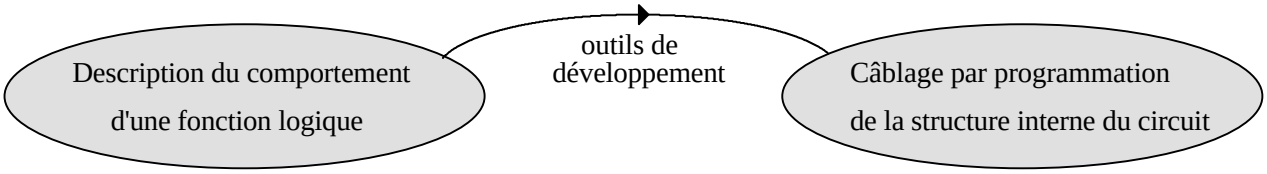
# FONCTIONS LOGIQUES ELEMENTAIRES ET LANGAGES COMPORTEMENTAUX

# INTRODUCTION

Jusqu'à présent, l'aspect expérimental de l'apprentissage de la logique se faisait à travers la découverte des fonctions logiques élémentaires contenues dans les circuits intégrés des familles 74xxx ou CD4000. Les expérimentations se limitaient souvent aux fonctions proposées par les fabricants de ces circuits car la mise en oeuvre de fonctions regroupant plusieurs de ces circuits nécessitait le câblage de nombreuses connexions et éventuellement la réalisation de câblage imprimé.
L'apparition des circuits logiques programmables de type PLD (*Programmable Logic Device*), CPLD (*Complexe PLD*) ou FPGA (*Field Programmable Gate Array)*) a permis de s'affranchir de cette limitation.
En effet, l'utilisateur peut créer, dans ces circuits, toutes les fonctions logiques qu'il souhaite avec comme limitations, la place disponible dans le circuit choisi et / ou la vitesse de fonctionnement de celui-ci.
Les outils de développement mis à la disposition des utilisateurs par les fabricants de ces circuits doivent donc permettre de passer de la description du comportement d'une fonction logique à son câblage dans le circuit et cela de la manière la plus simple possible.




La plupart du temps, la description du comportement des fonctions logiques est faite par l'utilisation de langage dit de "**description comportementale**".
Parmi ceux-ci, on peut citer :

→ Le langage ABEL-HDL (ABEL - ***H**ardware **D**escription **L**anguage*) qui a été créé par la société DATA I/O et utilisé ou imité par quasiment tous les concepteurs d'outils de développement pour ce type de circuit (XABEL pour XILINX, AHDL pour ALTERA, PLD pour ORCAD, XPLA pour PHILIPS, etc..).

→ Le langage VHDL (***V**ery Hight Speed Integrated Circuit, **H**ardware **D**escription **L**anguage*) qui a été créé pour le développement de circuits intégrés logiques complexes. Il doit son succès, essentiellement, à sa standardisation par l'*Institute of Electrical and Electronics Engineers* sous la référence IEEE1076 qui a permis d'en faire un langage unique pour la description, la modélisation, la simulation, la synthèse et la documentation.

→ Le langage VERILOG qui est proche du langage VHDL et qui est aussi très utilisé.

Comme on vient de le voir, VHDL et VERILOG ont des domaines d'utilisation plus vastes que la synthèse de circuits logiques. Ceci présente l'avantage d'un langage unique pour les différents niveaux de conception d'un projet qui va de la description et simulation du cahier des charges, à la description des fonctions logiques synthétisées dans les différents circuits.
Mais ces langages, pour s'adapter à ces différentes utilisations, sont plus complexes et délicats d'usage que le langage ABEL et ses dérivés qui ont une finalité quasi-exclusive de programmation de PLD ou FPGA.

Cette approche, par langages de description comportementale, présente l'avantage, lors de l'apprentissage, de s'intéresser davantage au comportement de la fonction qu'à la manière dont elle est réalisée. De plus, ce comportement peut aisément être simulé avec l'ordinateur, puis expérimenté dans le circuit logique programmable après un simple téléchargement toujours depuis l'ordinateur.
C'est cette approche qui va être exposée ici. Le document va fournir une ou plusieurs descriptions comportementales des différentes fonctions élémentaires de la logique combinatoire et surtout séquentielle.
Chaque fonction sera décrite en langage ABEL. Le résultat fourni par le logiciel sera commenté. Puis on donnera une description de la fonction en langage VHDL.
On a choisi d'utiliser en premier le langage ABEL car il présente les avantages d'être simple d'utilisation, très utilisé en synthèse de circuits logiques et disponible quasiment gratuitement.
En effet, le langage VHDL, bien que très utilisé, est plus difficile à maîtriser et nécessite, en règle générale, des logiciels plus complexes et onéreux.

**Convention d'écriture**
Le contenu des exemples en langage ABEL est donné avec une présentation comme ci-dessous.

```
Langage de description ABEL
```

Pour les exemples en VHDL, on adopte la présentation suivante :

```
Langage de description VHDL
```

***Remarque sur les descriptions en langage ABEL-HDL** : Tous les exemples fournis comportent, à la suite de la description de la fonction logique, une partie intitulée* `Test_vectors` *. Il s'agit de la description en langage ABEL des stimulis permettant de réaliser un test fonctionnel avec l'outils de simulation fourni par DATA I/O.*

***Remarque sur les descriptions en langage VHDL** : La bibliothèque (Library) nommée* `IEEE` *est standardisée et existe de manière quasi-identique dans tous les outils VHDL du marché.*
*Donc, dans les exemples fournis en langage VHDL, les 2 premières lignes de chaque description sont valables quelque soit le logiciel utilisé.*
*Par contre, la troisième ligne donne accès à une bibliothèque qui est propre à chaque éditeur de logiciel. En effet, les concepteurs de circuits logiques programmables sont habitués à la facilité d'écriture du langage ABEL aussi lorsqu'ils ont dû travailler avec VHDL, ils ont réclamé aux éditeurs d'outils VHDL une facilité identique. Ceux-ci ont donc créé des bibliothèques répondant à cette demande (en particulier la surcharge des opérateurs arithmétiques + et -).*

*Les exemples fournis ici ont été compilés avec le logiciel WARP de CYPRESS.*
*Pour pouvoir les compiler avec VIEWLOGIC, il faut remplacer la troisième ligne :*

```
USE work.std_arith.all;
```

*par les 2 lignes suivantes :*

```
LIBRARY synth;
USE synth.vhdlsynth.all;
```

***Remarque générale**: Le présent document ne se veut pas un cours de ABEL ou VHDL, aussi suppose-t-on que le lecteur est déjà initié à l'utilisation de ces langages et des outils de développement associés.*

# LOGIQUE COMBINATOIRE

***FONCTIONS DE BASE***

L'apprentissage de la logique commence, de manière incontournable, par des notions d'algèbre de BOOLE. Celles-ci étant acquises ou en voie de l'être, on aborde les premiers exemples concrets en utilisant des portes logiques élémentaires.
Ceci donne lieu à des expérimentations qui permettent d'étudier les aspects technologiques des circuits logiques (dans ce cas les circuits 74xx ou CD4000 sont bien utiles !!!).
C'est aussi l'occasion d'aborder le premier exemple d'utilisation d'un circuit logique programmable et donc de se familiariser avec le langage ABEL et sa syntaxe. On donne ci-dessous un exemple.

```
MODULE portes

"Inputs
A, B pin;

"Outputs
Y1,Y2,Y3,Y4,Y5,Y6,Y7 pin istype 'com'; out = [Y1..Y7];

Equations
Y1 = A & B;           "And
Y2 = A # B;           "Or
Y3 = A $ B;           "Exclusive or
Y4 = !A;              "Not
Y5 = !(A & B);        "Nand
Y6 = !(A # B);        "Nor
Y7 = !(A $ B);        "Exclusive Nor

Test_vectors
    ([A,B] -> [out])
     [0,0] -> [.X.];
     [0,1] -> [.X.];
     [1,0] -> [.X.];
     [1,1] -> [.X.];
END
```

L'outil de développement va interpréter la description en langage comportemental et proposer une solution pour "câbler" le circuit logique programmable de manière à ce qu'il remplisse la fonction décrite. Parmi les nombreux fichiers intermédiaires qu'il crée, il s'en trouvera qui contiennent la liste des équations booléennes générées. C'est cette liste qui est fournie après chaque exemple.
Dans le cas de l'exemple `PORTES` , on obtient :

```
Y1 = (A & B);
Y2 = (A # B);
Y3 = (A & !B # !A & B);
Y4 = (!A);
Y5 = (!A # !B);
Y6 = (!A & !B);
Y7 = (!A & !B # A & B);
```

On peut constater que, bien entendu, les équations fournies par le logiciel sont conformes à ce que l'on attend, mise à part la réalisation du NAND et du NOR. Ces fonctions n'existant pas, en tant que structure de base, dans les PLD, le logiciel propose des équations utilisant des OU et ET avec des entrées complémentées.

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity PORTES is
port (A,B :in std_logic;
 Y1,Y2,Y3,Y4,Y5,Y6,Y7:out std_logic);
end PORTES;

architecture ARCH_PORTES of PORTES is
begin
  Y1 <= A and B;
  Y2 <= A or B;
  Y3 <= A xor B;
  Y4 <= not A;
  Y5 <= A nand B;
  Y6 <= A nor B;
  Y7 <= not(A xor B);
end ARCH_PORTES;
```

Les équations générées à partir de la description en VHDL sont :

```
y1 = a * b
y2 = b + a
y3 = a * /b + /a * b
y4 = /a
y5 = /a + /b
y6 = /a * /b
y7 = a * b + /a * /b
```

On peut constater qu'elles sont, heureusement, identiques à celles générées à partir de la description en ABEL.

***FONCTIONS COMBINATOIRES COMPLEXES***

L'utilisation de l'algèbre de BOOLE et de ses méthodes de simplification par tableau de KARNAUGH amène à travailler sur des fonctions combinatoires complexes. C'est l'occasion d'utiliser le langage ABEL comme un langage de description de comportement. Voici 4 exemples classiques : le décodeur 7 segments hexadécimal, le décodeur 7 segments décimal, le démultiplexeur, le comparateur de magnitude pour lesquels il sera intéressant d'effectuer aussi la détermination des équations "à la main" (pour les plus simples) afin de vérifier la similitude des solutions.

**Décodeur 7 segments hexadécimal**

```
MODULE dec7seg3

"Decodeur 7 segments (de 0 a 9)
"Convention de notation des segments
"       ___
"     | 0 |
"   5 |___| 1
"     | 6 |
"   4 |___| 2
"       3

"Inputs
D3..D0 pin; DEC = [D3..D0];

"Outputs
S0..S6 pin istype 'com'; SEG = [S0..S6];

ON,OFF = 1,0;"Afficheur LED a cathode commune

Equations
     WHEN (DEC ==  0) THEN SEG = [ ON, ON, ON, ON, ON, ON,OFF];
ELSE WHEN (DEC ==  1) THEN SEG = [OFF, ON, ON,OFF,OFF,OFF,OFF];
ELSE WHEN (DEC ==  2) THEN SEG = [ ON, ON,OFF, ON, ON,OFF, ON];
ELSE WHEN (DEC ==  3) THEN SEG = [ ON, ON, ON, ON,OFF,OFF, ON];
ELSE WHEN (DEC ==  4) THEN SEG = [OFF, ON, ON,OFF,OFF, ON, ON];
ELSE WHEN (DEC ==  5) THEN SEG = [ ON,OFF, ON, ON,OFF, ON, ON];
ELSE WHEN (DEC ==  6) THEN SEG = [ ON,OFF, ON, ON, ON, ON, ON];
ELSE WHEN (DEC ==  7) THEN SEG = [ ON, ON, ON,OFF,OFF,OFF,OFF];
ELSE WHEN (DEC ==  8) THEN SEG = [ ON, ON, ON, ON, ON, ON, ON];
ELSE WHEN (DEC ==  9) THEN SEG = [ ON, ON, ON, ON,OFF, ON, ON];
ELSE WHEN (DEC == 10) THEN SEG = [ ON, ON, ON,OFF, ON, ON, ON];
ELSE WHEN (DEC == 11) THEN SEG = [OFF,OFF, ON, ON, ON, ON, ON];
ELSE WHEN (DEC == 12) THEN SEG = [ ON,OFF,OFF, ON, ON, ON,OFF];
ELSE WHEN (DEC == 13) THEN SEG = [OFF, ON, ON, ON, ON,OFF, ON];
ELSE WHEN (DEC == 14) THEN SEG = [ ON,OFF,OFF, ON, ON, ON, ON];
                      ELSE SEG = [ ON,OFF,OFF,OFF, ON, ON, ON];

test_vectors
(DEC -> SEG)
@const n = 0;
@repeat 16 {n -> .X.; @const n = n+1;}

END
```

Les équations fournies par le logiciel sont :

```
S0 = (D0 & D2 & !D3 # !D1 & !D2 & D3 # !D0 & !D2 # D1 & D2 # D1 & !D3 # !D0 & D3);
S1 = (!D2 & !D3 # D0 & D1 & !D3 # !D0 & !D2 # D0 & !D1 & D3 # !D0 & !D1 & !D3);
S2 = (!D1 & !D2 # D0 & !D2 # D0 & !D1 # D2 & !D3 # !D2 & D3);
S3 = (!D0 & !D2 & !D3 # D0 & D1 & !D2 # !D0 & D1 & D2 # !D1 & D3 # D0 & !D1 & D2);
S4 = (!D0 & !D2 # !D0 & D1 # D2 & D3 # D1 & D3);
S5 = (!D1 & D2 & !D3 # !D0 & D2 # !D0 & !D1 # !D2 & D3 # D1 & D3);
S6 = (!D1 & D2 & !D3 # D0 & D3 # !D0 & D1 # D1 & !D2 # !D2 & D3);
```

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous. Le choix d'un afficheur Anode ou Cathode commune se fera en complémentant ou non la sortie SEG. La méthode utilisée en ABEL conduirait à une écriture peu lisible dans le cas du VHDL.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DEC7SEG3 is
port (DEC :in std_logic_vector(3 downto 0);
    SEG:out std_logic_vector(0 to 6));
end DEC7SEG3;

architecture ARCH_DEC7SEG3 of DEC7SEG3 is
begin
SEG <= "1111110" WHEN DEC = 0
  ELSE "0110000" WHEN DEC = 1
  ELSE "1101101" WHEN DEC = 2
  ELSE "1111001" WHEN DEC = 3
  ELSE "0110011" WHEN DEC = 4
  ELSE "1011011" WHEN DEC = 5
  ELSE "1011111" WHEN DEC = 6
  ELSE "1110000" WHEN DEC = 7
  ELSE "1111111" WHEN DEC = 8
  ELSE "1111011" WHEN DEC = 9
  ELSE "1110111" WHEN DEC = 10
  ELSE "0011111" WHEN DEC = 11
  ELSE "1001110" WHEN DEC = 12
  ELSE "0111101" WHEN DEC = 13
  ELSE "1001111" WHEN DEC = 14
  ELSE "1000111";
end ARCH_DEC7SEG3;
```

Les équations générées sont :

```
seg_0 = /dec_3 * dec_2 * dec_0 + dec_3 * /dec_2 * /dec_1
        + /dec_2 * /dec_0 + dec_3 * /dec_0 + dec_2 * dec_1 + /dec_3 * dec_1
seg_1 = /dec_3 * dec_1 * dec_0 + dec_3 * /dec_1 * dec_0 + /dec_3 * /dec_1 * /dec_0
        + /dec_2 * /dec_0 + /dec_2 * /dec_1
seg_2 = /dec_2 * /dec_1 + /dec_2 * dec_0 + /dec_3 * dec_2
        + /dec_1 * dec_0 + dec_3 * /dec_2
seg_3 = dec_2 * dec_1 * /dec_0 + /dec_3 * /dec_2 * /dec_0 + /dec_2 * dec_1 * dec_0
        + dec_2 * /dec_1 * dec_0 + dec_3 * /dec_1
seg_4 = /dec_2 * /dec_0 + dec_3 * dec_2 + dec_1 * /dec_0 + dec_3 * dec_1
seg_5 = /dec_3 * dec_2 * /dec_1 + dec_3 * dec_1 + dec_3 * /dec_2
        + /dec_1 * /dec_0 + dec_2 * /dec_0
seg_6 = /dec_3 * dec_2 * /dec_1 + dec_1 * /dec_0 + dec_3 * /dec_2
        + dec_3 * dec_0 + /dec_2 * dec_1
```

On peut constater que ces équations sont identiques (à part une variante de groupement pour le segment 1) qu'elles soient générées à partir d'une description en ABEL ou en VHDL.

**Décodeur 7 segments décimal**

```
MODULE dec7seg4

"Decodeur 7 segments (de 0 a 9)
"Convention de notation des segments
"       ___
"     | 0 |
"   5 |___| 1
"     | 6 |
"   4 |___| 2
"       3

"Inputs
D3..D0 pin; DEC = [D3..D0];

"Outputs
S0..S6 pin istype 'com'; SEG = [S0..S6];

ON,OFF = 1,0;"Afficheur LED a cathode commune

@DCSET "autorise les simplifications logiques pour les etats non utilises
truth_table
(DEC -> [ S0, S1, S2, S3, S4, S5, S6])
  0  -> [ ON, ON, ON, ON, ON, ON,OFF];
  1  -> [OFF, ON, ON,OFF,OFF,OFF,OFF];
  2  -> [ ON, ON,OFF, ON, ON,OFF, ON];
  3  -> [ ON, ON, ON, ON,OFF,OFF, ON];
  4  -> [OFF, ON, ON,OFF,OFF, ON, ON];
  5  -> [ ON,OFF, ON, ON,OFF, ON, ON];
  6  -> [ ON,OFF, ON, ON, ON, ON, ON];
  7  -> [ ON, ON, ON,OFF,OFF,OFF,OFF];
  8  -> [ ON, ON, ON, ON, ON, ON, ON];
  9  -> [ ON, ON, ON, ON,OFF, ON, ON];

test_vectors
(DEC -> SEG)
@const n = 0;
@repeat 16 {n -> .X.; @const n = n+1;}

END
```

On peut remarquer que, grâce à l'utilisation de la commande `@DCSET` (*Don't Care*), le logiciel a fourni des équations simplifiées qui utilisent au mieux les cas non définis.

```
S0 = (!D2 & !D0 # D3 # D1 # D2 & D0);
S1 = (D1 & D0 # !D2 # !D1 & !D0);
S2 = (!D1 # D2 # D0);
S3 = (D1 & !D0 # D3 # D2 & !D1 & D0 # !D2 & D1 # !D2 & !D0);
S4 = (!D2 & !D0 # D1 & !D0);
S5 = (D2 & !D0 # D3 # D2 & !D1 # !D1 & !D0);
S6 = (D2 & !D0 # D3 # D2 & !D1 # !D2 & D1);
```

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous. On notera la méthode utilisée pour permettre les simplifications des cas non définis.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DEC7SEG4 is
port (DEC :in std_logic_vector(3 downto 0);
    SEG:out std_logic_vector(0 to 6));
end DEC7SEG4;

architecture ARCH_DEC7SEG4 of DEC7SEG4 is
begin
SEG <= "1111110" WHEN DEC = 0
  ELSE "0110000" WHEN DEC = 1
  ELSE "1101101" WHEN DEC = 2
  ELSE "1111001" WHEN DEC = 3
  ELSE "0110011" WHEN DEC = 4
  ELSE "1011011" WHEN DEC = 5
  ELSE "1011111" WHEN DEC = 6
  ELSE "1110000" WHEN DEC = 7
  ELSE "1111111" WHEN DEC = 8
  ELSE "1111011" WHEN DEC = 9
  ELSE "-------";
end ARCH_DEC7SEG4;
```

Les équations générées sont :

```
seg_0 = /dec_2 * /dec_0 + dec_2 * dec_0 + dec_1 + dec_3
seg_1 = /dec_1 * /dec_0 + dec_1 * dec_0 + /dec_2
seg_2 = dec_0 + /dec_1 + dec_2
seg_3 = dec_2 * /dec_1 * dec_0 + dec_1 * /dec_0
        + /dec_2 * /dec_0 + /dec_2 * dec_1 + dec_3
seg_4 = dec_1 * /dec_0 + /dec_2 * /dec_0
seg_5 = /dec_1 * /dec_0 + dec_2 * /dec_0 + dec_2 * /dec_1 + dec_3
seg_6 = dec_1 * /dec_0 + /dec_2 * dec_1 + dec_2 * /dec_1 + dec_3
```

On retrouve ici encore une petite différence dans un regroupement (segment 6). Autrement les équations sont parfaitement identiques.

**Démultiplexeur 3 vers 8 à sorties actives à 0 et à entrées de validation**

Le fonctionnement de ce décodeur est résumé dans la table de vérité donnée ci-dessous.

| G1 | G2 | B2 | B1 | B0 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | X | X | X | H | H | H | H | H | H | H | H |
| 0 | 1 | X | X | X | H | H | H | H | H | H | H | H |
| 1 | 1 | X | X | X | H | H | H | H | H | H | H | H |
| 1 | 0 | 0 | 0 | 0 | H | H | H | H | H | H | H | L |
| 1 | 0 | 0 | 0 | 1 | H | H | H | H | H | H | L | H |
| 1 | 0 | 0 | 1 | 0 | H | H | H | H | H | L | H | H |
| 1 | 0 | 0 | 1 | 1 | H | H | H | H | L | H | H | H |
| 1 | 0 | 1 | 0 | 0 | H | H | H | L | H | H | H | H |
| 1 | 0 | 1 | 0 | 1 | H | H | L | H | H | H | H | H |
| 1 | 0 | 1 | 1 | 0 | H | L | H | H | H | H | H | H |
| 1 | 0 | 1 | 1 | 1 | L | H | H | H | H | H | H | H |

```
MODULE decod3_8

"Inputs
B2,B1,B0 pin; B = [B2..B0];
G1,G2 pin;"entrees de validation

"Outputs
D7..D0 pin istype 'com'; D = [D7..D0];

Equations
     WHEN !((G1 == 1) & (G2 == 0)) THEN D = [1,1,1,1,1,1,1,1]
ELSE WHEN (B == 0) THEN D = [1,1,1,1,1,1,1,0]
ELSE WHEN (B == 1) THEN D = [1,1,1,1,1,1,0,1]
ELSE WHEN (B == 2) THEN D = [1,1,1,1,1,0,1,1]
ELSE WHEN (B == 3) THEN D = [1,1,1,1,0,1,1,1]
ELSE WHEN (B == 4) THEN D = [1,1,1,0,1,1,1,1]
ELSE WHEN (B == 5) THEN D = [1,1,0,1,1,1,1,1]
ELSE WHEN (B == 6) THEN D = [1,0,1,1,1,1,1,1]
ELSE WHEN (B == 7) THEN D = [0,1,1,1,1,1,1,1]

test_vectors
([G1,G2,B] -> D)
 [0,0,.X.] -> .X.;
 [0,1,.X.] -> .X.;
 [1,1,.X.] -> .X.;
@const n=0;
@repeat 8 {[1,0,n] -> .X.; @const n = n+1;}

END
```

On aurait pu aussi utiliser, pour la partie `equations`, l'écriture plus rapide (et plus satisfaisante pour le programmeur) donnée ci-dessous, mais celle-ci devient nettement moins lisible et il est donc préférable de l'éviter.

```
Equations
@const n=0;
@repeat 8 {!D[n] = (B == n) & G1 & !G2; @const n=n+1;}
```

Ici le logiciel utilise les équations complémentées qui sont plus simples à implanter dans la structure d'un PLD.

```
!D7 = (G1 & !G2 & B0 & B1 & B2);
!D6 = (G1 & !G2 & !B0 & B1 & B2);
!D5 = (G1 & !G2 & B0 & !B1 & B2);
!D4 = (G1 & !G2 & !B0 & !B1 & B2);
!D3 = (G1 & !G2 & B0 & B1 & !B2);
!D2 = (G1 & !G2 & !B0 & B1 & !B2);
!D1 = (G1 & !G2 & B0 & !B1 & !B2);
!D0 = (G1 & !G2 & !B0 & !B1 & !B2);
```

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous. Aucunes simplifications étant possibles, les équations ne peuvent être qu'identiques.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DECOD3_8 is
port (B :in std_logic_vector(2 downto 0);
      G1,G2 :in std_logic;
      D :out std_logic_vector(7 downto 0));
end DECOD3_8;

architecture ARCH_DECOD3_8 of DECOD3_8 is
begin
  D <= "11111111" when not (G1='1' and G2='0')
  else "11111110" when B=0
  else "11111101" when B=1
  else "11111011" when B=2
  else "11110111" when B=3
  else "11101111" when B=4
  else "11011111" when B=5
  else "10111111" when B=6
  else "01111111" when B=7
  else "11111111";
end ARCH_DECOD3_8;
```

**Comparateur de magnitude 4 bits**

Cette fonction réalise la comparaison de 2 nombres binaires de 4 bits. Trois sorties (actives à l'état logique 1) permettent de fournir le résultat de la comparaison. Une sortie pour "**supérieur**", une autre pour "**inférieur**" et enfin une troisième pour "**égal**".

```
MODULE comp4bit

"Inputs
A3..A0 pin; A = [A3..A0];
B3..B0 pin; B = [B3..B0];

"Outputs
plus,moins,egal pin istype 'com';

Equations
plus = (A > B);
moins = (A < B);
egal = (A == B);

Test_vectors
([A,B] -> [plus,moins,egal]);
@const n=0;
@repeat 16 {@const m=0;
            @repeat 16 {[n,m] -> .X.; @const m = m+1;}
             @const n = n+1;}

END
```

Bien entendu la recherche "à la main" est un travail énorme et de peu d'intérêt (tableau de KARNAUGH 16 x 16 !!!). Les équations générées sont conséquentes (équations contenants jusqu'à 16 termes de OU et chaque terme contenant 16 termes de ET (ou terme de produit : *Product Term*). Il faudra vérifier que les possibilités combinatoires du circuit cible peuvent accueillir de telles équations ou utiliser la commande " `@carry n` " qui limite à **n** le nombre de termes des équations.

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity COMP4BIT is
port (A,B :in std_logic_vector(3 downto 0);
      PLUS,MOINS,EGAL :out std_logic);
end COMP4BIT;

architecture ARCH_COMP4BIT of COMP4BIT is
begin
  PLUS <= '1' WHEN A > B ELSE '0';
  MOINS <= '1' WHEN A < B ELSE '0';
  EGAL <= '1' WHEN A = B ELSE '0';
end ARCH_COMP4BIT;
```

# LOGIQUE SEQUENTIELLE

La logique séquentielle utilise des fonctions logiques dans lesquelles le temps est une variable essentielle.
On distingue :

→ la logique séquentielle **asynchrone** dans laquelle les changements d'état des sorties peuvent se produire à des instants difficilement prédictibles (retards formés par les basculement d'un nombre souvent inconnu de fonctions),

→ la logique séquentielle **synchrone** dans laquelle les changements d'état des sorties se produisent tous à des instants quasiment connus (un temps de retard après un front actif de l'horloge).

Pour des raisons de simplification de câblage, on utilisait la logique asynchrone pour réaliser des fonctions complexes mettant en oeuvre de nombreux boîtiers.
Mais du fait de la difficulté qu'il y a à prévoir les instants de changement d'état des signaux, il apparaîssait souvent des impulsions indésirables (*Glitch*) pouvant générer des résultats non conformes aux attentes. D'où des difficultés souvent importantes dans la mise au point.
L'apparition des circuits logiques programmables permet maintenant de réaliser ces mêmes fonctions logiques complexes en fonctionnement complètement synchrone. La bonne prédictibilité des instants de changement d'état permet de réduire considérablement les aléas de fonctionnement.
*De l'avis unamine de tous les utilisateurs expérimentés de ce type de circuit, il faut utiliser impérativement des méthodes de synthèse synchrone.*

*La synthèse de fonctions logiques doit être effectuée en privilégiant un fonctionnement **synchrone**.*

Néanmoins, dans le cadre d'un apprentissage, il est apparaît nécessaire d'étudier aussi quelques fonctions logiques asynchrones.

## *FONCTIONS DE BASE*

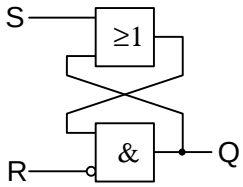
### *Fonctions asynchrones*

**Bascule RS**

```
MODULE rs_async

"Inputs
R,S pin;

"Outputs
Q pin istype 'com';

Equations
     WHEN (R == 1) & (S == 0) THEN Q = 0;
ELSE WHEN (R == 0) & (S == 1) THEN Q = 1;
ELSE WHEN (R == 0) & (S == 0) THEN Q = Q;
                              ELSE Q = .X.;

Test_vectors
([R,S] -> [Q])
[0,0] -> .X.;
[0,1] -> .X.;
[0,0] -> .X.;
[1,0] -> .X.;
[0,0] -> .X.;
[1,1] -> .X.;
[0,0] -> .X.;

END
```

L'équation générée est : `Q = (!R & S # !R & Q);`
On peut aussi l'écrire : `Q = !R & (S # Q);`

La structure remplissant cette fonction utilise une porte OU et une porte ET ainsi qu'une complémentation comme sur le schéma ci-dessous.



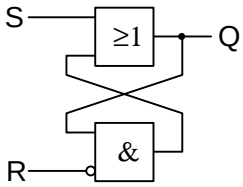

On peut remarquer que la fonction bascule RS asynchrone est bien remplie et que l'état R = S = 1 non connu à priori a été fixé à Q = 0 par le logiciel, ce que l'expérimentation pourra permettre de vérifier.

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity RS_ASYNC is
port (R,S :in std_logic;
      Q :out std_logic);
end RS_ASYNC;

architecture ARCH_RS_ASYNC of RS_ASYNC is
signal X :std_logic;
begin
  X <= '0' when R='1' and S='0'
  else '1' when R='0' and S='1'
  else  X  when R='0' and S='0'
  else '-';
  Q <= X;
end ARCH_RS_ASYNC;
```

L'équation générée est : `q = /r * q + s`

Cette fois le logiciel a choisi Q = 1 pour l'état R = S = 1. Ceci correspond à raccorder la borne Q à la sortie de la porte OU comme sur le schéma ci-dessous.

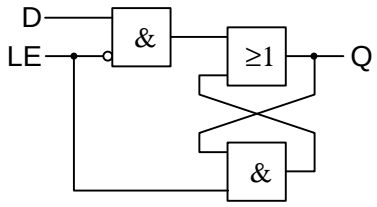


**Verrou (latch)**

```
MODULE verrou

"Inputs
D, LE pin;"D donnee a verrouiller ; LE autorisation de verrouillage

"Outputs
Q pin istype 'com';

equations
WHEN (LE == 0) THEN Q = D; ELSE Q = Q;

test_vectors
([LE,D] -> Q)
[0, 1] -> .X.;
[0, 0] -> .X.;
[0, 1] -> .X.;
[1, 1] -> .X.;
[1, 0] -> .X.;
[0, 0] -> .X.;
[1, 1] -> .X.;

END
```

L'équation générée ci-dessous se traduit par le schéma structurel fourni. On y voit apparaître la structure mémoire vue à l'exemple précédent (cellule OU / ET rebouclée) et un dispositif d'aiguillage commandé par LE.

```
Q = (D & !LE # LE & Q);
```




Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity VERROU is
port (LE,D :in std_logic;
      Q :out std_logic);
end VERROU;

architecture ARCH_VERROU of VERROU is
signal X :std_logic;
begin
  X <= D when LE='0' else X;
  Q <= X;
end ARCH_VERROU;
```

L'équation générée est : `q = le * q + /le * d`
Elle est identique à celle issue de la description ABEL.
Il est à noter que certains circuits logiques programmables possèdent des cellules logiques contenant des bascules configurables en `Latch` .

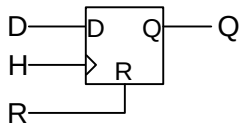
### ***Fonctions synchrones***

Toutes les fonctions synchrones sont construites à partir de bascules D maître-esclave car c'est l'élément de base de logique séquentielle qui est présent physiquement dans chaque cellule de quasiment tous les circuits logiques programmables (PLD, CPLD et FPGA).

**Bascule D**

```
MODULE d_ff

"Inputs
R,D,H pin;"R est une RAZ asynchrone

"Outputs
Q pin istype 'reg';

equations
Q.clk = H;
Q.aclr = R;
Q := D;

test_vectors
([H,R,D] -> [Q])
[.C.,0,1] -> .X.;
[ 0 ,0,0] -> .X.;
[ 0 ,1,0] -> .X.;
[.C.,0,1] -> .X.;
[.C.,0,0] -> .X.;

END
```

Le schéma que l'on obtient est celui de la cellule séquentielle de base des circuits de type PLD muni de sa remise à zéro asynchrone.



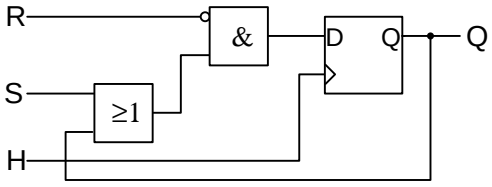

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity D_FF is
port (H,R,D :in std_logic;
      Q :out std_logic);
end D_FF;

architecture ARCH_D_FF of D_FF is
signal X :std_logic;
begin
  process(H,R)
   begin
   if R='1' then X <= '0';
   elsif (H'event and H='1') then X <= D;
   end if;
  end process;
  Q <= X;
end ARCH_D_FF;
```

**Bascule RS synchrone**

```
MODULE rs_sync

"Inputs
R,S,H pin;

"Outputs
Q pin istype 'reg';

equations
Q.clk = H;
     WHEN (R == 1) & (S == 0) THEN Q := 0;
ELSE WHEN (R == 0) & (S == 1) THEN Q := 1;
ELSE WHEN (R == 0) & (S == 0) THEN Q := Q;
ELSE WHEN (R == 1) & (S == 1) THEN Q := .X.;

test_vectors
([H,R,S] -> [Q])
[.C.,0,0] -> .X.;
[ 0 ,0,1] -> .X.;
[.C.,0,1] -> .X.;
[.C.,0,0] -> .X.;
[ 0 ,0,1] -> .X.;
[ 0 ,0,0] -> .X.;
[.C.,1,0] -> .X.;
[.C.,0,0] -> .X.;
[.C.,1,1] -> .X.;
[.C.,0,0] -> .X.;

END
```

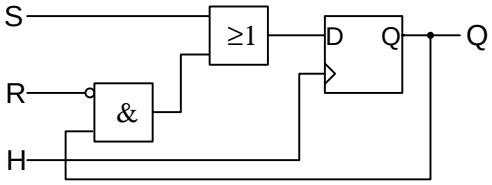
Le logiciel fournit l'équation suivante : `Q := (!R & S  # !R & Q);`
On peut remarquer qu'il s'agit de la même équation que pour la bascule RS asynchrone. La seule différence réside dans l'utilisation de l'affectation `:=` pour la RS synchrone.
On aurait pu aussi écrire cette équation de la manière suivante : `Q.D = (!R & S  # !R & Q.Q);` les attributs `.D` et `.Q` faisant référence à l'entrée `D` et à la sortie `Q` d'une bascule D maître-esclave.
Comme pour la version asynchrone, on écrit l'équation de la manière suivante : `Q := !R & (S # Q);`
On obtient un schéma structurel tel que celui qui est donné ci-dessous.



Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity RS_SYNC is
port (H,R,S :in std_logic;
      Q :out std_logic);
end RS_SYNC;

architecture ARCH_RS_SYNC of RS_SYNC is
signal X :std_logic;
begin
  process(H)
   begin
   if (H'event and H='1') then
     if R='1' and S='0' then X <= '0';
     elsif R='0' and S='1' then X <= '1';
     elsif R='0' and S='0' then X <= X;
     else X <= '-';
     end if;
  end if;
  end process;
  Q <= X;
end ARCH_RS_SYNC;
```

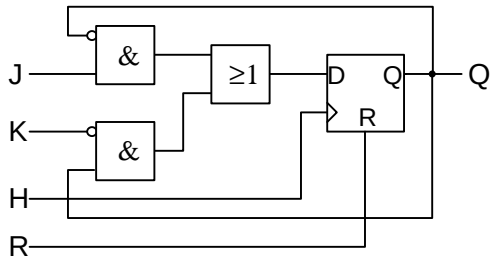
Comme pour la bascule RS asynchrone, le logiciel a choisi Q = 1 pour l'état R = S = 1. Ceci correspond au schéma suivant.



**Bascule JK**

```
MODULE jk_ff

"Inputs
R,J,K,H pin;"R est une RAZ asynchrone

"Outputs
Q pin istype 'reg';

equations
Q.clk = H;
Q.aclr = R;
     WHEN (K == 1) & (J == 0) THEN Q := 0;
ELSE WHEN (K == 0) & (J == 1) THEN Q := 1;
ELSE WHEN (K == 0) & (J == 0) THEN Q := Q;
                              ELSE Q := !Q;

test_vectors
([H,R,J,K] -> [Q])
[.C.,0,1,0] -> .X.;
[ 0 ,0,0,0] -> .X.;
[ 0 ,1,0,0] -> .X.;
[.C.,0,1,0] -> .X.;
[.C.,0,0,0] -> .X.;
[ 0 ,0,0,1] -> .X.;
[ 0 ,0,0,0] -> .X.;
[.C.,0,0,1] -> .X.;
[.C.,0,0,0] -> .X.;
[.C.,0,1,1] -> .X.;
[.C.,0,1,1] -> .X.;

END
```

L'équation obtenue est la suivante : `Q := (J & !Q # !K & Q);`
Elle se traduit par le schéma structurel donné ci-dessous dans lequel on voit apparaître une entrée R de remise à zéro asynchrone (celle de la bascule D).



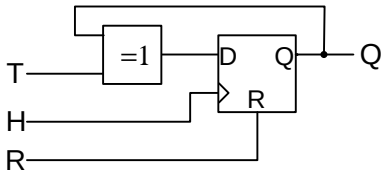
Une description en langage VHDL donnant un résultat identique est fournie ci-dessous. Elle donne aussi des équations identiques.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity JK_FF is
port (H,R,J,K :in std_logic;
      Q :out std_logic);
end JK_FF;

architecture ARCH_JK_FF of JK_FF is
signal X :std_logic;
begin
  process(H,R)
   begin
   if R='1' then X <= '0';
   elsif (H'event and H='1') then
     if K='1' and J='0' then X <= '0';
     elsif K='0' and J='1' then X <= '1';
     elsif K='1' and J='1' then X <= not X;
     else X <= X;
     end if;
   end if;
  end process;
  Q <= X;
end ARCH_JK_FF;
```

**Bascule T**

```
MODULE t_ff

"Inputs
R,T,H pin;"R est une RAZ asynchrone

"Outputs
Q pin istype 'reg';

equations
Q.clk = H;
Q.aclr = R;
WHEN (T == 1) THEN Q := !Q ELSE Q := Q;

test_vectors
([H,R,T] -> [Q])
[.C.,0,1] -> .X.;
[ 0 ,0,0] -> .X.;
[ 0 ,1,0] -> .X.;
[.C.,0,1] -> .X.;
[.C.,0,1] -> .X.;
[.C.,0,1] -> .X.;
[.C.,0,0] -> .X.;

END
```

Les équations obtenues se traduisent par le schéma donné ci-contre.
On voit clairement apparaître le rôle du OU exclusif. Si `T` vaut 1, en `D` on obtient `!Q` , sinon, on obtient `Q`.




Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity T_FF is
port (H,R,T :in std_logic;
      Q :out std_logic);
end T_FF;

architecture ARCH_T_FF of T_FF is
signal X :std_logic;
begin
  process(H,R)
   begin
   if R='1' then X <= '0';
   elsif (H'event and H='1') then
     if T='1' then X <= not X;
     else X <= X;
     end if;
   end if;
  end process;
  Q <= X;
end ARCH_T_FF;
```

***FONCTIONS COMPLEXES***

Les circuits logiques programmables ont été conçus pour des fonctionnements synchrones.
En effet, le câblage des entrées "horloge" des différentes bascules D formant les ressources séquentielles est prévu pour se raccorder très simplement au signal d'horloge distribué sur le circuit.
La création de fonctions complexes asynchrones n'y est donc pas naturelle. De plus, sachant que de telles structures risquent de poser de nombreux problèmes lors de leur assemblage avec d'autres fonctions, tout contribue à ne plus en concevoir.
Donc, ici, on n'étudiera que des structures synchrones (mis à part la remise à zéro asynchrone générale).
En mode totalement synchrone, toutes les entrées "horloge" sont reliées au même signal de cadencement. L'association, la mise en cascade des différentes fonctions se fait donc grâce à des entrées "autorisation d'horloge" (EN pour *Enable*) et des sorties qui définissent des fenêtres d'autorisation d'horloge. Un exemple est donné plus loin avec la mise en cascade de plusieurs compteurs.

### *Fonctions de comptage*

**Compteur synchrone**

```
MODULE compt_4

"Inputs
H,R pin;

"Outputs
Q3..Q0 pin istype 'reg';
Q = [Q3..Q0];

equations
Q.clk = H;
Q.aclr = R;
Q := Q + 1;

test_vectors
([H,R] -> Q)
@repeat 4  {[.C.,0] -> .X.;}
        [ 0 ,1] -> .X.;
@repeat 20 {[.C.,0] -> .X.;}

END
```

Le logiciel génére les équations suivantes :

```
Q3 := (!Q3 & Q2 & Q1 & Q0 # Q3 & !Q1 # Q3 & !Q2 # Q3 & !Q0);
Q2 := (!Q2 & Q1 & Q0 # Q2 & !Q1 # Q2 & !Q0);
Q1 := (Q1 & !Q0 # !Q1 & Q0);
Q0 := (!Q0);
```

Si l'on utilise l'opérateur OU exclusif, on obtient :

```
Q3 := !Q3 & (Q2 & Q1 & Q0) # Q3 & !(Q2 & Q1 & Q0);
Q2 := !Q2 & (Q1 & Q0) # Q2 & !(Q1 & Q0);
Q1 := !Q1 & Q0 # Q1 & !Q0;
Q0 := !Q0;
```

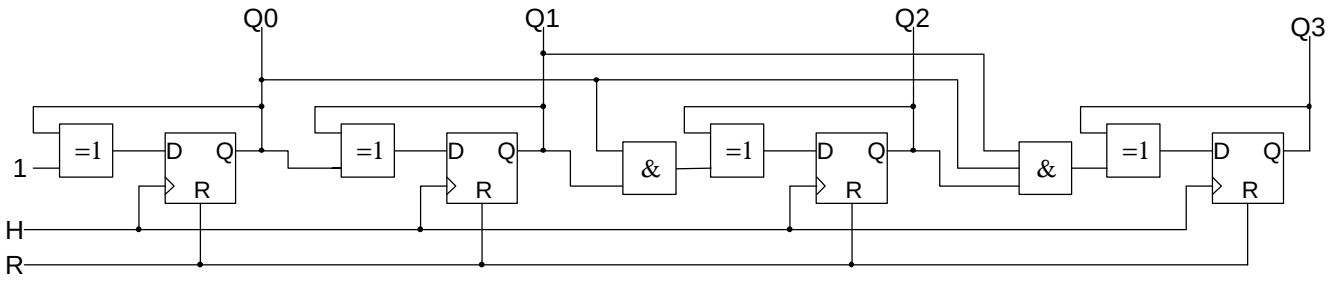
Soit :

```
Q3 := Q3 $ (Q2 & Q1 & Q0);
Q2 := Q2 $ (Q1 & Q0);
Q1 := Q1 $ Q0;
Q0 := Q0 $ 1;
```

On voit apparaître une relation utilisable à l'ordre **n** :

```
Qn := Qn $ (Qn-1 & Qn-2 & ... & Q1 & Q0);
```

Le schéma structurel d'un tel compteur est donné ci-dessous.




Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity COMPT_4 is
port (H,R :in std_logic;
      Q :out std_logic_vector(3 downto 0));
end COMPT_4;

architecture ARCH_COMPT_4 of COMPT_4 is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then X <= X + 1;
   end if;
  end process;
  Q <= X;
end ARCH_COMPT_4;
```

Les équations générées sont strictement identiques à celles de la description en ABEL.

Certains circuits logiques programmables ont des ressources séquentielles formées de bascules qui peuvent se configurer en bascule D maître-esclave, en bascule T ou en verrou. Si le précédent compteur est implanté dans un tel circuit, le logiciel choisira de construire le compteur avec des bascules T.
Dans ce cas, les équations seront :

```
T3 = Q2 & Q1 & Q0;
T2 = Q1 & Q0;
T1 = Q0;
T0 = 1;
```

T3 représente l'entrée T de la 3ème bascule T et Q2 la sortie Q de la 2ème bascule T.
La relation utilisable à l'ordre **n** est :

```
Tn = Qn-1 & Qn-2 & ... & Q1 & Q0;
```

Le schéma issu de ces équations est identique à celui qui a été fourni ci-dessus. En effet, dans ce schéma, on peut voir apparaître la structure de bascule T (bascule D avec OU exclusif).

**Décompteur synchrone**

```
MODULE decompt4

"Inputs
H,R pin;

"Outputs
Q3..Q0 pin istype 'reg';
Q = [Q3..Q0];

equations
Q.clk = H;
Q.aclr = R;
Q := Q - 1;

test_vectors
([H,R] -> Q)
@repeat 4  {[.C.,0] -> .X.;}
        [ 0 ,1] -> .X.;
@repeat 20 {[.C.,0] -> .X.;}

END
```

Les équations générées sont :

```
Q3 := (!Q3 & !Q2 & !Q1 & !Q0 # Q3 & Q1 # Q3 & Q2 # Q3 & Q0);
Q2 := (!Q2 & !Q1 & !Q0 # Q2 & Q1  # Q2 & Q0);
Q1 := (!Q1 & !Q0 # Q1 & Q0);
Q0 := (!Q0);
```

Comme pour le compteur, si l'on utilise l'opérateur OU exclusif, on obtient :

```
Q3 := !Q3 & (!Q2 & !Q1 & !Q0) # Q3 & !(!Q2 & !Q1 & !Q0);
Q2 := !Q2 & (!Q1 & !Q0) # Q2 & !(!Q1 & !Q0);
Q1 := !Q1 & !Q0 # Q1 & !!Q0;
Q0 := !Q0;
```

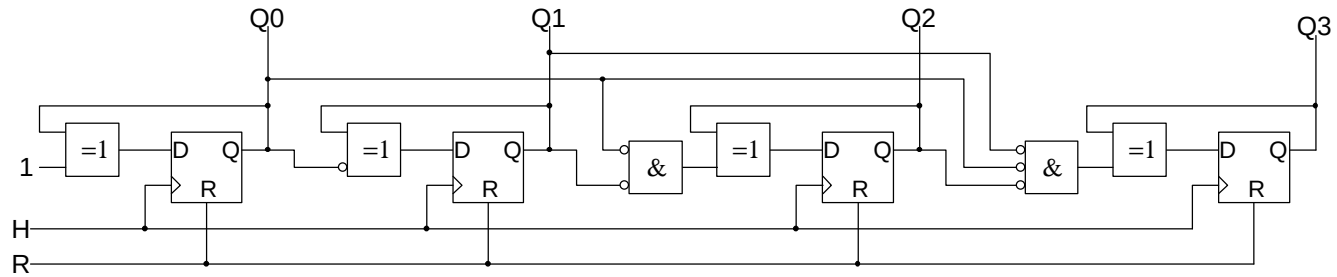
Soit :

```
Q3 := Q3 $ (!Q2 & !Q1 & !Q0);
Q2 := Q2 $ (!Q1 & !Q0);
Q1 := Q1 $ !Q0;
Q0 := Q0 $ 1;
```

On voit apparaître une relation utilisable à l'ordre **n** :

```
Qn := Qn $ (!Qn-1 & !Qn-2 & ... & !Q1 & !Q0);
```

Le schéma peut se déduire aisément du précédent exemple.




On voit apparaître pour chaque étage une structure identique (Bascule D, OU exclusif) qui est équivalente à une bascule T.
On peut donc en déduire la relation utilisable à l'ordre **n**, si l'on utilise des bascules T :

```
Tn = !Qn-1 & !Qn-2 & ... & !Q1 & !Q0;
```

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DECOMPT4 is
port (H,R :in std_logic;
      Q :out std_logic_vector(3 downto 0));
end DECOMPT4;

architecture ARCH_DECOMPT4 of DECOMPT4 is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then X <= X - 1;
   end if;
  end process;
  Q <= X;
end ARCH_DECOMPT4;
```

Les équations générées sont identiques à celles issues de la description en ABEL.

**Compteur décompteur synchrone**

```
MODULE updown4

"Inputs
H,R,UD pin;"UD est la commande de comptage (1) ou decomptage (0)

"Outputs
Q3..Q0 pin istype 'reg'; Q = [Q3..Q0];

equations
Q.clk = H;
Q.aclr = R;
WHEN (UD == 0) THEN Q := Q - 1; ELSE Q := Q + 1;

test_vectors
([H,R,UD] -> Q)
@repeat 4  {[.C.,0,1] -> .X.;}
        [ 0 ,1,1] -> .X.;
@repeat 20 {[.C.,0,1] -> .X.;}
@repeat 20 {[.C.,0,0] -> .X.;}
END
```

Les équations générées sont :

```
Q3 := (Q3 & Q0 & !UD # Q3 & !Q1 & UD # Q3 & Q2 & !Q0 # Q3 & !Q2 & Q1
     # !Q3 & !Q2 & !Q1 & !Q0 & !UD # !Q3 & Q2 & Q1 & Q0 & UD);
Q2 := (Q2 & Q0 & !UD # Q2 & !Q1 & UD # Q2 & Q1 & !Q0
     # !Q2 & !Q1 & !Q0 & !UD # !Q2 & Q1 & Q0 & UD);
Q1 := (!Q1 & !Q0 & !UD # Q1 & Q0 & !UD # Q1 & !Q0 & UD # !Q1 & Q0 & UD);
Q0 := (!Q0);
```

L'équation utilisable à l'ordre **n** aurait dû être :

```
Qn :=   UD & [Qn $ (Qn-1 & Qn-2 & .. & Q1 & Q0)]
     # !UD & [Qn $ (!Qn-1 & !Qn-2 & ... & !Q1 & !Q0)];
```

or les équations ci-dessus sont plus simples.
En effet le logiciel a trouvé des regroupements et il a donc optimisé les équations.
On peut mesurer sur ce simple exemple l'aide importante que procure l'utilisation d'un langage de description comportementale.
En cas d'utilisation de bascule T, on obtient :

```
Tn =   UD & (Qn-1 & Qn-2 & .. & Q1 & Q0)
     # !UD & (!Qn-1 & !Qn-2 & ... & !Q1 & !Q0);
```

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity UPDOWN4 is
port (H,R,UD :in std_logic;
      Q :out std_logic_vector(3 downto 0));
end UPDOWN4;

architecture ARCH_UPDOWN4 of UPDOWN4 is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then
     if UD='0' then X <= X - 1;
     else X <= X + 1;
     end if;
   end if;
  end process;
  Q <= X;
end ARCH_UPDOWN4;
```

**Compteur synchrone modulo N**

L'exemple propose un compteur modulo 10, c'est à dire un compteur qui possède 10 états de sortie différents. Ici, il compte de 0 à 9 puis repasse à 0 et continue ainsi.

```
MODULE modulo10

"Inputs
H,R pin;

"Outputs
Q3..Q0 pin istype 'reg'; Q = [Q3..Q0];

equations
Q.clk = H;
Q.aclr = R;
WHEN (Q >= 9) THEN Q := 0; ELSE Q := Q + 1;

test_vectors
([H,R] -> Q)
@repeat 4  {[.C.,0] -> .X.;}
        [ 0 ,1] -> .X.;
@repeat 15 {[.C.,0] -> .X.;}

END
```

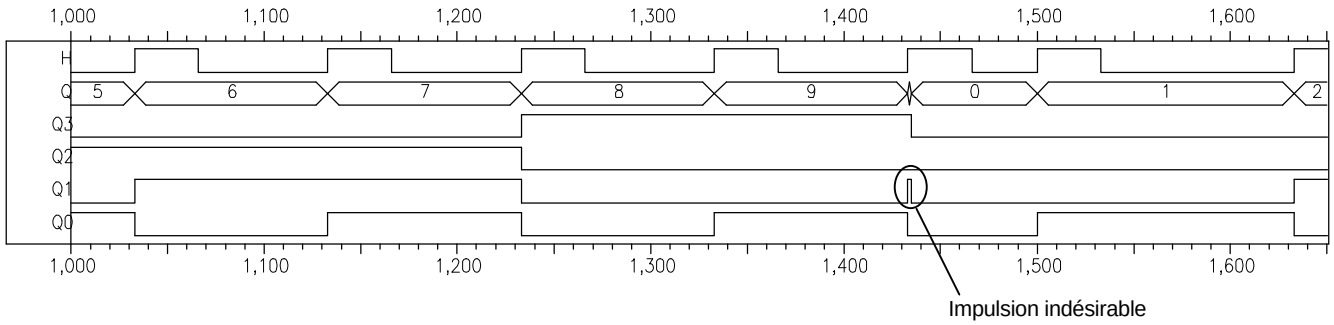
Ci-dessous, on donne les équations générées.

```
Q3 := (Q3 & !Q2 & !Q1 & !Q0 # !Q3 & Q2 & Q1 & Q0);
Q2 := (!Q3 & !Q2 & Q1 & Q0 # !Q3 & Q2 & !Q1 # !Q3 & Q2 & !Q0);
Q1 := (!Q3 & Q1 & !Q0 # !Q3 & !Q1 & Q0);
Q0 := (!Q2 & !Q1 & !Q0 # !Q3 & !Q0);
```

***Remarque** : le même compteur binaire synchrone mais avec une remise à zéro modulo **N** qui serait asynchrone, s'écrirait en langage ABEL :*

```
Q.clk = H;
Q.aclr = (Q == 10);
Q := Q + 1;
```

*Même si celà peut paraître plus simple, c'est à éviter absolument. En effet au moment de la remise à zéro se produira une impulsion (glitch) sur certaines sorties comme le montre les chronogrammes ci-dessous.*




*L'utilisation par une fonction séquentielle de la sortie Q1, risquera de prendre en compte cette impulsion et pourra l'interpréter comme un front actif, ce qui serait erroné.*
*Ce type d'erreur ne se produit pas si la conception est totalement synchrone (y compris les remises à zéro modulo N et les prépositionnements) comme dans la description de la fonction MODULO10 sur cette page.*

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity MODULO10 is
port (H,R :in std_logic;
      Q :out std_logic_vector(3 downto 0));
end MODULO10;

architecture ARCH_MODULO10 of MODULO10 is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then
    if X >= 9 then X <= "0000";
    else X <= X + 1;
    end if;
   end if;
  end process;
  Q <= X;
end ARCH_MODULO10;
```

Les équations générées sont :

```
q_3.D = q_3.Q * /q_2.Q * /q_1.Q * /q_0.Q + /q_3.Q * q_2.Q * q_1.Q * q_0.Q
q_2.D = /q_3.Q * /q_2.Q * q_1.Q * q_0.Q + /q_3.Q * q_2.Q * /q_0.Q
        + /q_3.Q * q_2.Q * /q_1.Q
q_1.D = /q_3.Q * q_1.Q * /q_0.Q + /q_3.Q * /q_1.Q * q_0.Q
q_0.D = /q_2.Q * /q_1.Q * /q_0.Q + /q_3.Q * /q_0.Q
```

Elles sont identiques à celles de la page précédente.
On peut remarquer que ces équations qui sont issues d'une description en VHDL, sont fournies par le logiciel avec une syntaxe voisine du langage ABEL.
`q_3.D` est l'entrée `D` de la bascule nommée `q3` et `q_3.Q` est la sortie `Q` de cette même bascule.

**Compteur synchrone à prépositionnement synchrone**

```
MODULE compt_p

"Inputs
H,R,LOAD,D3..D0 pin;"LOAD est la commande de prepositionnement a D
D = [D3..D0];

"Outputs
Q3..Q0 pin istype 'reg'; Q = [Q3..Q0];

equations
Q.clk = H;
Q.aclr = R;
WHEN (LOAD == 1) THEN Q := D; ELSE Q := Q + 1;

test_vectors
([H,R,LOAD,D] -> Q)
@repeat 4  {[.C.,0,0,5] -> .X.;}
        [ 0 ,1,0,5] -> .X.;
@repeat 9  {[.C.,0,0,5] -> .X.;}
        [.C.,0,1,5] -> .X.;
@repeat 3  {[.C.,0,0,9] -> .X.;}
        [ 0 ,0,1,9] -> .X.;
@repeat 10 {[.C.,0,0,9] -> .X.;}
END
```

Ci-dessous, on donne les équations générées.

```
Q3 := (!LOAD & !Q3 & Q2 & Q1 & Q0 # D3 & LOAD # !LOAD & Q3 & !Q1
     # !LOAD & Q3 & !Q2 # !LOAD & Q3 & !Q0);
Q2 := (!LOAD & !Q2 & Q1 & Q0 # !LOAD & Q2 & !Q1 # LOAD & D2 # !LOAD & Q2 & !Q0);
Q1 := (!LOAD & Q1 & !Q0 # !LOAD & !Q1 & Q0 # LOAD & D1);
Q0 := (LOAD & D0 # !LOAD & !Q0);
```

Si l'on utilise des OU exclusifs, on peut en déduire l'équation utilisable à l'ordre **n** :

```
Qn :=   !LOAD & [Qn $ (Qn-1 & Qn-2 & .. & Q1 & Q0)] # LOAD & Dn;
```

ou avec des bascules T :

```
Tn =   !LOAD & (Qn-1 & Qn-2 & .. & Q1 & Q0) # LOAD & Dn;
```

Une description en langage VHDL donnant un résultat identique avec des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity COMPT_P is
port (H,R,LOAD :in std_logic;
      D :in std_logic_vector(3 downto 0);
      Q :out std_logic_vector(3 downto 0));
end COMPT_P;

architecture ARCH_COMPT_P of COMPT_P is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then
     if LOAD = '1' then X <= D;
     else X <= X + 1;
     end if;
   end if;
  end process;
  Q <= X;
end ARCH_COMPT_P;
```

**Compteur synchrone cascadable**

Pour réaliser un compteur synchrone à partir de plusieurs compteurs synchrones, il faut que chaque compteur élémentaire soit pourvu d'une entrée de validation d'horloge et d'une sortie retenue. C'est ce qui est fait dans l'exemple ci-dessous

```
MODULE comptcas

"Inputs
H,R,EN pin;"EN est la validation de comptage

"Outputs
CO pin istype 'com';"CO est la sortie retenue
Q3..Q0 pin istype 'reg';
Q = [Q3..Q0];

equations
Q.clk = H;
Q.aclr = R;
WHEN (EN == 1) THEN Q := Q + 1; ELSE Q := Q;
WHEN (Q == 15) THEN CO = 1;

test_vectors
([H,R,EN] -> [Q,CO])
@repeat 4  {[.C.,0,1] -> .X.;}
        [ 0 ,1,1] -> .X.;
@repeat 9  {[.C.,0,1] -> .X.;}
        [.C.,0,0] -> .X.;
@repeat 9  {[.C.,0,1] -> .X.;}

END
```

Les équations générées sont :

```
CO = (Q3 & Q2 & Q1 & Q0);
Q3 := (!Q3 & Q2 & Q1 & Q0 & EN # Q3 & !Q0 # Q3 & !Q1 # Q3 & !Q2 # Q3 & !EN);
Q2 := (!Q2 & Q1 & Q0 & EN # Q2 & !Q0 # Q2 & !Q1 # Q2 & !EN);
Q1 := (!Q1 & Q0 & EN # Q1 & !Q0 # Q1 & !EN);
Q0 := (Q0 & !EN # !Q0 & EN);
```

Si l'on utilise l'opérateur OU exclusif, on obtient :

```
Q3 := Q3 $ (Q2 & Q1 & Q0 & EN);
Q2 := Q2 $ (Q1 & Q0 & EN);
Q1 := Q1 $ (Q0 & EN);
Q0 := Q0 $ EN;
```

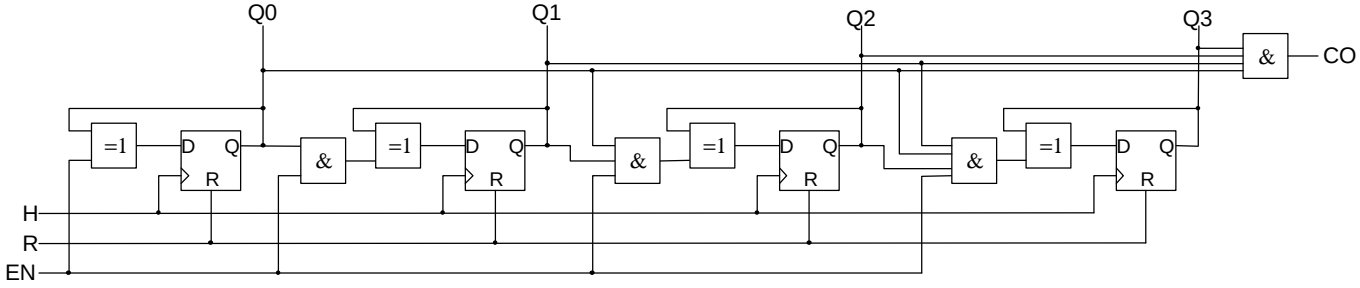
On voit apparaître une relation utilisable à l'ordre **n** :

```
Qn := Qn $ (Qn-1 & Qn-2 & ... & Q1 & Q0 & EN);
```

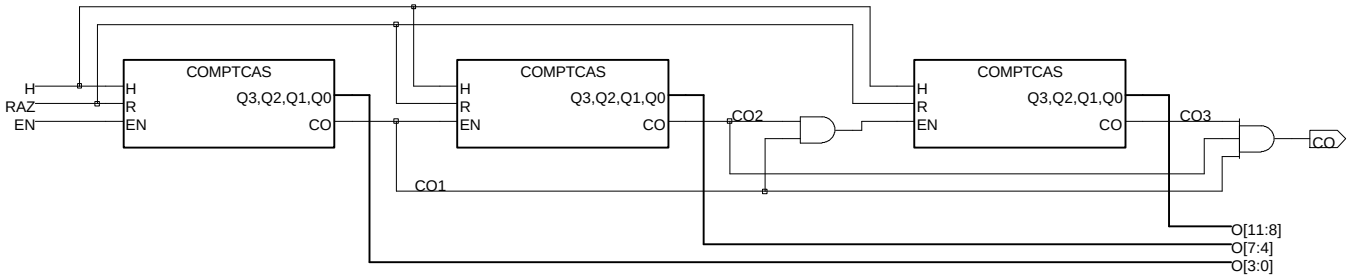
ou bien si l'on utilise des bascules T :

```
Tn = Qn-1 & Qn-2 & ... & Q1 & Q0 & EN;
```

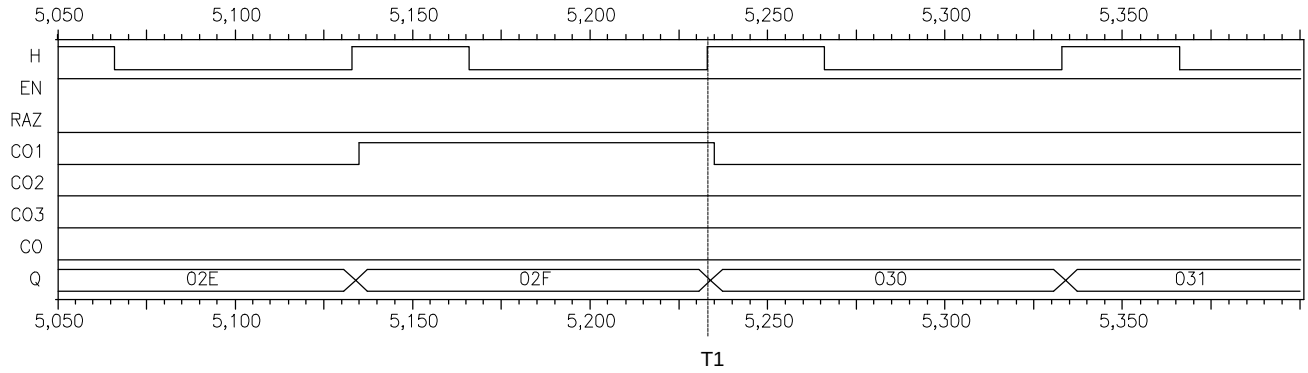
Le schéma structurel d'un tel compteur est donné ci-dessous.



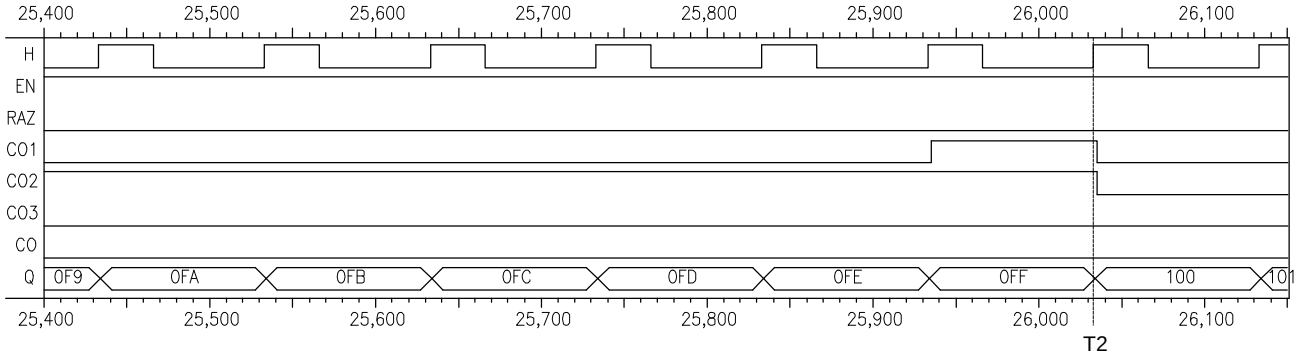
L'exemple ci-dessous montre un compteur 12 bits synchrone réalisé à partir de 3 compteurs 4 bits identiques à celui qui vient d'être décrit.



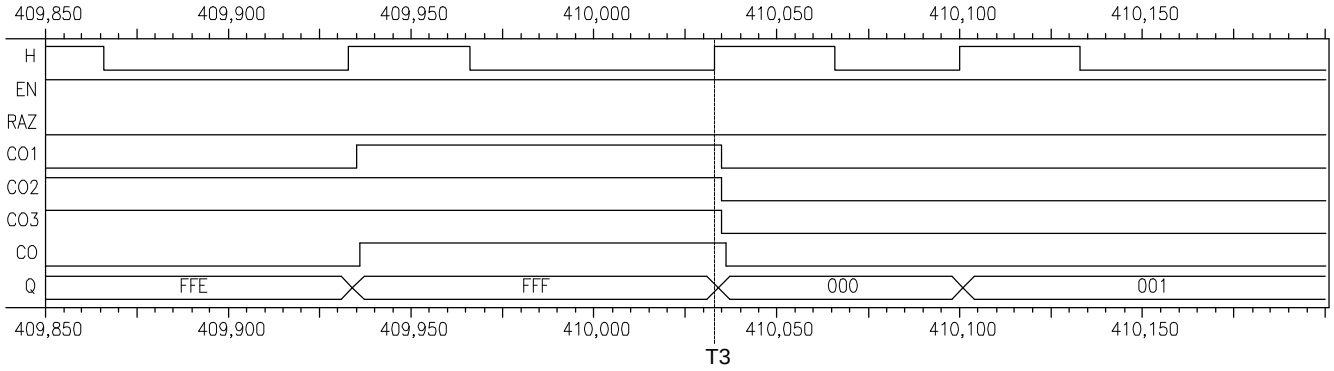

Les chronogrammes suivants montrent comment les compteurs se cascadent de manière synchrone.
Sur le 1er relevé le signal `CO1` ouvre une fenêtre (qui par définition dure exactement une période d'horloge) autorisant un front de l'horloge à être actif sur le 2ème compteur à l'instant T1. Celui-ci ne s'incrémente de manière synchrone qu'à partir de ce front d'horloge. Ses sorties changent d'état consécutivement au même front d'horloge que les sorties du 1er compteur. L'ensemble est donc bien synchrone.




Sur le 2ème relevé le signal `CO1.CO2` ouvre une fenêtre n'autorisant qu'un front d'horloge sur le 3ème compteur à l'instant T2. Ses sorties changent d'état consécutivement au même front d'horloge que les sorties des 1er et 2ème compteurs. L'ensemble est donc bien synchrone.




Le 3ème relevé montre que le signal `CO` ouvre une fenêtre qui pourrait être exploitée par un 4ème compteur. Le front d'horloge actif aurait lieu à l'instant T3.



Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity COMPTCAS is
port (H,R,EN :in std_logic;
      CO :out std_logic;
      Q :out std_logic_vector(3 downto 0));
end COMPTCAS;

architecture ARCH_COMPTCAS of COMPTCAS is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then
     if EN = '1' then X <= X + 1;
     else X <= X;
     end if;
   end if;
  end process;
  Q <= X;
  CO <= '1' when Q = 15 else '0';
end ARCH_COMPTCAS;
```

La description en langage VHDL du compteur 12 bits utilisant 3 compteurs `COMPTCAS` est donnée ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity COMPT12 is
port (H,RAZ,EN :in std_logic;
      CO :out std_logic;
      Q :out std_logic_vector(11 downto 0));
end COMPT12;

architecture ARCH_COMPT12 of COMPT12 is
signal X :std_logic_vector(11 downto 0);
signal CO1,CO2,CO3,EN1 :std_logic;
component COMPTCAS
  port (H,R,EN : in std_logic;
        CO : out std_logic;
        Q : out std_logic_vector(3 downto 0));
end component;
begin
  COMPTEUR1 : COMPTCAS port map(H,RAZ,EN,CO1,Q(3 downto 0));
  COMPTEUR2 : COMPTCAS port map(H,RAZ,CO1,CO2,Q(7 downto 4));
  EN1 <= CO1 and CO2;
  COMPTEUR3 : COMPTCAS port map(H,RAZ,EN1,CO3,Q(11 downto 8));
  CO <= CO1 and CO2 and CO3;
end ARCH_COMPT12;
```

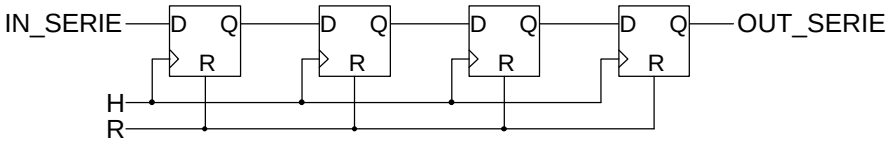
***Remarque** : la mise en cascade de 3 compteurs 4 bits, présentée ici, n'a qu'un intérêt didactique. En effet, il est plus efficace de décrire directement un compteur 12 bits, le logiciel fournissant d'ailleur les mêmes équations dans les 2 cas. Néanmoins, cet exemple expose la méthode à utiliser pour assembler entre elles des fonctions logiques de **manière synchrone.***

***Registres à décalage***

**Registre à décalage à droite**

```
MODULE decal_d

"Inputs
H,RAZ,IN_SERIE pin;

"Outputs
OUT_SERIE pin istype 'reg';
Q2..Q0 node istype 'reg';

equations
[OUT_SERIE,Q2..Q0].clk = H;
[OUT_SERIE,Q2..Q0].aclr = RAZ;
[OUT_SERIE,Q2..Q0] := [Q2..Q0,IN_SERIE];

test_vectors
([H,RAZ,IN_SERIE] -> [OUT_SERIE,Q2,Q1,Q0])
@repeat 2 {[.C.,0,1] -> .X.;}
@repeat 4 {[.C.,0,0] -> .X.;}
@repeat 4 {[.C.,0,1] -> .X.;}
           [ 0 ,1,1] -> .X.;
           [.C.,0,1] -> .X.;
@repeat 4 {[.C.,0,0] -> .X.;}

END
```

Le schéma structurel issu des équations générées par le logiciel est donné ci-dessous.




Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

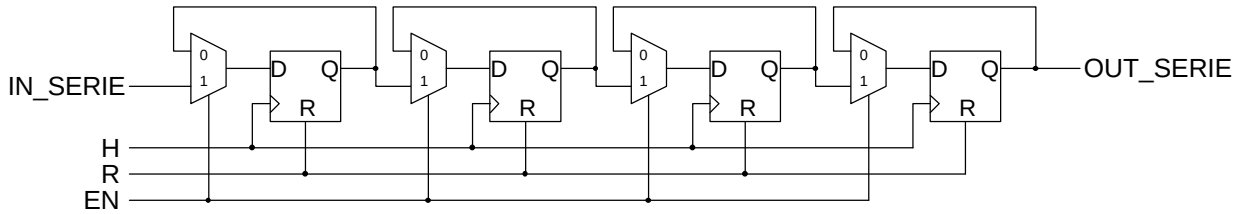
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DECAL_D is
port (H,R,IN_SERIE :in std_logic;
      OUT_SERIE :out std_logic);
end DECAL_D;

architecture ARCH_DECAL_D of DECAL_D is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then Q <= "0000";
   elsif (H'event and H='1') then Q <= Q(2 downto 0) & IN_SERIE;
   end if;
  end process;
  OUT_SERIE <= Q(3);
end ARCH_DECAL_D;
```

On peut remarquer, aussi bien en ABEL qu'en VHDL, la syntaxe extrêmement compacte et lisible permettant de décrire cette fonction.

**Registre à décalage à droite à autorisation d'horloge**

```
MODULE decal_de

"Inputs
H,RAZ,EN,IN_SERIE pin;

"Outputs
OUT_SERIE pin istype 'reg';
Q2..Q0 node istype 'reg';

equations
[OUT_SERIE,Q2..Q0].clk = H;
[OUT_SERIE,Q2..Q0].aclr = RAZ;
when EN == 1 then [OUT_SERIE,Q2..Q0] := [Q2..Q0,IN_SERIE];
else [OUT_SERIE,Q2..Q0] := [OUT_SERIE,Q2..Q0];

test_vectors
([H,RAZ,EN,IN_SERIE] -> [OUT_SERIE,Q2,Q1,Q0])
@repeat 2 {[.C.,0,1,1] -> .X.;}
@repeat 4 {[.C.,0,1,0] -> .X.;}
@repeat 4 {[.C.,0,1,1] -> .X.;}
           [ 0 ,1,1,1] -> .X.;
           [.C.,0,1,1] -> .X.;
@repeat 2 {[.C.,0,1,0] -> .X.;}
           [.C.,0,0,0] -> .X.;
@repeat 2 {[.C.,0,1,0] -> .X.;}

END
```

Le schéma structurel issu des équations générées par le logiciel est donné ci-dessous.




On remarque que le signal d'autorisation d'horloge `EN` agit sur des multiplexeurs (le symbole est un trapèze isocèle) qui place sur l'entrée `D`, soit la sortie de la bascule de gauche (décalage), soit la sortie de la même bascule (inactivité apparente de l'horloge qui remémorise l'état antérieur).

***Remarque** : On aurait pu être tenté d'écrire* `[OUT_SERIE,Q2..Q0].clk = H & EN;` *mais cette façon de faire conduisait à un fonctionnement asynchrone en introduisant un opérateur ET (donc un retard) dans le circuit d'horloge et la simplification apportée n'était qu'apparente. En effet le nombre de cellule utilisé risque d'être plus important (une cellule supplémentaire pour générer* `H & EN`*).*

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DECAL_DE is
port (H,R,EN,IN_SERIE :in std_logic;
      OUT_SERIE :out std_logic);
end DECAL_DE;

architecture ARCH_DECAL_DE of DECAL_DE is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then Q <= "0000";
   elsif (H'event and H='1') then
     if EN = '1' then Q <= Q(2 downto 0) & IN_SERIE;
     else Q <= Q;
     end if;
   end if;
  end process;
  OUT_SERIE <= Q(3);
end ARCH_DECAL_DE;
```

Encore une fois les équations fournies sont exactement les mêmes que celles issues de la description en ABEL.

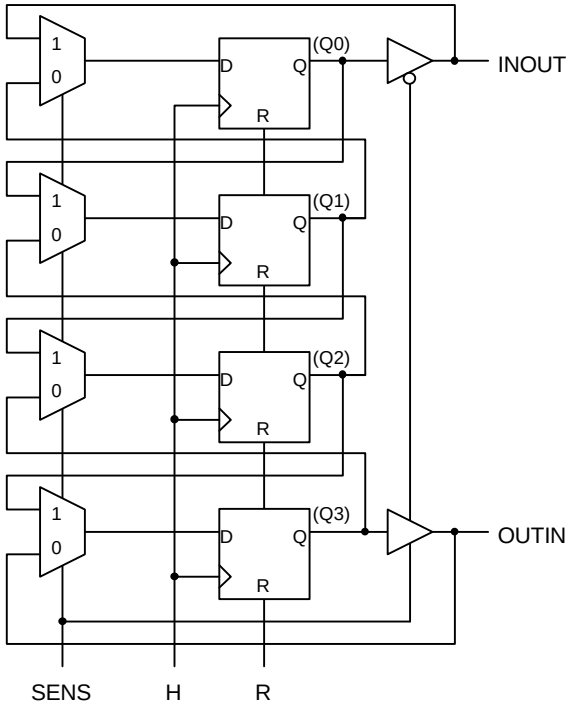
**Registre à décalage à droite ou à gauche**

```
MODULE decal_dg

"Inputs
H,RAZ,SENS pin;"SENS commande le sens de decalage (1 : droite, 0 : gauche)

"Outputs
Q3..Q0 node istype 'reg'; Q = [Q3..Q0];

"Bidirectionnels
INOUT,OUTIN pin;

"equivalences
c,x = .C.,.X.;

equations
Q.clk = H;
Q.aclr = RAZ;
INOUT.oe = !SENS;
OUTIN.oe = SENS;
WHEN (SENS == 1) THEN [Q3..Q0] := [Q2..Q0,INOUT];
ELSE [Q0..Q3] := [Q1..Q3,OUTIN];
OUTIN = Q3; INOUT = Q0;

test_vectors
([H,RAZ,SENS,INOUT,OUTIN] -> [INOUT,Q0,Q1,Q2,Q3,OUTIN])
@repeat 2 {[c,0,1,1,x] -> x;}
@repeat 4 {[c,0,1,0,x] -> x;}
@repeat 4 {[c,0,1,1,x] -> x;}
           [0,1,1,1,x] -> x;
           [c,0,1,1,x] -> x;
@repeat 2 {[c,0,1,0,x] -> x;}
           [0,0,0,x,0] -> x;
@repeat 2 {[c,0,0,x,1] -> x;}
@repeat 4 {[c,0,0,x,0] -> x;}
END
```

Les équations générées sont :

```
INOUT = Q0;
INOUT.OE = (!SENS);
OUTIN = Q3;
OUTIN.OE = (SENS);
Q3 := (SENS & Q2 # !SENS & OUTIN);
Q2 := (SENS & Q1 # !SENS & Q3);
Q1 := (!SENS & Q2 # SENS & Q0);
Q0 := (!SENS & Q1 # SENS & INOUT);
```

Ce qui correspond au schéma structurel ci-contre dans lequel une fonction de type `(!SENS & Q2 # SENS & Q0)` est représentée comme un multiplexeur (le symbole est un trapèze isocèle).



Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DECAL_DG is
port (H,R,SENS :in std_logic;
      IN_OUT,OUT_IN :inout std_logic);
end DECAL_DG;

architecture ARCH_DECAL_DG of DECAL_DG is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then Q <= "0000";
   elsif (H'event and H='1') then
     if SENS = '1' then Q <= Q(2 downto 0) & IN_OUT;
     else Q <= OUT_IN & Q(3 downto 1);
     end if;
   end if;
  end process;
  OUT_IN <= Q(3) when SENS = '1' else 'Z';
  IN_OUT <= Q(0) when SENS = '0' else 'Z';
end ARCH_DECAL_DG;
```

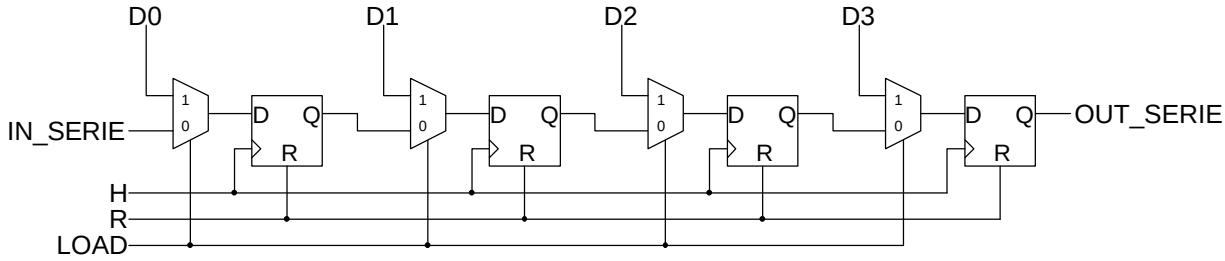
**Registre à décalage à chargement parallèle synchrone**

```
MODULE decal_p

"Inputs
H,RAZ,IN_SERIE,LOAD,D3..D0 pin;
D = [D3..D0];

"Outputs
OUT_SERIE pin istype 'reg';
Q2..Q0 node istype 'reg';

equations
[OUT_SERIE,Q2..Q0].clk = H;
[OUT_SERIE,Q2..Q0].aclr = RAZ;
WHEN (LOAD == 1) THEN [OUT_SERIE,Q2..Q0] := [D3..D0];
ELSE [OUT_SERIE,Q2..Q0] := [Q2..Q0,IN_SERIE];

test_vectors
([H,RAZ,IN_SERIE,LOAD,D] -> [OUT_SERIE,Q2,Q1,Q0])
@repeat 2 {[.C.,0,1,0,0] -> .X.;}
@repeat 4 {[.C.,0,0,0,0] -> .X.;}
@repeat 4 {[.C.,0,1,0,0] -> .X.;}
           [ 0 ,1,1,0,0] -> .X.;
           [.C.,0,1,0,0] -> .X.;
@repeat 4 {[.C.,0,0,0,0] -> .X.;}
           [.C.,0,0,1,6] -> .X.;
           [.C.,0,1,0,6] -> .X.;
@repeat 4 {[.C.,0,0,0,0] -> .X.;}

END
```

Les équations générées sont :

```
OUT_SERIE := (D3 & LOAD # !LOAD & Q2);
Q2 := (LOAD & D2 # !LOAD & Q1);
Q1 := (LOAD & D1 # !LOAD & Q0);
Q0 := (LOAD & D0 # !LOAD & IN_SERIE);
```

Le schéma structurel issu de ces équations est donné ci-dessous.



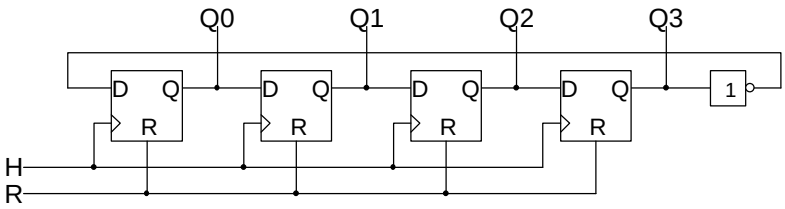
Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DECAL_P is
port (H,R,IN_SERIE,LOAD :in std_logic;
      D :in std_logic_vector(3 downto 0);
      OUT_SERIE :out std_logic);
end DECAL_P;

architecture ARCH_DECAL_P of DECAL_P is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then Q <= "0000";
   elsif (H'event and H='1') then
     if LOAD = '1' then Q <= D;
     else Q <= Q(2 downto 0) & IN_SERIE;
     end if;
   end if;
  end process;
  OUT_SERIE <= Q(3);
end ARCH_DECAL_P;
```

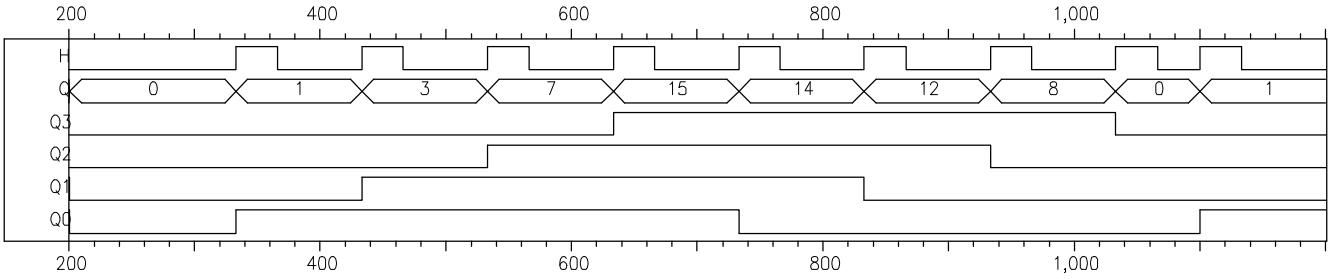
**Compteur en anneau : génération de séquence**

```
MODULE dec_seq

"Inputs
H,RAZ pin;

"Outputs
Q3..Q0 pin istype 'reg';
Q = [Q3..Q0];

Equations
Q.clk = H;
Q.aclr = RAZ;
[Q3..Q0] := [Q2..Q0,!Q3];

Test_vectors
([H,RAZ] -> Q)
@repeat  2 {[.C.,0] -> .X.;}
           [ 0 ,1] -> .X.;
@repeat 16 {[.C.,0] -> .X.;}

END
```

Il s'agit ici d'un cas particulier du registre à décalage à un seul sens. On obtient le schéma suivant :




Sur le chronogramme ci-dessous, on peut voir que si l'on raccorde les sorties aux entrées d'un sommateur analogique, on obtient une tension en escalier triangulaire voire une pseudo-sinusoïde en pondérant correctement le sommateur.



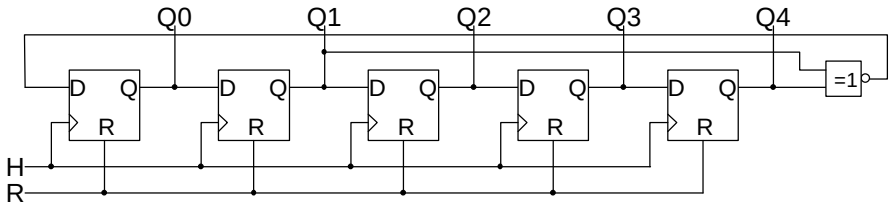
Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DEC_SEQ is
port (H,R :in std_logic;
      Q :out std_logic_vector(3 downto 0));
end DEC_SEQ;

architecture ARCH_DEC_SEQ of DEC_SEQ is
signal X :std_logic_vector(3 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "0000";
   elsif (H'event and H='1') then
    X <= X(2 downto 0) & not X(3);
   end if;
  end process;
  Q <= X;
end ARCH_DEC_SEQ;
```

**Compteur en anneau : génération de séquence pseudo-aléatoire**

```
MODULE dec_alea

"Inputs
H,RAZ pin;

"Outputs
Q4..Q0 pin istype 'reg'; Q = [Q4..Q0];

Equations
Q.clk = H;
Q.aclr = RAZ;
[Q4..Q0] := [Q3..Q0,(Q1 !$ Q4)];

Test_vectors
([H,RAZ] -> Q)
@repeat  2 {[.C.,0] -> .X.;}
           [ 0 ,1] -> .X.;
@repeat 34 {[.C.,0] -> .X.;}
END
```

Ici aussi il s'agit d'un cas particulier du registre à décalage à un seul sens. On obtient le schéma suivant :




Ici, on a décrit un générateur de séquence binaire pseudo-aléatoire à 5 étages. On démontre que la longueur maximale de la séquence d'un tel générateur (avant de retrouver la même configuration) est de $2^n - 1$ (**n** étant le nombre d'étage). Il existe de nombreuses autres possibilités dans les rebouclages qui conduisent à des résultats semblables.
Si l'on se contente d'un reblouclage par un seul OU exclusif à 2 entrées, on peut citer les configurations suivantes :

| Nombre d'étage **n** | Longueur de la séquence $2^n - 1$ | Numéros des sorties de registre connectées au OU exclusif |
|---|---|---|
| 4 | 15 | 1, 4 |
| 5 | 31 | 2, 5 |
| 6 | 63 | 1, 6 |
| 7 | 127 | 3, 7 |
| 9 | 511 | 4, 9 |
| 10 | 1023 | 3, 10 |
| 11 | 2047 | 2, 11 |
| 15 | 32767 | 1, 15 |
| 17 | 131071 | 3, 17 |

On choisira pour l'opérateur de rebouclage un OU exclusif (XOR) ou un NI exclusif (XNOR) suivant les possibilités d'initialisation des registres. En effet, si dans le générateur décrit ci-dessus (initialisation par une remise à zéro asynchrone) on utilise un rebouclage avec `Q1 $ Q4` au lieu de `Q1 !$ Q4`, le générateur restera continuellement à 0.
Si l'initialisation avait été une mise à 1 asynchrone, il aurait fallu utiliser le OU exclusif.

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DEC_ALEA is
port (H,R :in std_logic;
      Q :out std_logic_vector(4 downto 0));
end DEC_ALEA;

architecture ARCH_DEC_ALEA of DEC_ALEA is
signal X :std_logic_vector(4 downto 0);
signal Y :std_logic;
begin
  process(H,R)
   begin
   if R='1' then X <= "00000";
   elsif (H'event and H='1') then
    X <= X(3 downto 0) & Y;
   end if;
  end process;
  Y <= not (X(4) xor X(1));
  Q <= X;
end ARCH_DEC_ALEA;
```

# FONCTIONS DIVERSES

## *FONCTIONS DE MEMORISATION*

Les éléments de base de la mémorisation sont les bascules D de type "**verrou**" (*Latch* : activée par des niveaux) ou de type "**maître-esclave**" (*Flip-Flop* : activée par des fronts).

### Registre 8 bits

Il s'agit en fait d'un ensemble de bascules D formant un mot binaire de 8 bits. Ici on a choisi des bascules D maître-esclave car elles permettent un fonctionnement synchrone de l'ensemble dans lequel le registre sera inséré. La synthèse dans un circuit logique programmable sera aussi plus simple puisqu'il existe une bascule D maître-esclave par cellule de base.

```
MODULE registre

"Inputs
H,RAZ,OE,D7..D0 pin; D = [D7..D0];

"Outputs
Q7..Q0 pin istype 'reg'; Q = [Q7..Q0];

Equations
Q.clk = H;
Q.aclr = RAZ;
Q.oe = OE;
Q := D;

Test_vectors
([H,RAZ,OE,D] -> Q)
[.C.,0,0,10] -> .X.;
[ 0 ,0,1, 0] -> .X.;
[ 0 ,1,1, 0] -> .X.;
END
```

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity REGISTRE is
port (H,R,OE :in std_logic;
      D :in std_logic_vector(7 downto 0);
      Q :out std_logic_vector(7 downto 0));
end REGISTRE;

architecture ARCH_REGISTRE of REGISTRE is
signal X :std_logic_vector(7 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "00000000";
   elsif (H'event and H='1') then X <= D;
   end if;
  end process;
  Q <= X when OE = '1' else "ZZZZZZZZ";
end ARCH_REGISTRE;
```

Comme on peut le remarquer, il s'agit d'un registre utilisable sur un bus puisqu'il est équipé d'une sorties 3 états. En ABEL la sortie 3 états est précisée par `.oe` et en VHDL par l'état `'Z'`(non défini pour des signaux de type `bit` ou `bit_vector`).

**Registre 8 bits à accès bidirectionnel**

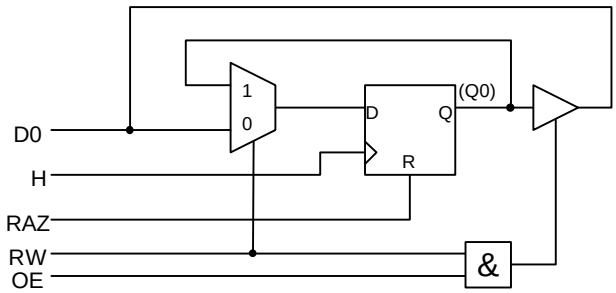
Il s'agit d'un ensemble de bascules D identique au précédent exemple, mais pour lequel les bornes d'entrées et sorties sont communes. Le sens de transfert est commandé par un signal de lecture-écriture.

```
MODULE regis_bi

"Inputs
H,RAZ,OE,RW pin;

"Outputs
Q7..Q0 node istype 'reg';
Q = [Q7..Q0];

"Inputs Outputs
D7..D0 pin istype 'com';
D = [D7..D0];

Equations
Q.clk = H;
Q.aclr = RAZ;
WHEN RW == 0 THEN Q := D; ELSE Q := Q;
D.oe = OE & RW;
D = Q;

Test_vectors
([H,RAZ,OE,RW,  D] -> [Q,D])
[.C.,0 ,0 ,0 , 10] -> .X.;
[ 0 ,0 ,1 ,0 ,.X.] -> .X.;
[ 0 ,0 ,1 ,1 ,.X.] -> .X.;
[ 0 ,0 ,1 ,0 ,  5] -> .X.;
[.C.,0 ,0 ,1 ,  5] -> .X.;
[ 0 ,0 ,1 ,1 ,.X.] -> .X.;
[ 0 ,0 ,0 ,1 ,.X.] -> .X.;
[ 0 ,1 ,1 ,1 ,.X.] -> .X.;

END
```

Les équations générées sont identiques pour chacun des 8 bits. On donne ci-dessous, celles relatives au bit 0.

```
D0 = (Q0);
D0.OE = OE & RW;
Q0 := (!RW & D0 # RW & Q0);
```

Ces équations peuvent se représenter par le schéma structurel suivant :



Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity REGIS_BI is
port (H,R,OE,RW :in std_logic;
      D :inout std_logic_vector(7 downto 0));
end REGIS_BI;

architecture ARCH_REGIS_BI of REGIS_BI is
signal X :std_logic_vector(7 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "00000000";
   elsif (H'event and H='1') then
     if RW = '0' then X <= D;
     end if;
   end if;
  end process;
  D <= X when (OE = '1' and RW = '1') else "ZZZZZZZZ";
end ARCH_REGIS_BI;
```

***FONCTION DE DIVISION DE FREQUENCE***

Lorsque le rapport de division de fréquence est fixe, on réalise cette fonction grâce à un compteur ou décompteur synchrone modulo **N** ou bien encore par un compteur en anneau.
Dans le cas où le rapport de division est fourni par un nombre binaire, on utilise couramment un système de comptage ou décomptage binaire synchrone prépositionnable de manière synchrone et dont l'ordre de prépositionnement est donné par la détection du compteur ou décompteur plein.

**Division de fréquence par comptage**

Le fonction réalisée ici divise la fréquence de l'horloge du compteur par un nombre compris entre 2 et 16. Le rapport de division est $2^4$ - N avec N < 15 (N est le nombre binaire chargé à chaque prépositionnement).

```
MODULE divfreq1

"Inputs
H,N3..N0 pin; N = [N3..N0];

"Outputs
Q3..Q0 node istype 'reg';
Q = [Q3..Q0];
DIV pin istype 'com';"sortie divisee par 16 - N (N < 15)

"Equivalences
c,x = .C.,.X.;

Equations
Q.clk = H;
WHEN (Q == 15) THEN { Q := N; DIV = 1;} ELSE {Q := Q + 1; DIV = 0;}

Test_vectors
([H,N] -> [Q,DIV])
@repeat 35 {[c, 0] -> x;}
@repeat 25 {[c, 5] -> x;}
@repeat 20 {[c,14] -> x;}
@repeat 10 {[c,15] -> x;}

END
```

Les équations générées sont :

```
DIV = (Q0 & Q1 & Q2 & Q3);
Q3 := (N3 & Q3 # Q0 & Q1 & Q2 & !Q3 # !Q2 & Q3 # !Q1 & Q3 # !Q0 & Q3);
Q2 := (Q2 & Q3 & N2 # Q0 & Q1 & !Q2 # !Q1 & Q2 # !Q0 & Q2);
Q1 := (Q1 & Q2 & Q3 & N1 # Q0 & !Q1 # !Q0 & Q1);
Q0 := (Q1 & Q2 & Q3 & N0 # !Q0);
```

Si on utilise l'opérateur OU exclusif, on peut écrire :

```
Q3 := (N3 & Q3) # [Q3 $ (Q2 & Q1 & Q0)];
Q2 := (N2 & Q3 & Q2) # [Q2 $ (Q1 & Q0)];
Q1 := (N1 & Q3 & Q2 & Q1) # (Q1 $ Q0);
Q0 := (N0 & Q3 & Q2 & Q1) # !Q0;
```

On voit apparaître dans ces équations, celles du compteur binaire synchrone auxquelles on a ajouté la condition de chargement lorsque le compteur est à sa valeur maximale.
On voit aussi que le logiciel a optimisé les équations car sinon celles-ci auraient dû être les suivantes :

```
Q3 := (N3 & DIV) # [Q3 $ (Q2 & Q1 & Q0)];
Q2 := (N2 & DIV) # [Q2 $ (Q1 & Q0)];
Q1 := (N1 & DIV) # (Q1 $ Q0);
Q0 := (N0 & DIV) # !Q0;
```

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DIV_FREQ1 is
port (H :in std_logic;
      N :in std_logic_vector(3 downto 0);
      DIV : out std_logic);
end DIV_FREQ1;

architecture ARCH_DIV_FREQ1 of DIV_FREQ1 is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H)
   begin
   if (H'event and H='1') then
     if Q = 15 then Q <= N;
     else Q <= Q + 1;
     end if;
   end if;
  end process;
  DIV <= '1' when Q = 15 else '0';
end ARCH_DIV_FREQ1;
```

**Division de fréquence par décomptage**

Le fonction réalisée ici divise la fréquence de l'horloge du décompteur par un nombre compris entre 2 et 16.

```
MODULE divfreq2

"Inputs
H,N3..N0 pin; N = [N3..N0];"

"Outputs
Q3..Q0 node istype 'reg'; Q = [Q3..Q0];
DIV pin istype 'com';"sortie divisee par N + 1 (N > 0)

Equations
Q.clk = H;
WHEN (Q == 0) THEN { Q := N; DIV = 1;} ELSE {Q := Q - 1; DIV = 0;}

Test_vectors
([H,N] -> [Q,DIV])
@repeat  5 {[.C., 0] -> .X.;}
@repeat 10 {[.C., 1] -> .X.;}
@repeat 20 {[.C., 5] -> .X.;}
@repeat 40 {[.C.,15] -> .X.;}

END
```

Les équations générées sont les suivantes :

```
DIV = (!Q0 & !Q1 & !Q2 & !Q3);
Q3 := (N3 & !Q0 & !Q1 & !Q2 & !Q3 # Q2 & Q3 # Q1 & Q3 # Q0 & Q3);
Q2 := (!Q0 & !Q1 & !Q2 & Q3 # !Q0 & !Q1 & !Q2 & N2 # Q1 & Q2 # Q0 & Q2);
Q1 := (!Q0 & !Q1 & Q3 # !Q0 & !Q1 & Q2 # Q0 & Q1 # !Q0 & !Q1 & N1);
Q0 := (!Q0 & Q3 # !Q0 & Q2 # !Q0 & Q1 # !Q0 & N0);
```

On peut s'apercevoir qu'il est difficile d'utiliser les opérateurs OU exclusif dans ces équations. On ne voit donc pas l'allure des équations d'un décompteur.

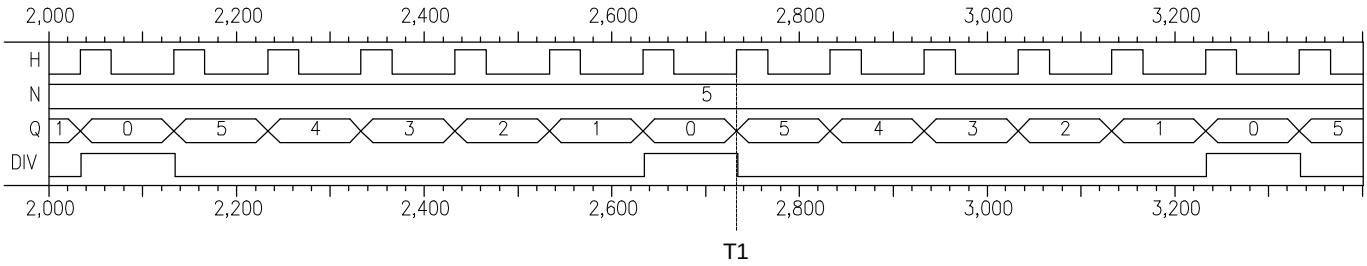
Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity DIV_FREQ2 is
port (H :in std_logic;
      N :in std_logic_vector(3 downto 0);
      DIV : out std_logic);
end DIV_FREQ2;

architecture ARCH_DIV_FREQ2 of DIV_FREQ2 is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H)
   begin
   if (H'event and H='1') then
     if Q = 0 then Q <= N;
     else Q <= Q - 1;
     end if;
   end if;
  end process;
  DIV <= '1' when Q = 0 else '0';
end ARCH_DIV_FREQ2;
```

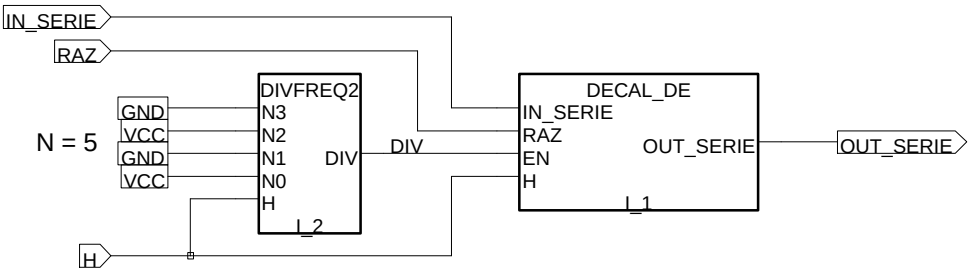
Les deux formes de diviseur de fréquence décrites ici sont équivalentes du point de vue de l'utilisation des ressources d'un composant logique programmable. La seconde présente l'avantage de fournir un rapport de division variant dans le même sens que le mot **N**.

**Utilisation d'un diviseur de fréquence en mode synchrone**

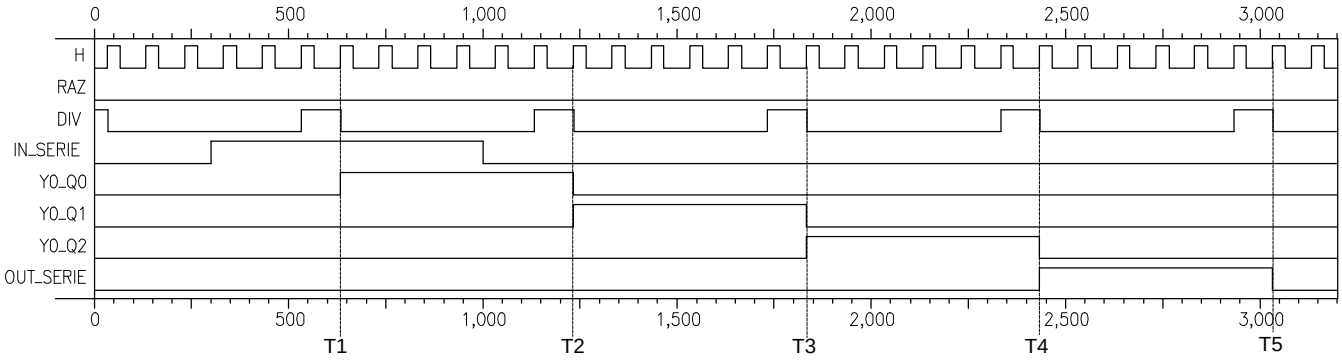
On donne ci-dessous les chronogrammes de simulation du diviseur de fréquence par décomptage `DIVFREQ2`.




Le signal de sortie `DIV` dont la fréquence est divisée par 6 (rapport de division N + 1) fournit un état logique 1 durant exactement une période de l'horloge `H`.
Il ouvre ainsi une fenêtre durant laquelle on est sûr de trouver un et un seul front actif de cette horloge (à l'instant `T1`).
Ce signal `DIV` peut donc être utilisé comme autorisation d'horloge pour une fonction logique réalisant ainsi un ensemble synchrone comme dans l'exemple ci-dessous.




La simulation de ce schéma donne les chronogrammes ci-dessous.




On voit nettement apparaître les fenêtres d'autorisation d'horloge qui sélectionnent les fronts actifs de l'horloge `H` aux instants T1, T2, T3, T4 et T5 pour former un ensemble complètement synchrone.

***FONCTIONS DE TYPE "MONOSTABLE"***

**Fonction Monostable**

La durée caractéristique de la fonction monostable est générée par un comptage binaire jusqu'à une valeur prédéfinie.
L'exemple fourni ci-dessous est celui d'un monostable redéclenchable qui peut être utilisé comme chien de garde.

```
MODULE monosta

"Inputs
H,START pin;"declenchement du monostable

"Outputs
Q3..Q0 node istype 'reg'; Q = [Q3..Q0];
EN pin istype 'com'; "sortie du monostable et autorisation de comptage

N = 12;"Valeur caracteristique de la duree du monostable

Equations
Q.clk = H;
     WHEN (START == 1) THEN Q := 0;
ELSE WHEN (EN == 1) THEN Q := Q + 1;
                    ELSE Q := Q;
EN = (Q != N);

Test_vectors
([H,START] -> [Q,EN])
            [.C.,0] -> .X.;
            [.C.,1] -> .X.;
@repeat 16 {[.C.,0] -> .X.;}
            [.C.,1] -> .X.;
@repeat  8 {[.C.,0] -> .X.;}
            [.C.,1] -> .X.;
@repeat 16 {[.C.,0] -> .X.;}

END
```

Le logiciel fournit les équations suivantes :

```
EN = (Q1 # !Q2 # !Q3 # Q0);
Q3 := (!Q3 & Q2 & Q1 & Q0 & !START & EN # Q3 & !Q0 & !START
     # Q3 & !Q1 & !START # Q3 & !Q2 & !START # Q3 & !START & !EN);
Q2 := (!Q2 & Q1 & Q0 & !START & EN
     # Q2 & !Q0 & !START # Q2 & !Q1 & !START # Q2 & !START & !EN);
Q1 := (!Q1 & Q0 & !START & EN # Q1 & !Q0 & !START # Q1 & !START & !EN);
Q0 := (Q0 & !START & !EN # !Q0 & !START & EN);
```

En utilisant l'opérateur OU exclusifs, on obtient les équations suivantes :

```
Q3 := [!Q3 & (Q2 & Q1 & Q0 & EN) # Q3 & !(Q2 & Q1 & Q0 & EN)] & !START;
Q2 := [!Q2 & (Q1 & Q0 & EN) # Q2 & !(Q1 & Q0 & EN)] & !START;
Q1 := [!Q1 & (Q0 & EN) # Q1 & !(Q0 & EN)] & !START;
Q0 := [!Q0 & EN # Q0 & !EN] & !START;
```

Soit :

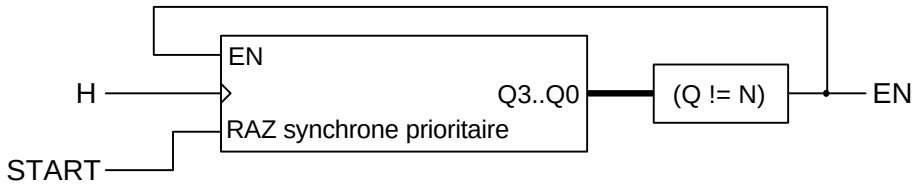
```
Q3 := [Q3 $ (Q2 & Q1 & Q0 & EN)] & !START;
Q2 := [Q2 $ (Q1 & Q0 & EN)] & !START;
Q1 := [Q1 $ (Q0 & EN)] & !START;
Q0 := [Q0 $ EN] & !START;
```

On voit apparaître une relation utilisable à l'ordre **n** :

```
Qn := [Qn $ (Qn-1 & Qn-2 & ... & Q1 & Q0 & EN)] & !START;
```

Ces équations montrent que la remise à zéro synchrone est prioritaire sur l'autorisation de comptage. C'est ce qui permet de rendre redéclenchable cette fonction monostable.

On peut décrire ces équations avec le schéma bloc ci-dessous.




Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity MONOSTA is
port (H,START :in std_logic;
      EN :out std_logic);
end MONOSTA;

architecture ARCH_MONOSTA of MONOSTA is
constant N : std_logic_vector(3 downto 0) := "1100";
signal Q :std_logic_vector(3 downto 0);
signal Y :std_logic;
begin
  process(H)
   begin
   if (H'event and H='1') then
     if START = '1' then Q <= "0000";
     elsif Y = '1' then Q <= Q + 1;
     else Q <= Q;
     end if;
   end if;
  end process;
  Y <= '1' when Q /= N else '0';
  EN <= Y;
end ARCH_MONOSTA;
```

Les équations générées sont :

```
en = q_0.Q + q_1.Q + /q_2.Q + /q_3.Q
q_3.D = /q_3.Q * q_2.Q * q_1.Q * q_0.Q * /start + q_3.Q * /q_0.Q * /start
        + q_3.Q * /q_1.Q * /start + q_3.Q * /q_2.Q * /start
q_2.D = /q_2.Q * q_1.Q * q_0.Q * /start + q_2.Q * /q_0.Q * /start
        + q_2.Q * /q_1.Q * /start
q_1.D = q_1.Q * /q_0.Q * /start + /q_1.Q * q_0.Q * /start
q_0.D = q_1.Q * /q_0.Q * /start + /q_2.Q * /q_0.Q * /start + /q_3.Q * /q_0.Q * /start
```

On peut remarquer que le signal `EN` n'apparaît pas dans les équations de Q0 à Q3 comme dans celles générées à partir de la description en ABEL.
L'optimisateur a remplacé `EN` par son expression dans les équations de Q0 à Q3 et a simplifié celles-ci au maximum.
On obtient ainsi un code plus compact mais qui occupe le même nombre de cellules. Les résultats au niveau temporelle aurait pu aussi être différents, mais comme l'ensemble est synchrone, ces différences ne seront pas retransmises sur les sorties.
Les deux solutions fournies ont donc des performances tout à fait semblables.

**Génération d'un train de N impulsions**

Il s'agit d'une variante du précédent monostable. Celui-ci est non redéclenchable et l'on utilise le bit de poids faible du compteur pour générer les impulsions. Le nombre d'impulsions générées est $\frac{N}{2}+1$.

```
MODULE gen_puls

"Inputs
H,START pin;"declenchement du train d impulsion

"Outputs
PULS pin istype 'com';
Q3..Q0 node istype 'reg'; Q = [Q3..Q0];
EN node istype 'com'; "autorisation de comptage

N = 14;"Il y aura N/2 + 1 impulsions

Equations
Q.clk = H;
     WHEN (EN == 1) THEN Q := Q + 1;
ELSE WHEN (START == 1) THEN Q := 0;
                       ELSE Q := Q;
EN = (Q != N);
PULS = Q0;

Test_vectors
([H,START] -> [Q,EN,PULS])
            [.C.,0] -> .X.;
            [.C.,1] -> .X.;
@repeat 16 {[.C.,0] -> .X.;}
            [.C.,1] -> .X.;
@repeat  8 {[.C.,0] -> .X.;}
            [.C.,1] -> .X.;
@repeat 16 {[.C.,0] -> .X.;}

END
```

Avant optimisation et routage, le logiciel fournit les équations suivantes :

```
PULS = (Q0);
Q3 := (!Q3 & Q2 & Q1 & Q0 & EN # Q3 & !Q0 & EN # Q3 & !Q1 & EN # Q3 & !Q2 & EN
     # Q3 & !EN & !START);
Q2 := (!Q2 & Q1 & Q0 & EN # Q2 & !Q0 & EN # Q2 & !Q1 & EN # Q2 & !EN & !START);
Q1 := (!Q1 & Q0 & EN # Q1 & !Q0 & EN # Q1 & !EN & !START);
Q0 := (Q0 & !EN & !START # !Q0 & EN);
EN = (!Q1 # !Q2 # !Q3 # Q0);
```

En utilisant l'opérateur OU exclusifs, on obtient les équations suivantes :

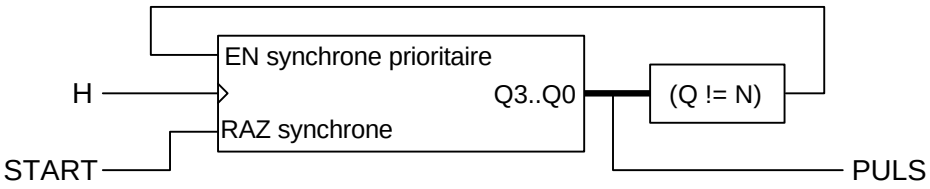
```
Q3 := {[Q3 $ (Q2 & Q1 & Q0)] & EN} # (Q3 & !EN & !START);
Q2 := {[Q2 $ (Q1 & Q0)] & EN} # (Q2 & !EN & !START);
Q1 := {[Q1 $ Q0] & EN} # (Q1 & !EN & !START);
Q0 := (!Q0 & EN) # (Q0 & !EN & !START);
```

On voit apparaître une relation utilisable à l'ordre **n** :

```
Qn := {[Qn $ (Qn-1 & Qn-2 & ... & Q1 & Q0)] & EN} # (Qn & !EN & !START);
```

Ces équations montrent que l'autorisation de comptage est prioritaire sur la remise à zéro synchrone. C'est ce qui permet de rendre non-redéclenchable cette fonction monostable.

On peut décrire ces équations avec le schéma bloc ci-dessous.




Après optimisation, les équations deviennent :

```
Q3.D = Q3 & !Q2 # Q3 & !Q1 # !Q3 & Q2 & Q1 & PULS.PIN # Q3 & !PULS.PIN & !START;
Q2.D = Q2 & !Q1 # !Q2 & Q1 & PULS.PIN # !Q3 & Q2 & !PULS.PIN
       # Q2 & !PULS.PIN & !START;
Q1.D = !Q1 & PULS.PIN # !Q3 & Q1 & !PULS.PIN # !Q2 & Q1 & !PULS.PIN
       # Q1 & !PULS.PIN & !START;
PULS.D = !Q3 & !PULS.PIN # !Q2 & !PULS.PIN # !Q1 & !PULS.PIN;
```

Le signal `EN` a été éliminé car il a été déclaré comme `node` et les équations ont été simplifiées.

Une description en langage VHDL donnant un résultat identique est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity GEN_PULS is
port (H,START :in std_logic;
      PULS :out std_logic);
end GEN_PULS;

architecture ARCH_GEN_PULS of GEN_PULS is
constant N : std_logic_vector(3 downto 0) := "1110";
signal Q :std_logic_vector(3 downto 0);
signal Y :std_logic;
begin
  process(H)
   begin
   if (H'event and H='1') then
     if Y = '1' then Q <= Q + 1;
     elsif START = '1' then Q <= "0000";
     else Q <= Q;
     end if;
   end if;
  end process;
  Y <= '1' when Q /= N else '0';
  PULS <= Q(0);
end ARCH_GEN_PULS;
```

Les équations générées qui sont données ci-dessous, sont identiques à celles issues de la description en ABEL.

```
q_3.D = puls.Q * /q_3.Q * q_2.Q * q_1.Q + /start * /puls.Q * q_3.Q
        + q_3.Q * /q_1.Q + q_3.Q * /q_2.Q
q_2.D = puls.Q * /q_2.Q * q_1.Q + /puls.Q * /q_3.Q * q_2.Q
        + /start * /puls.Q * q_2.Q + q_2.Q * /q_1.Q
q_1.D = /puls.Q * /q_2.Q * q_1.Q + /puls.Q * /q_3.Q * q_1.Q
        + /start * /puls.Q * q_1.Q + puls.Q * /q_1.Q
puls.D = /puls.Q * /q_1.Q + /puls.Q * /q_2.Q + /puls.Q * /q_3.Q
```

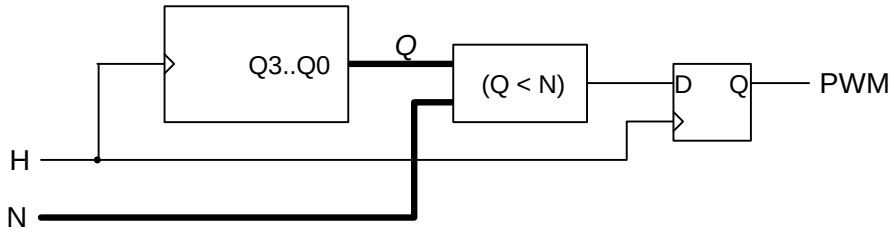
***AUTRES FONCTIONS***

**Génération d'un signal modulé en largeur d'impulsion (PWM)**

Il suffit de faire la comparaison entre la valeur de la sortie d'un compteur binaire synchrone et la valeur servant à moduler.

```
MODULE pwm

"Inputs
H,N3..N0 pin; N = [N3..N0];"Valeur caracteristique du rapport cyclique

"Outputs
Q3..Q0 node istype 'reg'; Q = [Q3..Q0];
PWM pin istype 'reg';

Equations
[Q,PWM].clk = H;
Q := Q + 1;
PWM := Q < N;

Test_vectors
([H,N] -> [Q,PWM])
@repeat 40 {[.C., 0] -> .X.;}
@repeat 40 {[.C., 1] -> .X.;}
@repeat 40 {[.C., 5] -> .X.;}
@repeat 40 {[.C.,10] -> .X.;}
@repeat 40 {[.C.,14] -> .X.;}
@repeat 40 {[.C.,15] -> .X.;}

END
```

La solution fournie par le logiciel est décrite dans le schéma bloc ci-dessous.




On aurait pu utiliser la sortie combinatoire du bloc (Q < N) , ce qui se serait écrit `PWM = Q < N;` au lieu de `PWM := Q < N;`

Cette solution présente le risque de voir apparaître des impulsions parasites. De plus la simplification que l'on constate alors sur le schéma bloc ne correspond à rien pour un circuit logique programmable. En effet la cellule qui génère le bloc (Q < N) contient une bascule (c'est l'organisation interne des cellules logiques de ces circuits). Si cette bascule n'est pas utilisée, elle est tout de même présente dans la structure.

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity PWM is
port (H :in std_logic;
      N :in std_logic_vector(3 downto 0);
      PWM :out std_logic);
end PWM;

architecture ARCH_PWM of PWM is
signal Q :std_logic_vector(3 downto 0);
begin
  process(H)
   begin
   if (H'event and H='1') then
     Q <= Q + 1;
     if Q < N then PWM <= '1'; else PWM <= '0';
     end if;
   end if;
  end process;
end ARCH_PWM;
```

**Compteur-décompteur à butées**

Il s'agit d'un compteur-décompteur binaire générant de manière cyclique une séquence de comptage jusqu'à une butée haute puis une séquence de décomptage jusqu'à une butée basse puis de nouveau la séquence de comptage et ainsi de suite.
Si l'on raccorde les sorties binaires de ce système à un convertisseur analogique-numérique, on obtiendra une forme d'onde triangulaire en marche d'escalier.

```
MODULE triangle

"Inputs
H,RAZ pin;

"Outputs
Q3..Q0 pin istype 'reg'; Q = [Q3..Q0];
SENS node istype 'com';"comptage 0; décomptage 1

"Constantes
UP = 0;   DN = 1;
BUTEE_HAUTE = 10;
BUTEE_BASSE = 5;

Equations
Q.CLK = H;
Q.ACLR = RAZ;
WHEN (SENS == UP) THEN Q := Q + 1; ELSE Q := Q - 1;

"SENS.CLK = H;
"SENS.ACLR = RAZ;
     WHEN (Q >= BUTEE_HAUTE) THEN SENS = DN;
ELSE WHEN (Q <= BUTEE_BASSE) THEN SENS = UP;
                             ELSE SENS = SENS;

Test_vectors
([H,RAZ,!Q] -> [SENS,Q])
"Demarrage du compteur apres un RAZ
  [0,1,.X.] -> .X.;
  @repeat 30 {[.C.,0,.X.] -> .X.;}
"Demarrage du compteur avec une valeur superieure aux butees
  [.P.,0,14] -> .X.;
  @repeat 30 {[.C.,0,.X.] -> .X.;}

END
```

Le logiciel fournit les équations suivantes :

```
Q3 := (Q3 & !Q0 & !SENS # Q3 & Q1 & SENS # Q3 & !Q2 & Q0 # Q3 & Q2 & !Q1
     # !Q3 & Q2 & Q1 & Q0 & !SENS # !Q3 & !Q2 & !Q1 & !Q0 & SENS);
Q2 := (Q2 & !Q1 & !SENS # Q2 & Q0 & SENS # Q2 & Q1 & !Q0
     # !Q2 & Q1 & Q0 & !SENS # !Q2 & !Q1 & !Q0 & SENS);
Q1 := (Q1 & !Q0 & !SENS # !Q1 & Q0 & !SENS # !Q1 & !Q0 & SENS # Q1 & Q0 & SENS);
Q0 := (!Q0);
SENS = (Q2 & Q1 & SENS # Q3 & Q1 # Q3 & Q2 # Q3 & SENS);
```

La simplification des équations ne permet pas de faire apparaître la forme d'un compteur décompteur. On remarque tout de même que le signal SENS est issu d'une cellule de mémorisation asynchrone de type bascule RS.

Le signal SENS aurait tout aussi bien pu être issu d'un registre (voir remarque dans l'exemple précédent). La solution présentée ici à l'avantage de compter et décompter en atteingnant juste les valeurs des butées. Si l'on avait utilisé un registre, il y aurait eu un dépassement des butées d'une unité à chaque fois.

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity TRIANGLE is
port (H,RAZ :in std_logic;
      Q :out std_logic_vector(3 downto 0));
end TRIANGLE;

architecture ARCH_TRIANGLE of TRIANGLE is
constant BUTEE_HAUTE : std_logic_vector(3 downto 0) := "1010";
constant BUTEE_BASSE : std_logic_vector(3 downto 0) := "0101";
constant UP : std_logic := '0';
constant DN : std_logic := '1';
signal X :std_logic_vector(3 downto 0);
signal SENS :std_logic;
begin
  process(H,RAZ)
   begin
   if RAZ = '1' then X <= "0000";
   elsif (H'event and H='1') then
     if SENS = UP then X <= X + 1; else X <= X - 1;
     end if;
   end if;
  end process;
  SENS <= DN when (X >= BUTEE_HAUTE)
     else UP when (X <= BUTEE_BASSE)
     else SENS;
  Q <= X;
end ARCH_TRIANGLE;
```

Les équations générées sont :

```
q_0.D = /q_0.Q
q_1.D = q_1.Q * /sens * /q_0.Q + /q_1.Q * sens * /q_0.Q
       + /q_1.Q * /sens * q_0.Q + q_1.Q * sens * q_0.Q
q_2.D = /q_2.Q * /q_1.Q * sens * /q_0.Q + /q_2.Q * q_1.Q * /sens * q_0.Q
       + q_2.Q * /sens * /q_0.Q + q_2.Q * /q_1.Q * q_0.Q + q_2.Q * q_1.Q * sens
q_3.D = /q_3.Q * /q_2.Q * /q_1.Q * sens * /q_0.Q
       + /q_3.Q * q_2.Q * q_1.Q * /sens * q_0.Q + q_3.Q * q_1.Q * /q_0.Q
       + q_3.Q * sens * q_0.Q + q_3.Q * /q_2.Q * /sens + q_3.Q * q_2.Q * /q_1.Q
sens = q_2.Q * q_1.Q * sens + q_3.Q * sens + q_3.Q * q_1.Q + q_3.Q * q_2.Q
```

On peut constater que les résultats sont équivalents en terme d'utilisation des ressources des circuits cibles mais que les regroupement effectuées sont parfois différentes (voir Q2 et Q3).

**Compteur GRAY (binaire réfléchi)**

```
MODULE gray

"Inputs
H,RAZ pin;

"Outputs
Q2..Q0 pin istype 'reg'; Q = [Q2..Q0];

Equations
Q.clk = H;
Q.aclr = RAZ;

Truth_table
(Q :> Q)
 0 :> 1;
 1 :> 3;
 3 :> 2;
 2 :> 6;
 6 :> 7;
 7 :> 5;
 5 :> 4;
 4 :> 0;

Test_vectors
([H,RAZ] -> Q)
            [ 0 ,1] -> .X.;
@repeat 10 {[.C.,0] -> .X.;}

END
```

L'utilisation d'une table de vérité de type séquentielle permet une description efficace.
Les équations générées sont :

```
Q2 := (Q1 & !Q0 # Q2 & Q0);
Q1 := (Q1 & !Q0 # !Q2 & Q0);
Q0 := (!Q2 & !Q1 # Q2 & Q1);
```

Elles font apparaître une complexité équivalente pour chaque sortie.

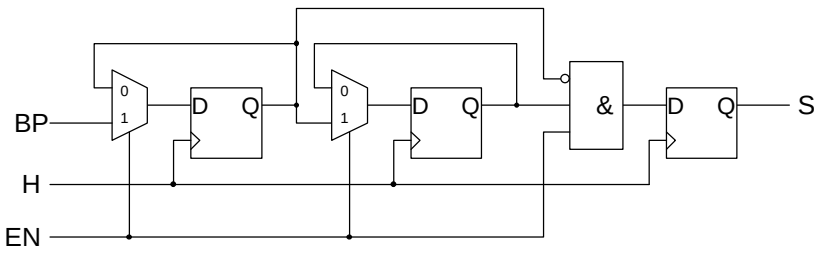
Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity GRAY is
port (H,R :in std_logic;
      Q :out std_logic_vector(2 downto 0));
end GRAY;

architecture ARCH_GRAY of GRAY is
signal X :std_logic_vector(2 downto 0);
begin
  process(H,R)
   begin
   if R='1' then X <= "000";
   elsif (H'event and H='1') then
    if X = "000" then X <= "001";
    elsif X = "001" then X <= "011";
    elsif X = "011" then X <= "010";
    elsif X = "010" then X <= "110";
    elsif X = "110" then X <= "111";
    elsif X = "111" then X <= "101";
    elsif X = "101" then X <= "100";
    elsif X = "100" then X <= "000";
    end if;
   end if;
  end process;
  Q <= X;
end ARCH_GRAY;
```

**Anti-rebonds pour bouton poussoir**

```
MODULE anti_reb

"Inputs
H,EN,BP pin;"BP est actif a 0

"Outputs
Q1,Q0 node istype 'reg'; Q =[Q1,Q0];
S pin istype 'reg';

Equations
[S,Q].clk = H;
when (EN == 1) then Q := [Q0,BP]; else Q := Q;
S := EN & Q1 & !Q0;

Test_vectors
([H,EN,BP] -> [Q0,Q1,S])
@repeat 3 {[.C.,0,1] -> .X.;}
           [.C.,1,1] -> .X.;
@repeat 3 {[.C.,0,1] -> .X.;}
           [.C.,1,1] -> .X.;
@repeat 3 {[.C.,0,1] -> .X.;}
           [.C.,1,1] -> .X.;
@repeat 2 {[.C.,0,1] -> .X.;}
           [.C.,0,0] -> .X.;
           [.C.,1,0] -> .X.;
@repeat 3 {[.C.,0,1] -> .X.;}
           [.C.,1,0] -> .X.;
           [.C.,0,1] -> .X.;
           [.C.,0,0] -> .X.;
           [.C.,0,1] -> .X.;
           [.C.,1,0] -> .X.;
@repeat 3 {[.C.,0,0] -> .X.;}
           [.C.,1,0] -> .X.;
@repeat 3 {[.C.,0,1] -> .X.;}
           [.C.,1,1] -> .X.;
@repeat 3 {[.C.,0,1] -> .X.;}
           [.C.,1,1] -> .X.;
@repeat 3 {[.C.,0,1] -> .X.;}

END
```

Les équations générées peuvent se traduire par le schéma structurel suivant :




La solution exposée ci-dessus trouve son efficacité lorsque la durée séparant 2 fronts actifs de l'horloge sur les bascules générant Q0 et Q1 est supérieure à la pseudo période des éventuelles rebonds. Si l'horloge a une fréquence élevée, une valeur utilisable pour la période du signal EN est de quelques dizaines de millisecondes.
Cette entrée d'autorisation d'horloge EN permet d'insérer ce montage dans un ensemble totalement synchrone qui utilise une horloge de fréquence élevée. Dans ce cas, le registre fournissant le signal S, peut être omis, le synchronisme des fonctions suivantes le remplaceront.

Une description en langage VHDL donnant un résultat et des équations identiques est fournie ci-dessous.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.std_arith.all;

entity ANTI_REB is
port (H,EN,BP :in std_logic;
      S :out std_logic);
end ANTI_REB;

architecture ARCH_ANTI_REB of ANTI_REB is
signal Q :std_logic_vector(2 downto 0);
begin
  process(H)
   begin
   if (H'event and H='1') then
     if EN = '1' then Q(1 downto 0) <= Q(0) & BP;
     else Q(1 downto 0) <= Q(1 downto 0);
     end if;
     Q(2) <= EN and Q(1) and (not Q(0));
   end if;
  end process;
S <= Q(2);
end ARCH_ANTI_REB;
```

# ANNEXE 1 : ELEMENTS DE LANGAGE ABEL

Le langage de description ABEL-HDL (***H**ardware **D**escription **L**anguage*) est un langage créer par la société DATA I/O. Comme le sigle HDL l'indique, il s'agit d'un langage de description de structure. Dans certains ouvrages, la traduction du sigle HDL est ***H**ight Level **D**escription **L**anguage* ce qui signifie aussi que ce langage permet une description des structures au niveau de son comportement.
C'est ce niveau de description qui est utilisé tout au long du document et qui est à privilégier lors des descriptions de fonctions logiques. Le niveau de description structurelle est essentiellement rencontré dans les fichiers générés par les logiciels de synthèse de circuits logiques programmables. Ceux-ci permettent de renseigner l'utilisateur sur la manière précise dont le circuit sera "câblé".

***Quelques règles :***

→ Le langage ABEL ne fait aucune distinction entre majuscule et minuscule pour les mots réservés ou mots clés. Par contre, il les différencie dans les noms attribués par l'utilisateur (identificateurs).
→ Ces identificateurs doivent être constitués uniquement à partir des 26 lettres de l'alphabet (non accentués), des chiffres 0 à 9 et du caractère "_" (souligné), commencer par une lettre, ne pas se terminer par le souligné et ne pas dépasser 31 caractères (éviter les noms trop longs : 8 à 10 caractères maximum semble raisonnable).
→ Les commentaires débutent par " (double guillemet) et se termine par la fin de ligne ou un autre double guillemet.
→ Les chaînes de caractères sont encadrées par des guillemets simples ' (attention, on utilise fréquemment le guillemet simple en tant qu'apostrophe dans les commentaires et ceci créé une erreur : la solution consiste à mettre 2 guillemets simples).
→ Les instructions se terminent par ";" (point virgule).
→ Toutes descriptions en langage ABEL est consistuée de `MODULE` organisée en :
- une zone de déclaration des entrées (`pin`), des sorties (`pin`) et des signaux internes (`node`),
- une zone de description de la fonction logique du module (`Equations`, `Truth_table`, `State_diagram`)
- et de manière non obligatoire, une zone réservée à la simulation fonctionnelle (`Test_vectors`).

Afin de comprendre rapidement les éléments importants du langage ABEL-HDL, on fournit 2 exemples de fonctions électroniques classiques décrites au niveau comportemental. Ces exemples sont commentés dans la partie droite de la feuille en face des points développés.

Le 1er exemple est combinatoire. Il s'agit d'un additionneur de 2 mots de 4 bits.

```
MODULE ADD4
```

`MODULE` est un mot réservé qui annonce le début de la description nommée `ADD4`. Le fichier contenant cette description doit porter le même nom que le module (ici le fichier s'appelle `ADD4.ABL`). De nombreuses parties du logiciel de traitement du langage ABEL fonctionnant sous DOS, le nom du fichier et donc du module ne doit pas **dépasser 8 caractères**. La fin de la description du module correspond au mot clé `END` placé en fin de fichier.

```
"Inputs
A3..A0,B3..B0 pin;
A = [A3..A0];
B = [B3..B0];
I = [B3..B0,A3..A0];

"Outputs
S3..S0,CO pin istype 'com';
S = [S3..S0];
```

Cette partie est la zone des **déclarations**.
C'est ici que l'on déclare les bornes (`pin`) ou les noeuds internes (`node`) ainsi que leurs attributs (`istype 'com'` pour signifier que la sortie est de type combinatoire).
L'écriture `A3..A0` est une écriture équivalente à `A3,A2,A1,A0`.
La déclaration `A = [A3..A0]` permet de définir un ensemble sous un seul nom. Ce n'est qu'une facilité d'écriture car ABEL ignore la notion de bus.

```
Equations
S = A + B;
when (S < A) # (S < B)
then CO = 1 else CO = 0;
```

Ici commence la zone de **description** du module.
Le mot clé `Equations` marque le début d'un ensemble d'équations booléennes (par exemple l'équation donnant `S`) ou comportementales (l'équation donnant `CO`) décrivant le comportement logique du module.
La syntaxe utilisée pour définir une affectation distingue les affectations combinatoires (signe égal simple " = "), des affectations séquentielles (*registered*)(signe " := ").
Il existe d'autres mots clé permettant de décrire une fonction logique. Il s'agit des mots clé "`truth_table`" (table de vérité) et "`state_diagram`" (diagramme d'état).

```
Test_vectors
([I] -> [S,CO])
@const n = 0;
@repeat 256 {
   [n] -> .X.;
   @const n = n + 1;}
```

`Test_vectors` est le mot clé qui débute la zone des instructions réservées au simulateur.
La ligne suivante précise, à droite du symbole "`->`" les entrées et à gauche les sorties à visualiser.
Les lignes suivantes permettent, grâce à une boucle, de tester toutes combinaisons des 8 entrées.

```
END
```

Fin de la description du module `ADD4`.

**Syntaxe utilisée pour générer les vecteurs de test**

→ La directive `@const` permet une nouvelle déclaration de constante en dehors de zone des déclarations. Elle est donc utilisée ici pour déclarer que `n` est une nouvelle constante dont la valeur est 0.

→ La directive `@repeat` permet de répéter le bloc entre accolades { } un nombre de fois égal à la valeur mentionnée (ou à la valeur du résultat fourni par une expression). Ici, le simulateur répétera 256 fois le bloc entre accolades.
Ce bloc comprend 2 actions :
- L'action `n -> .X.;` demande au simulateur de visualiser les grandeurs des sorties avec les entrées valant `I` = `n` = 0.
- L'action `@const n = n+1;` attribue à la constante `n` une nouvelle valeur (`n` +1). Il s'agit de l'incrémentation de `n`.

Le 2ième exemple est un exemple séquentiel : il s'agit d'un compteur binaire synchrone modulo 60 à remise à zéro asynchrone (RAZ) et synchrone (RESET), à autorisation d'horloge (HOLD) et à sortie retenue active à 0 (HOLDO).

```
MODULE mod60a

"Inputs
HOLD, RESET, RAZ pin 2,3,4;
clock pin 1;

"Outputs
Q5,Q4,Q3,Q2,Q1,Q0 pin
      19,18,17,16,15,14 istype 'reg';
out = [Q5..Q0];
HOLDO pin 13 istype 'com';



Equations
out.ACLR = RAZ;


out.CLK = clock;



WHEN (RESET # (out == 59)) THEN out := 0;
      ELSE WHEN (HOLD) THEN out := out;
           ELSE out := out + 1;
HOLDO = (out != 59);



Test_vectors
([clock,RAZ,RESET,HOLD,!out] -> [out,HOLDO])
"test du RAZ asynchrone
[ 0 ,1,0,0,.X.] -> .X.;
"test de comptage
[.C.,0,0,0,.X.] -> .X.;
[.C.,0,0,0,.X.] -> .X.;
"test du RESET synchrone
[.C.,0,1,0,.X.] -> .X.;
[.C.,0,0,0,.X.] -> .X.;
"test de l'autorisation de comptage
[.C.,0,0,1,.X.] -> .X.;
"test de comptage
@repeat 10 {[.C.,0,0,0,.X.] -> .X.;}
"prépositionnement pour eviter le comptage
jusqu'a 60
[.P.,0,0,0, 55] -> .X.;
"test de comptage au dela de 60
@repeat 10 {[.C.,0,0,0,.X.] -> .X.;}
"test du RAZ asynchrone
[ 0 ,1,0,0,.X.] -> .X.;
"test de comptage
[.C.,0,0,0,.X.] -> .X.;

END
```

→ Dans les déclarations de noms de bornes (`pin`), celles-ci se sont vues affecter un numéro de broche.

→ On peut remarquer que les bornes de sortie sont ici des 2 types évoqués dans les commentaires du 1er exemple : `'reg'` ou `'com'` (registered ou combinatoire).

→ Toutes sorties de type `'reg'` doit comporter un signal d'horloge (`.CLK`).

→ Eventuellement ces types de sorties peuvent comporter d'autres signaux comme :
- des mises à 0 ou 1 asynchrone (`.ACLR` ou `.ASET`),
- des mises à 0 ou 1 synchrone (`.CLR` ou `.SET`) etc..

→ L'équation `out.ACLR = RAZ;` indique donc que le signal `RAZ` commande la mise à 0 asynchrone des signaux de sortie formant `out`.

→ L'équation `out.CLK = clock;` indique que les signaux de sortie formant `out` (`[Q5..Q0]`) sont issus de registres dont l'horloge est `clock` (structure synchrone).

→ L'équation donnant `out` est écrite en utilisant l'affectation conditionnelle `WHEN..THEN..ELSE`
L'équation donnant `HOLDO` se lit :
`HOLDO` vaut 1 si `out` est différent de 59.

→ On remarquera que l'affectation dans l'équation donnant `out` est indiquée par un signe "`:=`" qui signifie qu'il s'agit d'une sortie d'un système logique séquentiel.

→ Ici les vecteurs de test sont utilisés pour vérifier les différentes possibilités de fonctionnement du compteur.

→ Une nouvelle constante prédéfinie (`.C.`) est utilisée ici pour signifier que la variable `clock` va être une impulsion positive (0→1→0).
D'autres constantes prédéfinies existent :
- `.D.` : front descendant (1→0),
- `.F.` : entrée ou sortie flottante,
- `.K.` : impulsion négative (1→0→1),
- `.P.` : signal de prépositionnement de registre
- `.U.` : front montant (0→1),
- `.X.` : non déterminé,
- `.Z.` : variable 3 états.

# ANNEXE 2 : ELEMENTS DE LANGAGE VHDL

Le langage de description VHDL (***V**ery Hight Speed Integrated Circuit, **H**ardware **D**escription **L**anguage* ou bien dans d'autres ouvrages ***V**ery Hight Scale Integrated Circuit, **H**ight Level **D**escription **L**anguage*) est un langage qui a été créé dans les années 70 pour le développement de circuits intégrés logiques complexes. Il doit son succès, essentiellement, à sa standardisation en Décembre 1987 par l'*Institute of Electrical and Electronics Engineers* sous la référence IEEE1076 qui a permis d'en faire un langage unique pour la description, la modélisation, la simulation, la synthèse et la documentation.

***Quelques règles :***

→ Le langage VHDL ne fait aucune distinction entre majuscule et minuscule. Pour faciliter la lecture, de nombreux auteurs écrivent en minuscule les mots réservés du langage et en majuscule les noms attribués par l'utilisateur.

→ Ces noms attribués par l'utilisateur doivent être constitués uniquement à partir des 26 lettres de l'alphabet (non accentuées), des chiffres de 0 à 9 et du caractère "_" (souligné), commencer par une lettre et ne pas se terminer par le souligné.

→ Les commentaires débutent par "`--`" (double tiret) et se prolongent jusqu'à la fin de la ligne.

→ L'organisation de toute description en langage VHDL fait appel à la paire `entity` - `architecture`. L'entité décrit l'interface externe et l'architecture le fonctionnement interne.

Afin de comprendre rapidement les éléments importants du langage VHDL, on fournit 2 exemples de fonctions électroniques classiques décrites au niveau comportemental. Ces exemples sont commentés dans la partie droite de la feuille en face des points développés.

Le 1er exemple est combinatoire. Il s'agit d'un additionneur de 2 mots de 4 bits.

```
library ieee;
library synth;
use synth.vhdlsynth.all;
use ieee.std_logic_1164.all;


entity ADD_4 is
port (
  A, B : in std_logic_vector(3 downto 0);
  S : buffer std_logic_vector(3 downto 0);
  CO : out std_logic);
end ADD_4;
```

`library` est un mot réservé qui indique l'utilisation d'une bibliothèque.
L'accès aux définitions et fonctions qu'elle contient se fait grâce au mot réservé `use`.

`entity` est aussi un mot réservé qui annonce la définition de l' entité appelée `ADD_4` (vision de l'extérieur de la description).
Le mot-clé `port` permet de définir le nom, le mode et le type des signaux de l'entité.
Il y a 4 modes possibles :

→ `in` : entrée,

→ `out` : sortie,

→ `inout` : entrée/sortie bidectionnelle,

→ `buffer` : sortie servant d'entrée interne à l'entité.

Les types sont nombreux (entiers, réels, tableaux, enregistrement, etc..), mais l'electronicien se servira surtout des `bit` (signal binaire) et des `bit_vector` (bus). Ces types ne peuvent prendre que 2 valeurs `0` et `1` aussi des types dérivés et plus complets ont été définis dans la bibliothèque IEEE, les `std_logic` et les `std_logic_vector` (possibilité, entre autre, d'occuper l'état haute impédance symbolisé par "`Z`" ou un état quelconque "-").

```
architecture ARCH_ADD_4 of ADD_4 is
begin

S <= A + B + CI;


CO <= '1' when
  (S < (A + CI)) or (S < (B + CI))
      else '0';
end ARCH_ADD_4;
```

`architecture` est le mot-clé pour définir le début de la description interne de l'entité.
Ici, il s'agit d'une structure combinatoire (fonctionnement concurent ou parallèle).
Le symbole `<=` est une affectation inconditionnelle. Les 2 membres de cette affectation doivent être de même type.
L'affectation avec `when ... else` est l'affectation conditionnelle réservée aux fonctionnements concurents.
Le fonctionnement est dit "concurent" et correspond bien à la notion d'électronique combinatoire car toutes les opérations comprises entre `begin` et `end` sont effectuées simultanément et s'affectent mutuellement de manière immédiate.

***Opérations avec des "bit" et "bit_vector"***

Les opérations supportées normalement par les signaux de type `bit` et `bit_vector` sont les suivants :

→ Les affectations ou assignations : **`<=`** (pour des variables on utilise **`:=`**)

→ Les opérations logiques : **`and nand or nor xor not`**.

→ Les opérations relationnels entre `bit` ou `bit_vector` , le résultat étant de type `boolean` :
**`=`** (égal), **`/=`** (différent), **`<`** et **`<=`** (inférieur et inférieur ou égal), **`>`** et **`>=`** (supérieur et supérieur ou égal).

→ L'opération de concaténation : **`&`**.

Il existe aussi les opérations de décalage et rotation à droite ou à gauche.

Dans l'exemple donné ici, on réalise l'opération addition "**`+`**" entre deux `bit_vectors`. Ceci n'est normalement pas permis et aurait dû générer une erreur. En effet les opérations arithmétiques (**`+ - * /`** etc...) ne sont utilisables que pour des variables de type `integer` ou `real`.
Devant les demandes des concepteurs de circuits logiques programmables, les éditeurs des compilateurs VHDL ont ajouté des librairies permettant de **surcharger** les opérateurs **`+`** et **`-`** pour les types `bit`, `bit_vector`, `std_logic`, `std_logic_vector` afin de permettre les opérations d'incrémentation et décrémentation très courantes dans la synthèse de fonctions logiques.

Le 2ème exemple est séquentiel. Il s'agit d'un compteur binaire synchrone modulo 60 à remise à zéro asynchrone (`RAZ`) et synchrone (`RESET`), à autorisation d'horloge (`HOLD`) et à sortie retenue active à 0 (`HOLDO`).

```
library ieee;
library synth;
use synth.vhdlsynth.all;
use ieee.std_logic_1164.all;

entity MOD60A is
port (
RESET,RAZ,CLK,HOLD : in std_logic;
Q : out std_logic_vector(5 downto 0);
HOLDO : out std_logic);
end MOD60A;


architecture COMPTEUR of MOD60A is
signal X : std_logic_vector(5 downto 0);
begin


 process(CLK,RAZ)
 begin




if (RAZ='1') then X <= "000000";


  elsif (CLK'EVENT and CLK = '1') then

    if (HOLD = '1') then
      if ((X < 59) and (RESET = '0')) then
        X <= X + 1;
      else X <= "000000";
      end if;
    end if;
  end if;
 end process;
 Q <= X;
 HOLDO <= '0' when (X = 59) else '1';
end COMPTEUR;
```

On retrouve de la même manière que dans l'exemple précédent :

→ les accès aux bibliothèques,

→ la description externe de l'entité, avec ses entrées et sorties,

L'utilisation du `signal` nommé `X` dans l'architecture permet de définir `Q` comme une sortie (autrement, il aurait fallu que `X` soit de type `buffer`). Cette méthode facilite la réutilisation de cette entité. En effet, vue de l'extérieur, `MOD60A` ne présente que des entrées ou des sorties.

→ la description de l'architecture interne.
Celle-ci met en oeuvre un fonctionnement séquentiel et un fonctionnement concurent (l'affectation du signal `HOLDO`).

Une fonction séquentielle se décrit comme un processus (`process`) qui est effectué à chaque changement d'état d'un des signaux auquel il est déclaré sensible (ici `CLK` et `RAZ`).
Les modifications apportés aux valeurs de signaux par les instructions prennent effet à la fin du processus.

A l'intérieur d'un processus, on utilise l'affectation inconditionnelle `<=` (comme en combinatoire) et l'affectation conditionnelle `if ... then ... else ... end if`.

La description se lit alors assez naturellement.
SI (`RAZ` est à 1) ALORS les sorties sont mises à zéro (de manière asynchrone puisqu'indépendantes de l'horloge),
SINON, SI il y a un front montant sur `CLK` , ALORS ,
SI (`HOLD` est à 1) ALORS
SI ((les sorties affichent une valeur plus petite que 59) ET (que le signal `RESET` est à 0)) ALORS
il y a incrémentation des sorties
SINON les sorties sont remises à zéro.

L'élaboration de `HOLDO` est faite en dehors du processus donc en mode concurent (combinatoire).