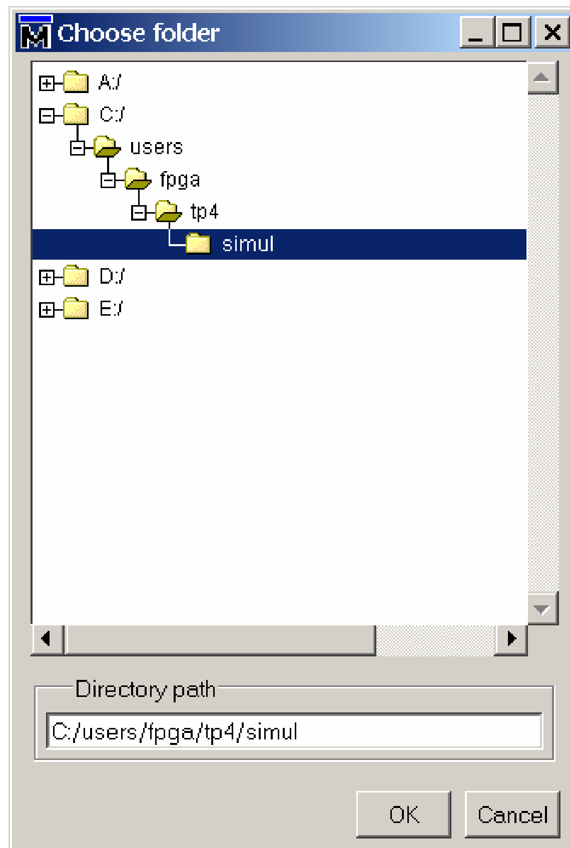
9.5 Travail pratique N°5

Ce TP est consacré entièrement à ModelSim. Créez un répertoire `simul` dans le projet TP4 et copiez y les fichiers `fir.vhd` et `fir_pkg.vhd`. Lancez ModelSim :

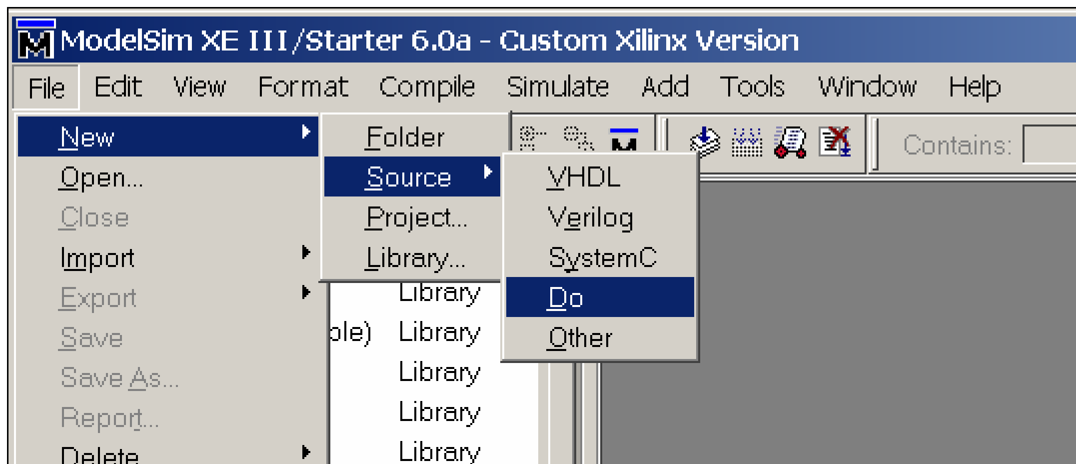


Puis changez le répertoire de travail :





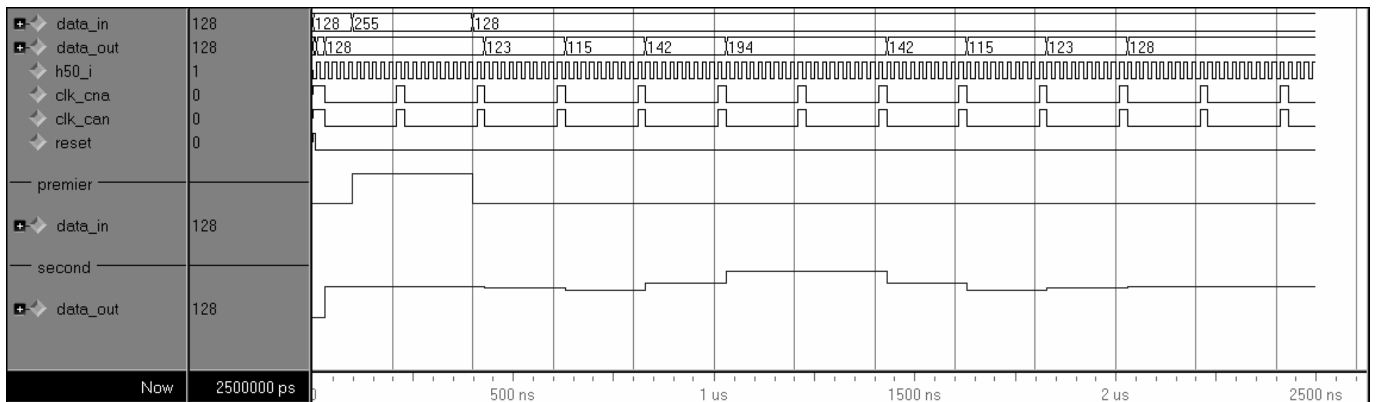
Créez une libraire `work`, puis compilez les fichiers `fir_pkg.vhd` et `fir.vhd` (le package en premier). Vous pouvez créer un fichier de commande (fichier `do`) pour automatiser ces taches en faisant :



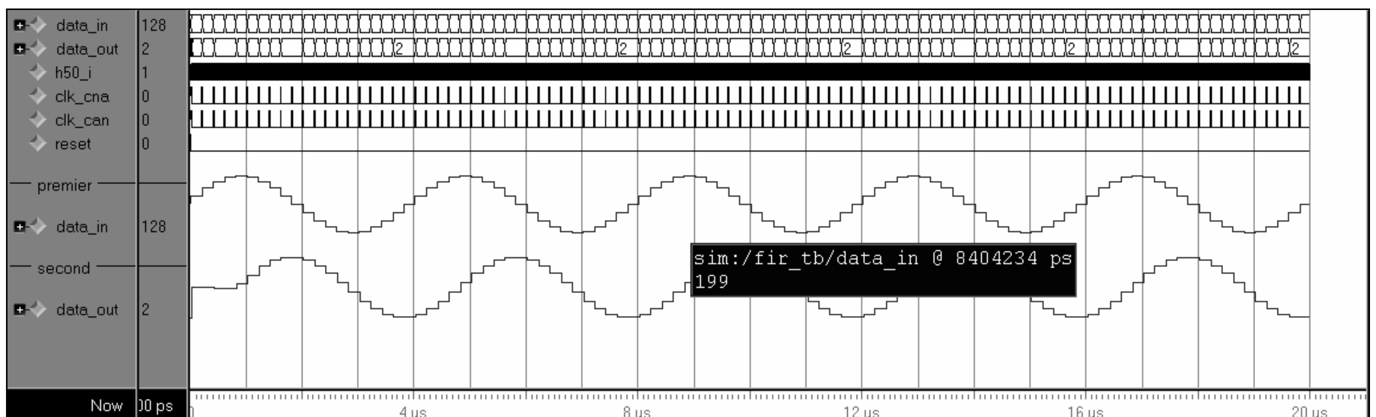
Vous l'appellerez fir.do. Ce fichier doit être lancé en tapant : `do fir.do` dans la fenêtre transcript de ModelSim.

Vous pouvez maintenant créer votre testbench. Vous en ferez trois versions :

1) Ecriture d'un testbench identique à celui de TP4 (réponse impulsionnelle). Vous devez obtenir le chronogramme suivant :

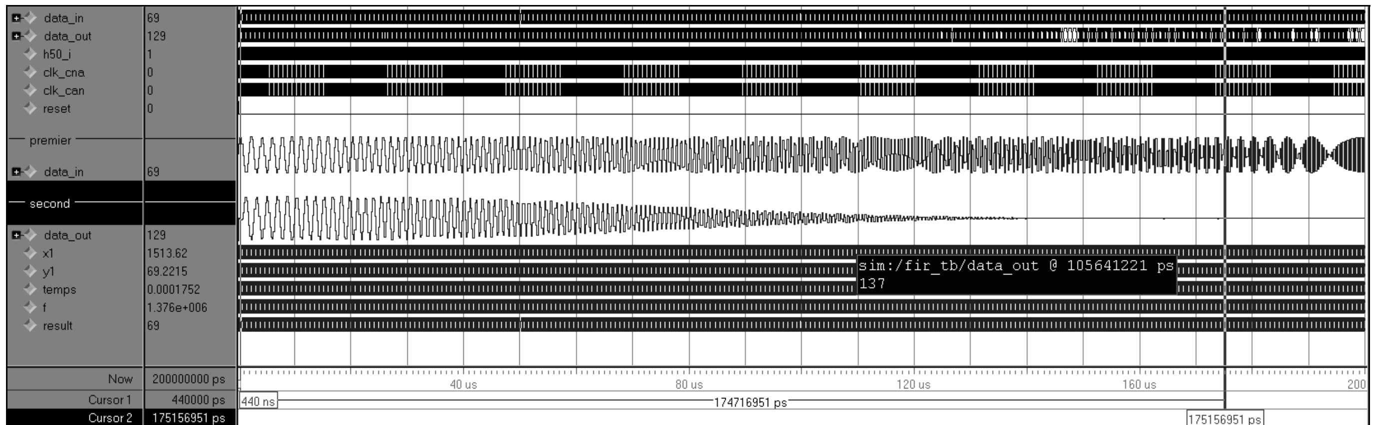


2) Création d'un fichier contenant un sinus de fréquence 250 kHz avec Matlab, écriture d'un testbench avec lecture des stimuli dans un fichier `in.dat` et écriture de la réponse dans un fichier `out.dat` puis visualisation de la réponse du filtre sous Matlab. Un fichier `convert.vhd` contenant les trois fonctions de conversion vous sera fourni. Vous devez obtenir le chronogramme suivant :



ATTENTION : à cette fréquence, le filtre a un gain légèrement supérieur à 1. Pour éviter un débordement de calcul, il faut créer un signal sinusoïdal dont l'amplitude est comprise entre 8 et 248.

3) Ecriture d'un testbench qui génère un sinus dont la fréquence varie entre 500 kHz et 1.5 MHz afin de visualiser la réponse en fréquence du FIR. Faites attention à la durée de la simulation. Son amplitude sera comprise entre 28 et 228 pour éviter tout débordement. Vous devez obtenir le chronogramme suivant :



9.6 Travail pratique N°6

9.6.1 Cahier des charges

Le but de ce TP est de réaliser un FIR à structure parallèle 8 coefficients comme celui du §7.4.3 du cours. Vous reprendrez pour cela le travail réalisé pour TP4. Les coefficients seront identiques ainsi que la méthode de simulation par la réponse impulsionnelle. Il n'est pas nécessaire d'utiliser un package avec des composants, le design complet pouvant être mis dans le top level suivant :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

entity fir_par is
  port( H50_I : in std_logic;
        Reset : in std_logic;
        data_in : in std_logic_vector(7 downto 0);
        CLK_CAN : out std_logic;
        CLK_CNA : out std_logic;
        data_out : out std_logic_vector(7 downto 0));
end fir_par;

architecture a1 of fir_par is
     signal data_i : std_logic_vector(7 downto 0);
     signal conv : std_logic_vector(18 downto 0);
     ...
begin

     CLK_CAN <= ce_symb;
     CLK_CNA <= ce_symb;
     data_i <= not data_in(7)&data_in(6 downto 0);
     data_out <= not conv(14)&conv(13 downto 7);

     ...

     process(H50_I, Reset) begin
          if (Reset='1') then
               ce_symb <= '0';
               ...
          elsif (H50_I'event and H50_I='1') then
               ce_symb <= not ce_symb;
               if (ce_symb = '1') then
                    ...
               end if;
          end if;
     end process;
end;
```

Le FIR peut normalement fonctionner à une fréquence supérieure à 50 MHz. Mais le CAN ne peut pas dépasser les 32 Ms/s. La fréquence du signal `ce_symb` qui sert d'horloge pour le CAN et le CNA a donc été fixée à 25 MHz ce qui doit conduire à un filtre qui commence à atténuer à 5 MHz.

Vous allez réaliser un premier FIR à structure parallèle sans pipeline. Vous noterez la quantité de logique utilisée et vous placerez une contrainte de 18 ns sur l'horloge H50_I en rajoutant les lignes suivantes à la fin du fichier de contraintes maquette.ucf :

```
NET "H50_I" TNM_NET = "H50_I";
TIMESPEC "TS_H50_I" = PERIOD "H50_I" 18 ns HIGH 50 %;
```

Vous pourrez vérifier que cette contrainte est respectée à la fin du rapport d'implémentation. Vous réaliserez ensuite un deuxième FIR à structure parallèle avec pipeline. Vous noterez la quantité de logique utilisée et vous placerez une contrainte de 8 ns sur l'horloge H50_I.

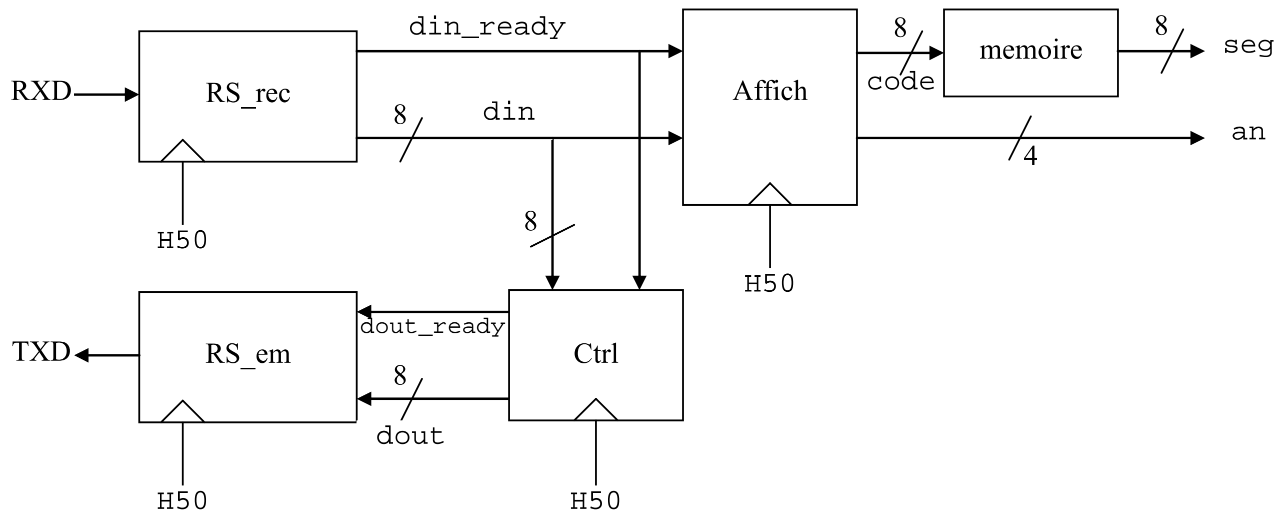
9.6.2 Réalisation pratique

Vous allez créer un nouveau projet TP6. La réalisation pratique sera similaire à celle décrite dans TP4.

9.7 Projet

9.7.1 Cahier des charges

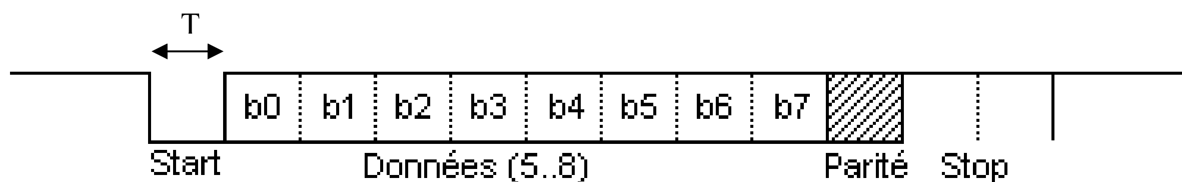
Le but de ce projet est de recevoir des caractères par le port série de la carte et de les afficher sur les 4 afficheurs 7 segments. Les caractères seront envoyés en utilisant l'application Windows HyperTerminal. Le synoptique général pourrait être le suivant :



Pour avoir des renseignements sur la liaison série (ou RS-232), il faut chercher sur Internet avec par exemple l'URL suivante :

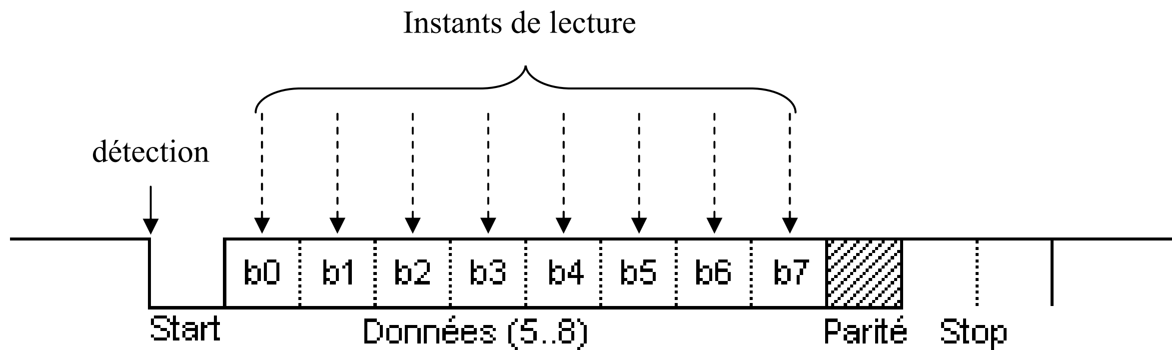
http://www.tavernier-c.com/serie.htm

Le signal sur l'entrée RXD du FPGA est de la forme suivante :



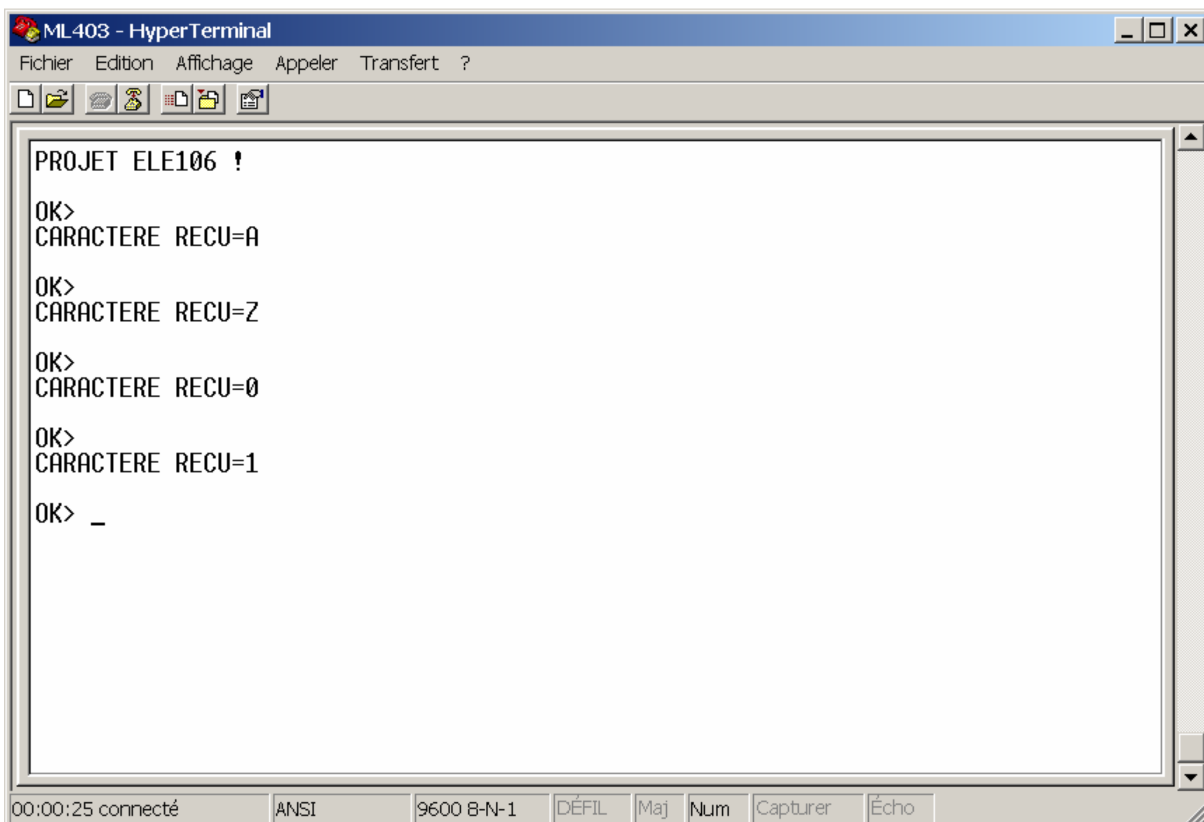
La liaison se fera avec un débit égal à 9600 bauds, sur 8 bits sans parité avec un bit de stop (le contrôle de flux sera désactivé sur le port série du PC). T, la durée du moment élémentaire, sera donc égale à 1/9600 soit 104,17 µs ou encore 5208 périodes de l'horloge à 50 MHz. Pour

lire de manière fiable la donnée sur RXD, il faudra détecter le front descendant du bit Start, puis lire les 8 bits en se plaçant au milieu du moment élémentaire :



Lorsque le décodeur RS-232 `RS_rec` a lu un caractère, son code ASCII est disponible sur `din` et le signal `din_ready` passe à 1 pendant une période de l'horloge à 50 MHz.

Le contrôleur `Ctrl` qui gère les messages envoyés au PC détecte qu'un caractère a été reçu et renvoie un message vers HyperTerminal via le bloc `RS_em`. Voici un exemple de dialogue :

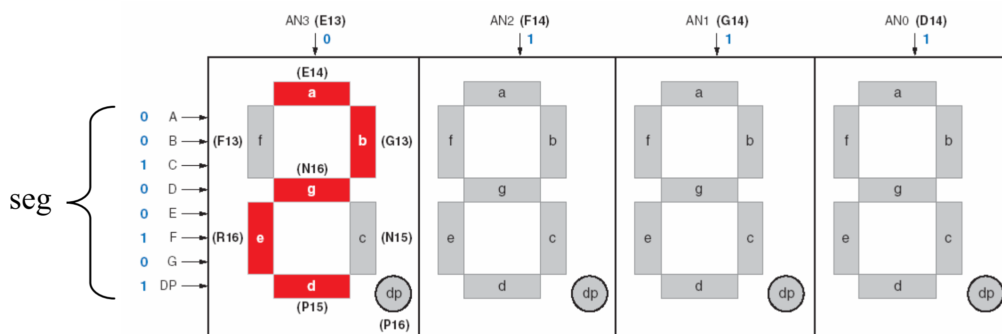


Lorsque le codeur RS-232 `RS_em` voit le signal `dout_ready` passer à 1 pendant une période de l'horloge à 50 MHz, il lit le code ASCII disponible sur `dout` puis l'envoie sur TXD. Le format du signal à envoyer sur TXD est le même que celui reçu sur RXD. La durée d'un moment élémentaire sera égale à 5208 périodes de l'horloge à 50 MHz.

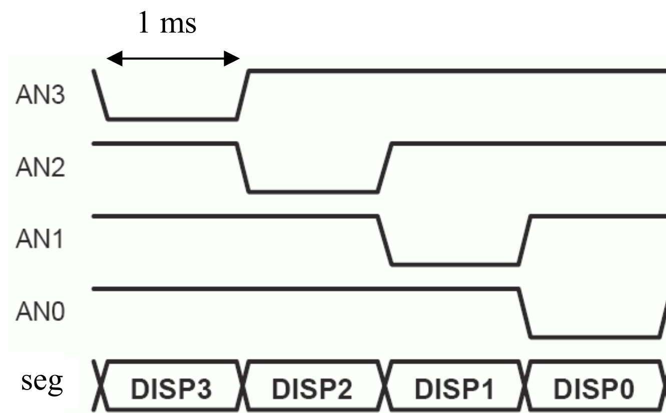
Quand `din_ready` passe à 1, le composant `Affich` va mémoriser le caractère pour l'envoyer sur les afficheurs 7 segments. Les caractères s'afficheront les uns à coté des autres, de droite à gauche puis repasseront à droite quand les 4 afficheurs seront occupés. La mémoire servira à convertir le code ASCII à afficher en un mot de 8 bits allumant ou éteignant les 7 segments plus le point décimal dp. Le signal de sortie `seg` est branché sur les segments de la manière suivante :

seg(7)	dp
seg(6)	g
seg(5)	f
seg(4)	e
seg(3)	d
seg(2)	c
seg(1)	b
seg(0)	a

Le signal `an` permet d'allumer alternativement les 4 afficheurs qui sont multiplexés afin de diminuer la consommation du montage :



Voici le séquencement à réaliser pour allumer successivement tous les afficheurs. La durée d'affichage d'un caractère sera égale à 1 ms. Il faut appliquer sur `seg` au bon moment les caractères à afficher :



9.7.2 Réalisation pratique

Vous allez créer un nouveau projet appelé `Projet`. Le top level suivant vous est proposé (conforme au synoptique général) :

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.projet_pkg.all;

entity projet is
    port(H50M : in std_logic;
         Reset : in std_logic;
            RXD : in std_logic;
            TXD : out std_logic;
            an : out std_logic_vector(3 downto 0);
           seg : out std_logic_vector(7 downto 0));
end projet;
architecture comporte of projet is
  signal dout : std_logic_vector(7 downto 0);
  signal din : std_logic_vector(7 downto 0);
  signal code : std_logic_vector(7 downto 0);
  signal din_ready : std_logic;
  signal dout_ready : std_logic;
begin
  ctrl : controle port map(H50M, Reset, din, din_ready, dout, dout_ready);
  rs_in : rs_rec port map(H50M, Reset, RXD, din, din_ready);
  rs_out : rs_em port map(H50M, Reset, dout, dout_ready, TXD);
  septseg : memoire port map(code, seg);
  aff : affiche port map(H50M, Reset, din, din_ready, code, an);
end comporte ;
```

Vous créerez un package `projet_pkg`, un testbench `projet_tb` (sans utiliser HDLBencher bien sur) ainsi qu'un fichier de commande ModelSim `projet.do`. Le design sera totalement synchrone avec l'horloge à 50 MHz.