PROCESSEURS ET ARCHITECTURES NUMÉRIQUES (PAN)

SUPPORT DE COURS

ANNÉE 2018/2019

SUMANTA CHAUDHURI
JEAN-LUC DANGER
GUILLAUME DUC
TARIK GRABA
ULRICH KÜHNE
YVES MATHIEU
ALEXIS POLTI
LAURENT SAUVAGE

TELECOM PARISTECH

Table des matières

1
De la logique combinatoire à l'arithmétique

Dans ce chapitre, nous allons introduire la notion de logique combinatoire en présentant les bases de la logique booléenne (cf. section 1.1), puis les fonctions logiques élémentaires (cf. section 1.2) ainsi quelques fonctions plus complexes mais souvent utilisées (cf. section 1.3). Nous verrons ensuite comment représenter des nombres (cf. section 1.4) et comment réaliser des opérations sur ces nombres (cf. section 1.5). Enfin, nous introduirons le concept de temps de propagation dans les opérateurs logiques (cf. section 1.6).

1.1 La logique booléenne

1.1.1 Introduction

Soit E l'ensemble à deux éléments $0, 1$.

Une *variable logique* est un élément de E. Elle ne possède donc que deux états : 0 ou 1.

Une *fonction logique* est une application de $E \times E \ldots \times E$ dans E qui associe à un n-uplet de variables logiques $(e_0, e_1, \ldots, e_n)$, souvent appelées *entrées*, une variable logique $s = F(e_0, e_1, \ldots, e_n)$, souvent appelée *sortie*.

On distingue deux catégories de fonctions logiques, en fonction de leur comportement temporel.

Une *fonction combinatoire* est une fonction logique pour laquelle la sortie ne dépend que de l'état actuel des entrées :

$$\forall t, s(t) = F(e_0(t), e_1(t), \ldots, e_n(t))$$

Une *fonction séquentielle* est une fonction logique pour laquelle la sortie dépend de l'état actuel de ses entrées mais également de leurs états passés :

$$s(t) = F(e_0(t), e_1(t), \ldots, e_n(t), e_0(t - t_1), e_1(t - t_1) \ldots)$$

Dans ce chapitre, nous ne traiterons que des fonctions combinatoires. Les fonctions séquentielles seront introduites dans le chapitre 2.

1.1.2 Représentation des fonctions logiques

Il existe plusieurs méthodes pour décrire une fonction logique combinatoire. Ces méthodes sont toutes équivalentes et le choix de l'une ou l'autre ne va dépendre que du contexte d'utilisation.

Table de vérité La première méthode consiste à lister, pour chacune des valeurs possibles de ses entrées, la valeur de la sortie de la fonction. Cette liste, présentée très souvent sous forme de table, est appelée *table de vérité*.

L'inconvénient majeur de cette méthode est que cette table peut être très grande. En effet, si la fonction prend en entrée N variables logiques, il faut 2^N lignes dans cette table pour lister l'ensemble des valeurs possibles des entrées.

Exemple d'une table de vérité d'une fonction prenant en entrée deux variables logiques a et b :

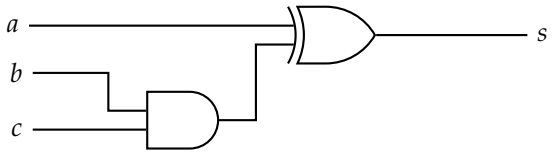
a	b	s
0	0	0
0	1	0
1	0	0
1	1	1

TABLE 1.1: Exemple d'une fonction prenant en entrée deux variables logiques a et b

Équation analytique Cette méthode consiste à donner l'équation de la sortie de la fonction en fonction de ses entrées. La liste et la signification des opérateurs seront données dans la section sur les fonctions logiques de base.

Exemple d'une fonction prenant en entrée trois variables logiques a, b et c : $s = a \cdot b + \bar{c}$.

Représentation schématique graphique Cette méthode consiste à représenter graphiquement une fonction booléenne à l'aide à l'aide du schéma normalisé [1] des fonctions de base et de traits servant à indiquer la "connexion" d'une sortie à une entrée.



1. Il existe plusieurs normes pour représenter ces fonctions de base. Dans ce cours, nous utiliserons la norme "américaine" (issue de ANSI 91-1984) car c'est l'une des plus utilisées. Il existe également une norme "européenne" (IEC 60617-12).

Description fonctionnelle Cette méthode consiste à décrire, en langage naturel, le comportement de la sortie en fonction des entrées de la fonction.

Exemple : la sortie s vaut 1 si, et seulement si, au moins une des entrées a ou b vaut 1.

Langage de description de matériel Cette méthode consiste à décrire la fonction dans un langage particulier, appelé *langage de description de matériel* (*Hardware Description Language* — HDL), facilement compréhensible par un ordinateur. Il en existe plusieurs et dans ce cours nous utiliserons SystemVerilog.

Ce langage sera introduit tout au long de ce cours et une synthèse est présentée page 87.

Exemple :

```
logic A, B, C, S;
always@(*)
   S <= A ^ B | C;
```

1.2 *Fonctions logiques élémentaires*

Cette section présente les fonctions logiques élémentaires, également appelées *portes logiques*, couramment utilisées par la suite.

1.2.1 *L'inverseur (NOT)*

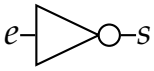
Description

La sortie vaut 1 si, et seulement si, l'entrée vaut 0.

Table de vérité *Équation* *Symbole*

e	s
0	1
1	0

$s = \overline{e}$



Description en SystemVerilog

```
logic s, e;
always@(*)
   s <=  ~e;
```

Explications :

— La première ligne permet de déclarer deux *signaux* (de type `logic` donc sur 1 bit), nommés `s` et `e`

— La deuxième ligne indique que ce qui suit (donc la troisième ligne) doit être ré-évalué dès que nécessaire (dans notre cas, dès que `e` va changer, ce qui correspond bien à un comportement combinatoire)

— `<=` est l'opérateur d'affectation (c'est-à-dire que la valeur située à droite de cet opérateur est affectée au signal situé à gauche de l'opérateur)

— `~` est l'opérateur unaire d'inversion bit-à-bit (donc notre opérateur NON)

— La troisième ligne indique donc qu'il faut affecter au signal `s` le complément de la valeur du signal `e`

1.2.2 *Le ET (AND)*

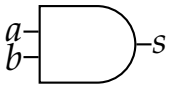
Description

La sortie vaut 1 si, et seulement si, les deux entrées valent 1. [2]

Table de vérité *Équation* *Symbole*

a	b	s
0	0	0
0	1	0
1	0	0
1	1	1

$s = a \cdot b$



2. La fonction ET peut être aussi interprétée comme une fonction de forçage à zéro d'un signal : dans l'expression $s = valid \cdot b$, le signal s ne vaut b que si $valid = 1$ sinon il vaut 0.

Description en SystemVerilog

```
logic s, a, b;
always@(*)
   s <=  a & b;
```

En SystemVerilog, & représente l'opérateur ET bit-à-bit.

1.2.3 Le OU (OR)

Description

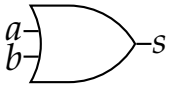
La sortie vaut 0 si, et seulement si, les deux entrées valent 0.[3]

3. La fonction OU peut être aussi interprétée comme une fonction de forçage à un d'un signal : dans l'expression $s = force + b$, le signal s ne vaut b que si $force = 0$ sinon il vaut 1.

Table de vérité *Équation* *Symbole*

a	b	s
0	0	0
0	1	1
1	0	1
1	1	1

$s = a + b$



Description en SystemVerilog

```
logic s, a, b;
always@(*)
   s <=  a | b;
```

En SystemVerilog, | représente l'opérateur OU bit-à-bit.

1.2.4 Le NON ET (NAND)

Table de vérité

Description

Il s'agit de la fonction complémentaire du ET.

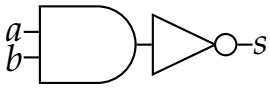
La sortie vaut 0 si, et seulement si, les deux entrées valent 1.

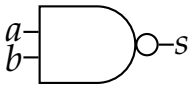
Table de vérité *Équation*

a	b	s
0	0	1
0	1	1
1	0	1
1	1	0

$s = \overline{a \cdot b}$

Symbole





Description en SystemVerilog

```
logic s, a, b;
always@(*)
   s <=  ~(a & b);
```

1.2.5 *Le NON OU (NOR)*

Description

Il s'agit de la fonction complémentaire du OU.
La sortie vaut 1 si, et seulement si, les deux entrées valent 0.

Table de vérité *Équation*

a	b	s
0	0	1
0	1	0
1	0	0
1	1	0

$$s = \overline{a + b}$$

Symbole

Description en SystemVerilog

```
logic s, a, b;
always@(*)
   s <=  ~(a | b);
```

1.2.6 *Le OU exclusif (XOR)*

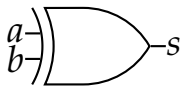
Description

La sortie vaut 1 si une, et seulement une, des deux entrées vaut 1.[4]

Table de vérité *Équation* *Symbole*

a	b	s
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{aligned} s &= a \oplus b \\ &= a \cdot \overline{b} + \overline{a} \cdot b \end{aligned}$$



Description en SystemVerilog

```
logic s, a, b;
always@(*)
   s <=  a ^ b;
```

En SystemVerilog, ^ représente l'opérateur OU exclusif bit-à-bit.

4. La fonction OU exclusif peut être aussi interprétée comme une fonction de sélection entre une donnée et son complémentaire : dans l'expression $s = selcomp \oplus b$, le signal s ne vaut b que si $selcomp = 0$ sinon il vaut $\overline{b}$

1.2.7 *Le NON OU exclusif (XNOR)*

Description

Il s'agit de la fonction complémentaire du OU exclusif.
La sortie vaut 1 si, et seulement si, les deux entrées sont égales.

Table de vérité *Équation*

a	b	s
0	0	1
0	1	0
1	0	0
1	1	1

$$s = \overline{a \oplus b}$$
$$= a \cdot b + \overline{a} \cdot \overline{b}$$

Symbole

Description en SystemVerilog

```
logic s, a, b;
always@(*)
   s <=  ~(a ^ b);
```

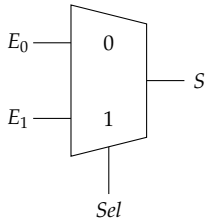
1.3 *Fonctions logiques importantes*

Les fonctions logiques élémentaires vues précédemment dans la section 1.2 permettent de construire toutes les fonctions logiques combinatoires. Dans cette section, nous présenterons quelques fonctions logiques un peu plus élaborées couramment utilisées.

1.3.1 *Le multiplexeur*

Un multiplexeur[5] à N entrées, est une fonction logique dont la sortie est égale à l'une de ses N entrées. Le choix parmi les N entrées se fait grâce à une entrée particulière de sélection.

5. Parfois appelé également fonction d'aiguillage



Multiplexeur à deux entrées Le multiplexeur le plus simple est le multiplexeur à deux entrées, dont le schéma est le suivant :

Si l'entrée *Sel* vaut 0, la sortie S du multiplexeur vaudra la valeur présente sur l'entrée E_0. De même, si *Sel* vaut 1, la sortie S vaudra la valeur présente sur l'entrée E_1.

La table de vérité du multiplexeur à deux entrées est la suivante :

Sel	E_1	E_0	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Le comportement de ce multiplexeur peut également être décrit par l'équation suivante :

$$S = Sel \cdot E_1 + \overline{Sel} \cdot E_0$$

Enfin, il peut également être décrit de plusieurs manières en SystemVerilog :

```
logic S, Sel, E0, E1;
always@(*)
   S <=  (Sel & E1) | (~Sel & E0);
```

Le code précédent est une simple transcription de l'équation analytique du multiplexeur. Il existe cependant d'autres constructions, plus proches des langages informatiques classiques, qui permettent également de décrire le comportement du multiplexeur.

```
logic S, Sel, E0, E1;
always@(*)
   if (Sel) S <= E1;
   else S <= E0;
```

La construction `if...else` fonctionne comme dans les langages informatiques classiques. La condition du test est située entre parenthèses juste après le `if` et doit être une expression renvoyant vrai ou faux. En SystemVerilog, un signal (ici `Sel`) valant 1 est considéré comme vrai (dans ce cas la branche principale du `if` est prise) et un signal valant 0 est considéré comme faux (dans ce cas la branche `else` est prise).

```
logic S, Sel, E0, E1;
always@(*)
   case(Sel)
      1'b0: S <= E0;
      1'b1: S <= E1;
   endcase
```

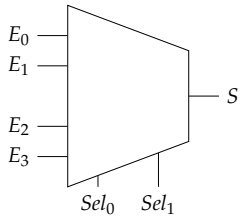
La construction `case...endcase`[6] permet de tester simplement les différentes valeurs[7] possibles de la condition (ici `Sel`) sans avoir à imbriquer de nombreux `if...else`.

6. N'oubliez pas le `endcase`.

7. La construction `1'b0` est décrite en détail page 16. Ici, `1'b0` représente la valeur logique 0 et `1'b1` représente la valeur logique 1.

Multiplexeur à n entrées On peut généraliser et disposer de multiplexeurs à 2^N entrées. Dans ce cas, il faut N entrées de sélection afin de pouvoir choisir une parmi les 2^N entrées.

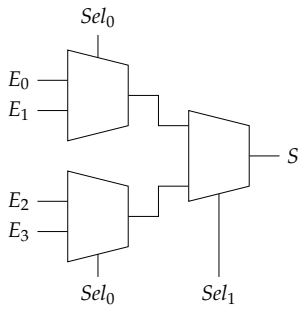
Exemple, le multiplexeur à 4 entrées :



dont l'équation est :

$$S = \overline{Sel_1} \cdot \overline{Sel_0} \cdot E_0 + \overline{Sel_1} \cdot Sel_0 \cdot E_1 + Sel_1 \cdot \overline{Sel_0} \cdot E_2 + Sel_0 \cdot Sel_1 \cdot E_3$$

Ce multiplexeur peut être réalisé grâce à trois multiplexeurs à 2 entrées :



1.3.2 *Le décodeur*

Un décodeur est une fonction logique à N entrées et 2^N sorties[8] dont une, et une seule, sortie vaut 1, le numéro de cette sortie active étant la valeur présente sur l'entrée, considérée comme un nombre entier codé sur N bits (voir section 1.4.1 sur la représentation des nombres entiers).

8. On utilise également la terminologie décodeur N vers 2^N, exemple : décodeur 2 vers 4

Exemple : Décodeur 2 vers 4 La table de vérité du décodeur 2 vers 4 est la suivante :

E_1	E_0	S_0	S_1	S_2	S_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Les équations logiques des sorties sont les suivantes :

$$S_0 = \overline{E_0} \cdot \overline{E_1}$$
$$S_1 = E_0 \cdot \overline{E_1}$$
$$S_2 = \overline{E_0} \cdot E_1$$
$$S_3 = E_0 \cdot E_1$$

De même que pour le multiplexeur, il existe plusieurs manières pour décrire le décodeur en SystemVerilog :

```
logic E0, E1, S0, S1, S2, S3, S4;
always@(*)
begin
   S0 <= ~E0 & ~E1;
   S1 <=  E0 & ~E1;
   S2 <= ~E0 &  E1;
   S3 <=  E0 &  E1;
end
```

Il s'agit ici simplement d'une transcription directe des équations décrivant les sorties en fonction des entrées. Notez ici l'utilisation de la construction `begin...end` nécessaire lorsque plusieurs opérations (affectation, `if...else`, `case...endcase`, etc.) sont décrites au sein du même bloc `always`.

```
logic S0, S1, S2, S3, S4;
logic [1:0] E;
always@(*)
begin
   S0 <= 1'b0;
   S1 <= 1'b0;
   S2 <= 1'b0;
   S3 <= 1'b0;
   if (E == 2'b00) S0 <= 1'b1;
   if (E == 2'b01) S1 <= 1'b1;
   if (E == 2'b10) S2 <= 1'b1;
   if (E == 2'b11) S3 <= 1'b1;
end
```

Ce code introduit plusieurs nouvelles notions en SystemVerilog :

— `logic [1:0] E` déclare un vecteur de deux bits. Le premier nombre entre crochet (ici 1) indique l'indice du bit de poids fort[9] dans le vecteur et le second nombre (ici 0) indique l'indice du bit de poids faible. La dimension en bits du vecteur est donc la différence entre ces deux nombres plus un (donc ici $1 - 0 + 1 = 2$). On peut accéder à la valeur d'un des bits du vecteur à l'aide de l'opérateur `[]` (donc ici `E[1]` est le bit de poids fort et `E[0]` le bit de poids faible).

— Un vecteur de bits (comme E), peut être comparé avec un nombre. Le contenu du vecteur est alors considéré comme un nombre entier représenté en binaire.

— Dans un bloc `always`, si plusieurs valeurs sont affectées à un même signal, la dernière (dans l'ordre d'écriture du code) "gagne". Cette propriété est utilisée ici pour affecter 0 par défaut aux sorties, sauf si elles sont concernées par la suite par un des `if`.

9. La définition de bit de poids fort/faible est donnée plus loin dans la section 1.4.1

```
logic S0, S1, S2, S3, S4;
logic [1:0] E;
```

```
always@(*)
begin
   S0 <= 1'b0;
   S1 <= 1'b0;
   S2 <= 1'b0;
   S3 <= 1'b0;
   case (E)
      2'b00: S0 <= 1'b1;
      2'b01: S1 <= 1'b1;
      2'b10: S2 <= 1'b1;
      default: S3 <= 1'b1; // Correspond au cas E==2'b11
   endcase
end
```

Dans cet exemple, les `if` précédents sont remplacés par une construction `case...endcase`. Le mot clé `default` permet de définir la branche du case prise dans le cas où aucune autre ne l'est. Afin d'éviter des constructions qui ne décriraient pas de la logique combinatoire, il est recommandé de toujours spécifier une branche `default` dans une construction `case...endcase`.

```
logic S0, S1, S2, S3, S4;
logic [1:0] E;
always@(*)
begin
   S0 <= (E == 2'b00);
   S1 <= (E == 2'b01);
   S2 <= (E == 2'b10);
   S3 <= (E == 2'b11);
end
```

L'opérateur == renvoie vrai ou faux en fonction du résultat de la comparaison. Donc `(E == 2'b00)` renvoie vrai si les deux bits de `E` sont à 0 et faux sinon. Lorsque l'on affecte le booléen vrai à un signal (ici `S0`), ce signal prend la valeur 1. De même, si on affecte le booléen faux à un signal, ce dernier prend la valeur 0 [10].

10. Il aurait été possible d'utiliser la construction `if (E == 2'b00) S0 <= 1'b1; else S0 <= 1'b0;` mais elle est fortement déconseillée car très inélégante.

1.4 *Représentation des nombres*

Jusqu'à présent, nous avons traité uniquement des variables logiques et réalisé des opérations logiques. Nous allons voir dans cette section comment des nombres peuvent être représentés et manipulés.

1.4.1 *Nombres entiers naturels*

Représentation générale Un entier naturel N se représente, dans une base b par un n-uplet $(a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$ tel que [11] :

$$N = a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \ldots + a_1 \cdot b^1 + a_0 \cdot b^0$$

11. Attention, les opérateurs $+$ et $\cdot$ reprennent ici leur signification mathématique classique, c'est-à-dire l'addition et la multiplication, contrairement au chapitre précédent où ils correspondaient aux opérateurs logiques OU et ET.

Dans cette représentation :

— a_i est le chiffre de rang i et appartient à un ensemble de b symboles (0 à $b-1$)

— a_{n-1} est appelé le chiffre le plus significatif

— a_0 est appelé le chiffre le moins significatif

Les bases [12] les plus couramment utilisées sont :

— $b = 10$: représentation décimale, $a_i \in \{0,1,2,3,4,5,6,7,8,9\}$

— $b = 16$: représentation hexadécimale, $a_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$

— $b = 8$: représentation octale, $a_i \in \{0,1,2,3,4,5,6,7\}$

— $b = 2$: représentation binaire, $a_i \in \{0,1\}$. Un chiffre binaire est également appelé *bit* (abréviation de l'anglais *binary digit*)

12. Dans la suite, en cas de risque de confusion, la base dans laquelle est représentée un nombre sera indiqué en indice, exemple : $100_{(10)}$, $100_{(2)}$, $100_{(16)}$

Représentation binaire non signée Nous nous intéresserons dans la suite à la représentation des nombres en binaire (base 2). Dans cette base, un nombre entier naturel peut donc se représenter par le n-uplet $(a_{n-1}, a_{n-2}, \ldots, a_1, a_0)$ tel que :

$$N = a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \ldots + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Où :

— a_i est un bit (il appartient à un ensemble de 2 symboles : 0 ou 1)

— a_{n-1} est appelé le bit le plus significatif (MSB : *Most Significant Bit*)

— a_0 est appelé le bit le moins significatif (LSB : *Least Significant Bit*)

Conversion entre hexadécimal et binaire Il est très simple de passer d'un nombre représenté en hexadécimal à un nombre représenté en binaire non signé. Il suffit de concaténer la représentation binaire sur 4 bits de chacun des chiffres hexadécimaux.

Exemple :

$$A1F_{(16)} = 1010\ 0001\ 1111_{(2)}$$

En effet :

$$\begin{aligned}
A1F_{(16)} &= 10 \cdot 16^2 + 1 \cdot 16^1 + 15 \cdot 16^0 \\
&= (1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) \cdot 16^2 \\
&\quad + (0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) \cdot 16^1 \\
&\quad + (1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \cdot 16^0 \\
&= 1 \cdot 2^{11} + 0 \cdot 2^{10} + 1 \cdot 2^9 + 0 \cdot 2^8 \\
&\quad + 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 \\
&\quad + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1010\ 0001\ 1111_{(2)}
\end{aligned}$$

De même, pour passer d'une représentation binaire non signée à une représentation hexadécimale, il suffit de regrouper les bits 4 par 4 (en partant des bits de poids faibles) et en transformant chacun des groupes en un chiffre hexadécimal.

Exemple :

$$10\ 1011\ 0000_{(2)} = 2B0_{(16)}$$

Représentation binaire modulo En pratique, sur le matériel, les nombres vont être représentés et manipulés sur un nombre fixe fini de bits. Donc un nombre N va être représenté, modulo 2^n, sur n bits.

Sur n bits, il est donc possible de représenter les nombres entiers naturels compris dans l'intervalle $[0, 2^n - 1]$.

En SystemVerilog En SystemVerilog, il est souvent nécessaire de manipuler des constantes dans la description d'un module. Or, indiquer la valeur `10` dans un code est ambigu : dans quelle base doit être interprétée cette valeur (s'agit-il de $10_{(2)} = 2_{(10)}$, de $10_{(10)}$ ou de $10_{(16)} = 16_{(10)}$) et sur combien de bits doit être représentée cette valeur ?

Il est recommandé d'utiliser la construction SystemVerilog suivante afin d'être rigoureux : `N'Bvvvv` où :

— `N` est le nombre de bits sur lesquels la valeur doit être représentée

— `B` est la base dans laquelle la valeur qui suit est représentée : `b` pour binaire, `h` pour hexadécimal et `d` pour décimal

— `vvvv` est la valeur de la constante dans la base `B`

Exemples :

```
logic [3:0] P; // Vecteur de 4 bits
logic [7:0] Q; // Vecteur de 8 bits
logic [7:0] R; // Vecteur de 8 bits

always @(*)
begin
   P <= 4'b1001;
   // Identique à P <= 4'd9 ou P <= 4'h9

   Q <= 8'h5A;
   // Identique à Q <= 8'b01011010 ou P <= 8'd90

   R <= 8'd127;
   // Identique à R <= 8'b01111111 ou R <= 8'h7F
end
```

1.4.2 *Nombres entiers relatifs*

Il existe plusieurs méthodes pour représenter les nombres entiers relatifs. La plus couramment utilisée est la représentation en *complément à 2* (CA2).

Comme vu précédemment, sur n bits, les nombres sont représentés modulo 2^n, c'est-à-dire que 2^n a la même représentation que 0, $2^n + 1$ a la même représentation que 1, etc.

Si on étend ce principe aux nombres négatifs, il faudrait que -1 ait la même représentation que $2^n - 1$ (soit $11\ldots11_{(2)}$), que -2 ait la même représentation que $2^n - 2$ (soit $11\ldots10_{(2)}$), etc.

Par *convention*, dans la représentation en complément à 2, les nombres dont le bit de poids fort vaut 1 vont être considérés comme des nombres négatifs et les nombres dont le bit de poids fort vaut 0 vont être considérés comme des nombres positifs [13].

13. Ce qui permet de garder une compatibilité avec la représentation non signée

En complément à 2, on peut donc représenter, sur n bits, les nombres signés compris dans l'intervalle $[-2^{n-1}, 2^{n-1} - 1]$.

Exemple de représentation sur 4 bits ($n = 4$) :

Binaire CA2	Décimal	Binaire non signé	Décimal
1000	-8	1000	8
1001	-7	1001	9
1010	-6	1010	10
1011	-5	1011	11
1100	-4	1100	12
1101	-3	1101	13
1110	-2	1110	14
1111	-1	1111	15
0000	0	0000	0
0001	1	0001	1
0010	2	0010	2
0011	3	0011	3
0100	4	0100	4
0101	5	0101	5
0110	6	0110	6
0111	7	0111	7

D'un point de vue mathématique, un nombre N représenté en complément à 2 sur n bits par $(a_{n-1}, a_{n-2}, \ldots, a_0)$ vaut :

$$N = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

1.4.3 *Règles d'extension de signe*

Les règles d'extension permettent de passer d'un nombre représenté sur n bits au même nombre représenté sur $n + m$ bits.

Entier non signé Dans le cas non signé (entiers naturels), l'extension consiste à rajouter m bits à 0 en tête (poids fort).

Exemple : $9_{(10)}$ se représente sur 4 bits non signés sous la forme $1001_{(2)}$. Sur 6 bits non signés, il se représente sous la forme $001001_{(2)}$.

Complément à 2 Dans le cas d'un nombre représenté en complément à 2, l'extension est un peu plus complexe car il ne faut pas oublier que le bit de poids fort porte l'information du signe. Pour un nombre

en complément à deux, l'extension se fait en dupliquant le bit de poids fort.

Exemples : $-7_{(10)}$ se représente sur 4 bits en complément à deux sous la forme $1001_{(2)}$. Sur 6 bits non signés, il se représente sous la forme $111001_{(2)}$. $5_{(10)}$ se représente sur 4 bits en complément à deux sous la forme $0101_{(2)}$. Sur 6 bits non signés, il se représente sous la forme $000101_{(2)}$.

Preuve : Soit $N = (a_{n-1}, a_{n-2}, \ldots, a_0)$ un entier relatif représenté en complément à 2 sur n bits.

$$\begin{aligned} N &= -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \\ &= -a_{n-1}2^{n-1} \cdot (2-1) + \sum_{i=0}^{n-2} a_i 2^i \\ &= -a_{n-1}2^{n} + a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i \\ N &= -a_{n-1}2^{n} + \sum_{i=0}^{n-1} a_i 2^i \end{aligned}$$

D'où $N = (a_{n-1}, a_{n-1}, a_{n-2}, \ldots, a_0)$ représenté sur $n+1$ bits.

1.4.4 *Virgule fixe*

Un nombre décimal D peut être approximé en base 2 par un vecteur $(a_{n-1}, a_{n-2}, \ldots, a_1, a_0, a_{-1} \ldots a_{-m})$ tel que :

$$D = a_{n-1} \cdot 2^{n-1} + \ldots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + a_{-1} \cdot 2^{-1} + \ldots a_{-m} \cdot 2^{-m}$$

Où :

— $(a_{n-1}, \ldots a_0)$ est la partie entière de D (sur n bits)

— $(a_{-1}, \ldots a_{-m})$ est la partie fractionnaire de D (sur m bits)

— 2^{-m} représente la précision de cette approximation

Cette représentation est appelée représentation en virgule fixe[14]. Elle présente l'avantage de conserver les mêmes opérateurs d'addition et de soustraction que ceux des nombres entiers.

14. Fixe car une fois fixé le nombre de bits de la partie entière et de la partie fractionnaire, ils ne changent pas. Il existe d'autres méthodes, notamment la représentation en virgule flottante dont vous avez probablement déjà entendu parler qui est nettement plus complexe à mettre en œuvre au niveau des opérateurs arithmétiques et donc que nous n'aborderons pas dans ce cours.

1.4.5 *Conclusion*

En conclusion, lorsque l'on représente un nombre en binaire, il est très important de préciser la convention choisie (non signée, complément à 2, virgule fixe…) ainsi que le nombre de bits sur lequel est représenté ce nombre.

1.5 *Opérateurs arithmétiques*

Dans cette section, nous allons présenter comment réaliser une addition et une soustraction, sur des nombres binaires (non signés ou en complément à 2), en utilisant les fonctions logiques de base introduites dans la section 1.2.

1.5.1 *Addition*

Introduction Il est possible d'additionner deux nombres binaires en utilisant l'algorithme élémentaire [15] consistant à additionner chaque chiffre des deux opérandes depuis les chiffres de poids faible vers les chiffres de poids fort et en propageant la retenue.

15. Algorithme que vous avez appris à l'école primaire

Exemple : Addition non signée de $3_{(10)} = 11_{(2)}$ et $2_{(10)} = 10_{(2)}$

$$\begin{array}{cccc} & 1 & & \\ & 0 & 1 & 1 \\ + & 0 & 1 & 0 \\ \hline = & 1 & 0 & 1 \end{array}$$

Additionneur élémentaire On a donc plusieurs additions élémentaires prenant en entrées un bit de chacun des deux opérandes du calcul (a_i et b_i) et une retenue dite entrante (r_i) venant de l'addition élémentaire précédente, et produisant en sortie un bit du résultat (s_i) et une retenue dite sortante (r_{i+1}) destinée à l'addition élémentaire suivante.

Cette addition élémentaire peut s'écrire arithmétiquement sous l'équation [16] :

16. Le $+$ et le $\cdot$ représentent ici les opérateurs arithmétiques d'addition et de multiplication.

$$a_i + b_i + r_i = 2 \cdot r_{i+1} + s_i$$

Les deux sorties r_{i+1} et s_i étant des booléens, il est possible d'exprimer leur valeur sous forme de fonctions booléennes des entrées a_i, b_i et r_i. La table de vérité de cet opérateur d'addition élémentaire est la suivante :

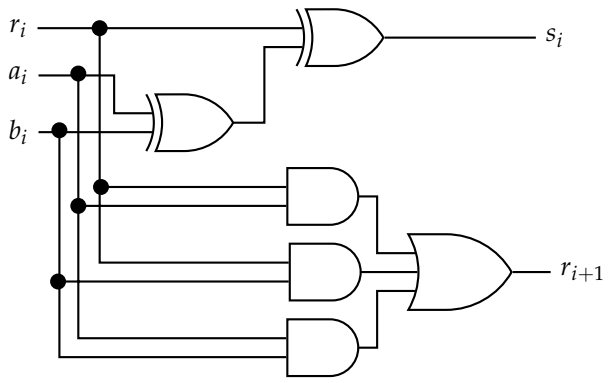
a_i	b_i	r_i	r_{i+1}	s_i	Décimal
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

D'un point de vue analytique on obtient les équations [17] suivantes :

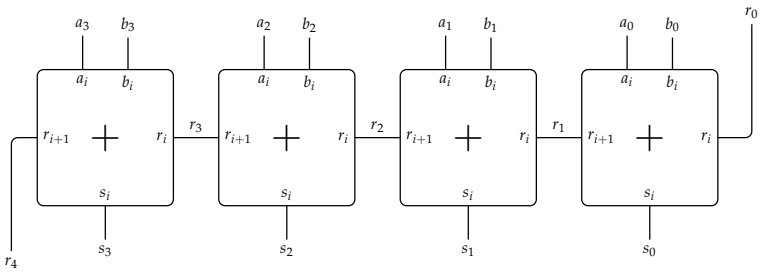
17. Le $+$ et le $\cdot$ représentent ici les opérateurs logiques OU et ET.

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus r_i \\ r_{i+1} &= a_i \cdot b_i + a_i \cdot r_i + b_i \cdot r_i \end{aligned}$$

L'additionneur sur un bit peut être représenté par le schéma suivant :



Additionneur complet Pour réaliser l'additionneur complet, il suffit de connecter ensemble des opérateurs élémentaires sur 1 bits [18] :



18. Cette structure est appelée additionneur à propagation de retenue. Il existe d'autres structures d'additionneur offrant des compromis entre le nombre d'opérateurs logiques utilisés et le temps de calcul différents.

Dynamique de l'addition Il faut faire attention au fait que l'addition de deux nombres binaires non signés (respectivement en complément à deux) sur n bits produit un résultat qui peut être représenté en non signé (respectivement en complément à deux) sur $n + 1$ bits.

Afin d'être certain du résultat de l'addition d'un nombre non signé représenté sur n bits et d'un nombre non signé représenté sur m bits est d'étendre au préalable ces deux opérandes sur $max(n, m) + 1$ bits (en utilisant la règle d'extension vue précédemment dans la section 1.4.3). Le résultat, non signé, sera représenté sur $max(n, m) + 1$ bits. L'éventuelle retenue sortante de l'addition n'est pas à considérer.

De même, afin d'être certain du résultat de l'addition d'un nombre représenté sur n bits en complément à 2 et d'un nombre représenté sur m bits en complément à 2 est d'étendre au préalable ces deux opérandes sur $max(n, m) + 1$ bits (en utilisant la règle d'extension de signe vue précédemment dans la section 1.4.3). Le résultat sera représenté sur $max(n, m) + 1$ bits en complément à 2. L'éventuelle retenue sortante de l'addition n'est pas à considérer.

En SystemVerilog

```
logic [7:0] A;
logic [7:0] B;
logic [7:0] C;
logic retenue;

always @(*)
begin
   {retenue, C} <= A + B;
   // Si on n'a pas besoin de la retenue sortante, on peut
   // écrire : C <= A + B;
end
```

L'addition de deux vecteurs de 8 bits produit en SystemVerilog un résultat sur 9 bits. Si ce résultat est affecté à un vecteur de 8 bits, seuls les 8 bits de poids faible sont gardés (addition modulo 2^8). L'opérateur {} qui permet de concaténer plusieurs signaux permet de récupérer, comme réalisé dans le code précédent, la retenue et le résultat du calcul.

1.5.2 *Soustracteur*

On peut également réaliser une soustraction de deux nombres à partir de soustracteurs élémentaires sur un bit.

Soustracteur élémentaire D'un point de vue arithmétique, on a [19] :

$$a_i - b_i - r_i = -2 \cdot r_{i+1} + s_i$$

19. Le $+$ et le $\cdot$ représentent ici les opérateurs arithmétiques d'addition et de multiplication.

La table de vérité correspondante est :

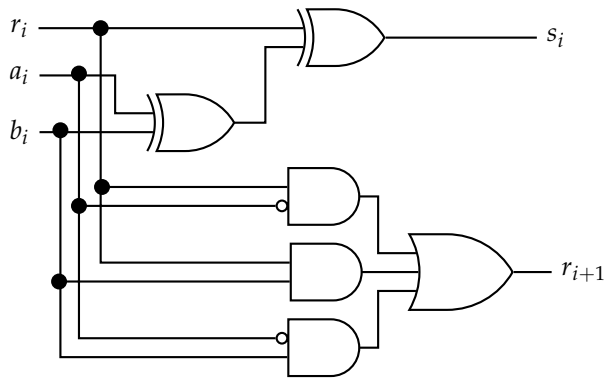
a_i	b_i	r_i	r_{i+1}	s_i	Décimal
0	0	0	0	0	0
0	0	1	1	1	-1
0	1	0	1	1	-1
0	1	1	1	0	-2
1	0	0	0	1	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	1	1	-1

D'un point de vue logique, si on exprime les sorties (r_{i+1} et s_i) en fonction des entrées et des fonctions logiques élémentaires, on obtient [20] :

$$s_i = a_i \oplus b_i \oplus r_i$$
$$r_{i+1} = \overline{a_i} \cdot b_i + \overline{a_i} \cdot r_i + b_i \cdot r_i$$

20. Le $+$ et le $\cdot$ représentent ici les opérateurs logiques OU et ET.

Ce soustracteur sur un bit peut être représenté par le schéma suivant :



1.6 Du temps de propagation au temps de calcul

À cause des contraintes physiques liées à leur réalisation concrète [21], les portes logiques ne peuvent pas changer d'état instantanément lorsque leurs entrées changent.

21. Un aperçu de la réalisation physique de ces portes en logique CMOS est donné dans le chapitre 5.

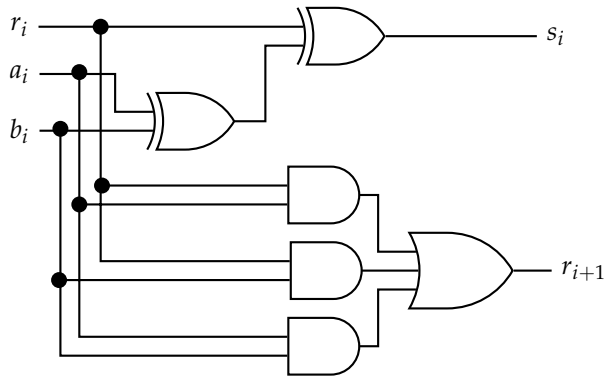
Le *temps de propagation* d'une porte, noté t_p, est le temps entre le changement de la valeur d'une entrée et la stabilisation de la valeur de la sortie de la porte.

Pendant ce laps de temps, la valeur de la sortie de la porte peut ne pas correspondre à la fonction logique réalisée avec les valeurs actuelles des entrées de la porte. Cette valeur de sortie ne doit donc pas être prise en compte durant cette période.

La prise en compte de ce temps de propagation va être importante pour déterminer la vitesse maximale de fonctionnement de l'implémentation d'une fonction logique et plus généralement le temps de calcul d'une fonction arithmétique quelconque.

1.6.1 Exemple de l'additionneur 1 bit

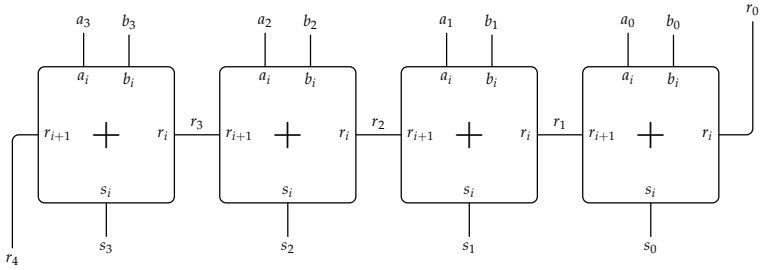
Soit l'additionneur complet sur 1 bit (tel qu'introduit en section 1.5.1) réalisé de la façon suivante :



Si on suppose que chacune des portes élémentaires (ici les XOR à deux entrées, les ET à deux entrées et le OU à trois entrées) ont toutes un temps de propagation $t_p = 1$ ns, une fois les entrées a_i, b_i et r_i stables, les sorties seront stables et correctes au bout de 2 ns. Le temps de propagation de cette implémentation de l'additionneur 1 bit est donc de 2 ns.

1.6.2 *Exemple de l'additionneur à propagation de retenue*

À partir de cet l'additionneur 1 bit, nous construisons un additionneur à propagation de retenue :



Une fois les entrées $a_0, \ldots, a_3$, $b_0, \ldots, b_3$ et r_0 stables, les sorties s_0 et r_1 du premier additionneur 1 bit seront stables au bout de 2 ns. Comme l'entrée r_1 du deuxième additionneur 1 bit n'est stable qu'au bout de 2 ns, ses sorties s_1 et r_2 ne seront stables que 2 ns plus tard, c'est-à-dire au bout de 4 ns.

Donc, pour cet additionneur à propagation de retenue sur 4 bits, les sorties s_3 et r_4 ne seront stables qu'au bout de 8 ns. Le temps de propagation, ou temps de calcul, de cet additionneur est donc est de 8 ns.

2
La logique séquentielle

Dans ce chapitre nous allons introduire la logique séquentielle synchrone en introduisant la notion de mémorisation (cf. section 2.1). Nous présentons ensuite la bascule D, élément mémorisant principal utilisé pour la mise en œuvre de la logique synchrone (cf. section 2.3) ainsi que les contraintes temporelles permettant de garantir le fonctionnement (cf. section 2.3.3). Enfin, nous présenterons quelques applications de la logique synchrone (cf. section 2.5).

2.1 *Mémorisation et logique séquentielle*

Les opérateurs logiques et les opérateurs de calcul combinatoires présentés au chapitre 1 ont la propriété suivante :

— Pour une même valeur des entrées présentées on obtient toujours la même valeur en sortie. En d'autre termes, les opérateurs combinatoires n'ont pas de mémoire.

De plus ces opérateurs possèdent un temps de propagation qu'il faut respecter pour être sûr que le résultat en sortie soit valide.

Dans ce cas, comment utiliser ces opérateurs pour enchainer plusieurs calculs consécutifs de façon fiable ?

2.2 *Logique séquentielle synchrone*

Prenons comme exemple la fonction combinatoire F de la figure2.1.

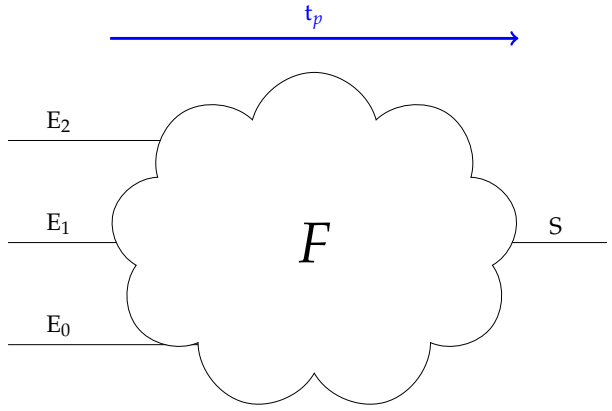


FIGURE 2.1: Un bloc combinatoire

Ce bloc a trois entrées E_0, E_1 et E_2 et une sortie S. Si nous modifions l'une des entrées, il faut attendre t_p pour que le résultat soit valide.

Nous voulons enchaîner plusieurs calculs et obtenir la séquence suivante :

1. $S(0) = F(E_0(0), E_1(0), E_2(0))$
2. $S(1) = F(E_0(1), E_1(1), E_2(1))$
3. $S(2) = F(E_0(2), E_1(2), E_2(2))$
4. ...

Il faut s'assurer que les entrées ne sont pas modifiées tant que la sortie n'est pas valide. Les entrées venant du monde extérieur ou d'un autre bloc de calcul, nous n'avons pas la garantie qu'elles restent stables.

Pour cela nous devons ajouter des éléments pour capturer les valeurs des entrées et les empêcher de changer durant le calcul, comme illustré sur la figure 2.2.

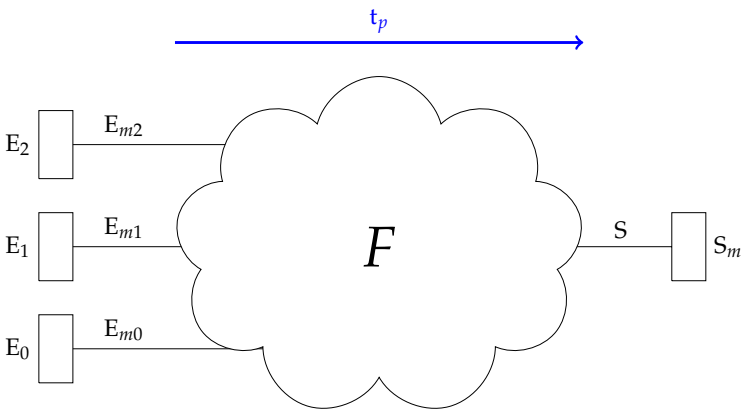


FIGURE 2.2: On capture les entrées pour les empêcher de changer durant un calcul

Pour se simplifier la tâche, la capture et la mémorisation des entrées se fera en même temps, de façon synchrone.

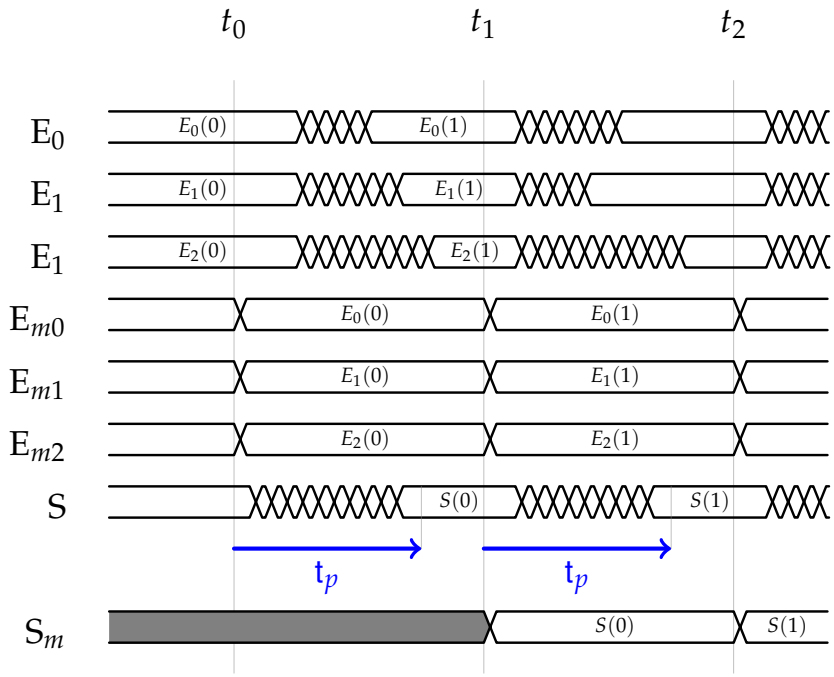


FIGURE 2.3: Chronogramme d'un calcul séquentiel synchrone

Une fois que nous sommes sûrs que le résultat est valide (après un temps t_p), nous pouvons capturer le résultat en sortie et en même temps présenter de nouvelles valeurs sur les entrées.

La sortie, ainsi capturée, peut à son tour être utilisée comme entrée d'un autre bloc de calcul.

Tout ceci est résumé dans le chronogramme 2.3. Nous avons la séquence suivante :

— à l'instant t_0 les entrées sont capturées (on dit aussi échantillonnées)

— à $t_0 + t_p$ la sortie du bloc combinatoire est valide

— à $t_1 > t_0 + t_p$ la sortie est échantillonnée et de nouvelles entrées capturées

— et ainsi de suite ...

Cette logique est dite séquentielle synchrone. Nous avons la garantie que les calculs effectués sont corrects tant que l'intervalle entre les instants d'échantillonnage est supérieur au temps de propagation du bloc combinatoire.

Dans la suite, nous verrons quel composant est utilisé pour échantillonner et mémoriser les signaux et comment l'ensemble est synchronisé.

2.3 *La bascule D*

Le composant de base de la logique séquentielle synchrone est la *bascule D*. Elle peut aussi être appelée *dff* ou encore *flipflop* et parfois *registre*.

La figure 2.4 montre le schéma d'une bascule D. Une bascule D possède deux entrées et une sortie :

— Une entrée particulière, l'horloge **clk** symbolisée par un triangle.

— Une entrée pour la donnée, **D**.

— Une sortie pour la donnée mémorisée, **Q**.

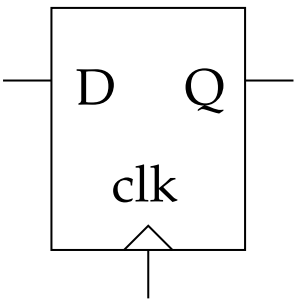


FIGURE 2.4: La bascule D

L'horloge **clk** sert à synchroniser toutes les bascules d'un circuit. L'échantillonnage des signaux se faisant à chacun de ses fronts montants.

Le fonctionnement de la bascule D est le suivant :

— A chaque front montant de l'horloge **clk** (passage de $0 \rightarrow 1$) l'entrée **D** est copiée sur la sortie **Q**. On dit de la donnée est échantillonnée.

— Entre deux fronts d'horloge, la sortie **Q** ne change pas, elle est mémorisée.

Ce comportement peut, comme pour la logique combinatoire, être représenté par une table de vérité (voir la table 2.1).

Description SystemVerilog

Le bloc de code 2.1 est la description SystemVerilog d'une bascule avec une entrée `D` et une sortie `Q`. À chaque front montant (`posedge`) de l'horloge `clk`, l'entrée est copiée sur la sortie. Sinon, la sortie `Q` ne change pas d'état.

D	clk	Q	→ effet
0	↑	0	→ (*copie de D sur Q*)
1	↑	1	→ (*copie de D sur Q*)
×	0	Q	→ (*Q conserve sa valeur*)
×	1	Q	→ (*Q conserve sa valeur*)
×	↓	Q	→ (*Q conserve sa valeur*)

TABLE 2.1: Table de vérité d'une bascule D

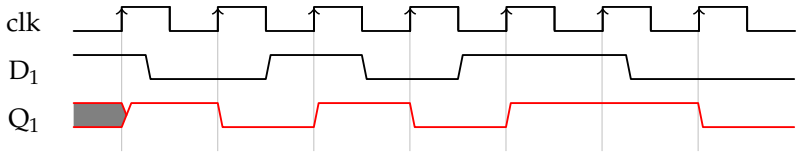
```
module dff (
            input clk,
            input D,
            output logic Q
            );
always @(posedge clk)
    Q <= D;
endmodule
```

CODE 2.1: Description SystemVerilog d'une bascule D

2.3.1 *Utilisation des bascules D*

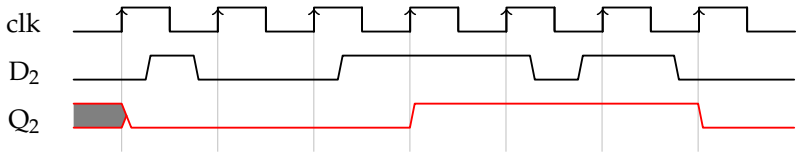
1. Échantillonnage :

Une bascule D sert à échantillonner les données en entrée. Comme nous le voyons sur le chronogramme suivant, la valeur de l'entrée D est capturée à chaque front montant de l'horloge. Cette valeur est conservée jusqu'au front suivant.



2. Filtrage :

Par son fonctionnement, une bascule D filtres les changements de son entrée qui se produisent entre les fronts de l'horloge. Comme le montre le chronogramme suivant, les changements sur le signal d'entrée de durée inférieure à la période de l'horloge n'apparaissent pas sur la sortie.



2.3.2 *Les registres*

Comme nous sommes souvent amenés à manipuler des mots de plusieurs bits, on utilise pour cela plusieurs bascules. Cet assemblage de bascules D est appelé *registre*.

Le symbole d'un registre est le même que celui d'une bascule D. On peut préciser le nombre de bits manipulés sur l'entrée et la sortie du registre (voir la figure 2.5).

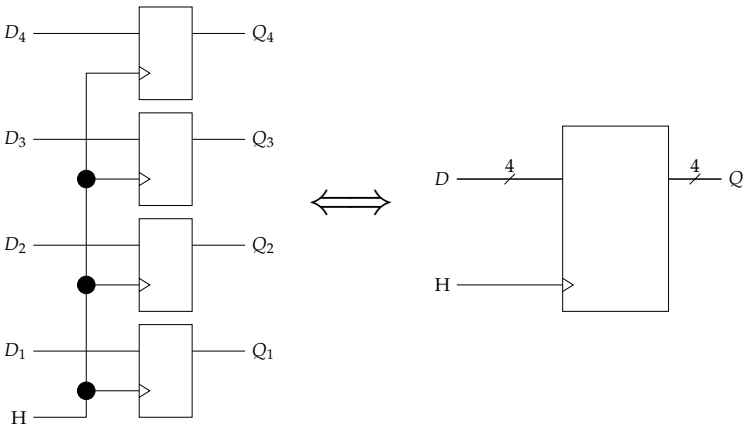


FIGURE 2.5: Un exemple de registre de 4bits de large

Description SystemVerilog

```
module dff (
            input clk,
            input[3:0] D,
            output logic[3:0] Q
            );
always @(posedge clk)
    Q <= D;
endmodule
```

CODE 2.2: Description SystemVerilog d'un registre de 4bits

2.3.3 *Les contraintes et les performances temporelles*

Pour qu'une bascule D fonctionne correctement, certaines précautions sont à prendre. En effet, les données présentées en entrée doivent être stables au moment du front de l'horloge. Ceci se traduit par des *contraintes temporelles* sur les signaux arrivant aux bascules.

La figure 2.6 schématise ces contraintes. Pour une bascule D, les trois temps suivants, sont définis :

t_{su} : temps de pré–positionnement (*setup*)

t_h : temps de maintien (*hold*)

t_{co} : temps de propagation (*clock to output*).

Pour que la bascule D fonctionne correctement, il faut que la donnée présentée en entrée soit stable au front d'horloge, il faut aussi que la donnée reste stable le temps que la mémorisation se fasse :

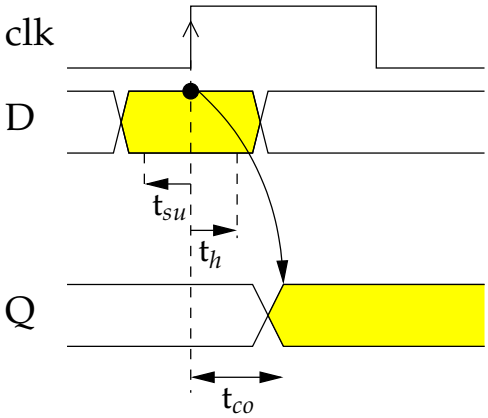


FIGURE 2.6: Contraintes temporelles d'une bascule D

— elle doit avoir atteint sa valeur t_{su} avant le front d'horloge,

— cette valeur doit être maintenue t_h après le front d'horloge.

La copie de l'entrée sur la sortie se fait avec un retard de t_{co}.

Performances d'un opérateur synchrone

Comment calculer la fréquence maximale de fonctionnement d'un bloc de logique synchrone ?

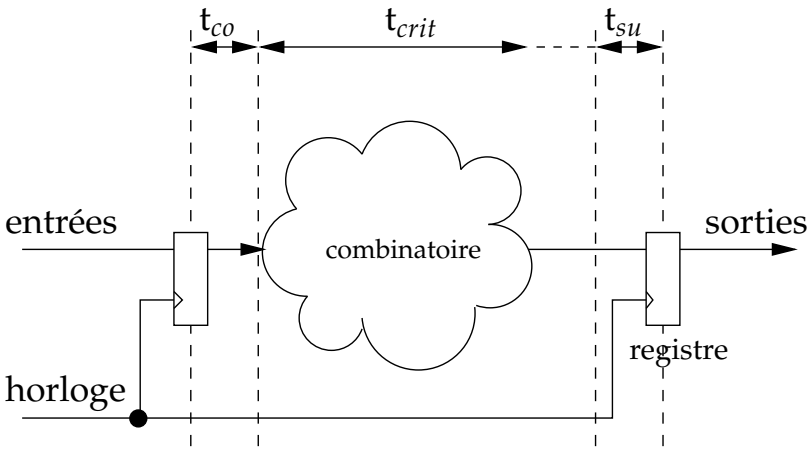


FIGURE 2.7: Période minimale de l'horloge dans un bloc de logique séquentielle

Un bloc de logique synchrone, peut être vu comme un ensemble de blocs combinatoires précédés et suivis de registres. Les entrées de ces blocs combinatoires sortent des registres après un front d'horloge et les sorties sont échantillonnées au front suivant.

La période du signal d'horloge doit être suffisamment grande pour permettre aux sorties de tous les blocs combinatoires de se stabiliser avant le front où elles sont échantillonnées.

On définit ce qu'on appelle le chemin critique, comme le chemin qui a le temps de parcours le plus long dans les blocs combinatoires. On notera t_{crit} ce temps de parcours.

Résumons :

— Au front d'horloge les données en entrée sont échantillonnées.

— Au bout de t_{co} elles arrivent en entrée de blocs combinatoires.

— t_{crit} plus tard on arrive aux bascules suivantes.

— Pour respecter les contraintes temporelles sur l'échantillonnage des bascules, il faut attendre au moins t_{su} avant d'échantillonner.

La figure 2.7 montre ce parcours.
Il faut donc que la période de l'horloge, T_{clk} vérifie :

$$T_{clk} > t_{co} + t_{crit} + t_{su}$$

Ou, exprimé en termes de fréquence de fonctionnement :

$$F_{clk} < F_{max} = \frac{1}{t_{co} + t_{crit} + t_{su}}$$

2.3.4 *L'initialisation*

L'état initial d'une bascule D, au moment où le circuit est mis sous tension, n'est pas connu. Il dépend de plusieurs facteurs, parmi lesquels :

— la technologie utilisée pour la fabrication du circuit,

— l'architecture interne de la bascule,

— des micro-variations technologiques entre les éléments d'un même circuit,

— le bruit ambiant ...

Pour avoir un comportement prédictible, il faut pouvoir initialiser les bascules dans un état connu. Pour cela, un signal supplémentaire, que l'on peut contrôler, doit être utilisé. Ce signal particulier doit permettre de forcer l'état initial d'une bascule.

Si l'état initial est `0` on parle de *reset*. Si l'état initial est `1` on parle alors de *preset*.

Remise à zéro asynchrone :

Pour forcer l'état initial des bascules après la mise sous tension du circuit, on dispose généralement d'un signal supplémentaire de remise à zéro asynchrone. L'action de ce signal est globale, agissant sur toutes les bascules en même temps, indépendamment de l'horloge.

Les bascules possédant ce signal reset asynchrone sont représentées avec une entrée supplémentaire, comme on peut voir sur la figure 2.8.

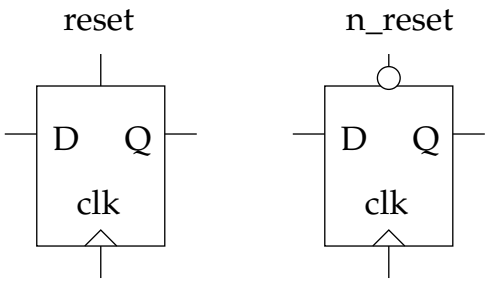


FIGURE 2.8: Shéma de bascules D avec reset asynchrone.

Ce reset asynchrone peut être :

positif : initialise la bascule dès qu'il passe à `1`

négatif : initialise la bascule dès qu'il passe à `0`

Dans le schéma de la figure 2.8, le reset *négatif* est symbolisé par le rond, symbolisant l'inversion, sur l'entrée n_reset.

L'effet d'un reset asynchrone est immédiat et la bascule est maintenue dans cet état initial tant que le signal reset est maintenu dans son état actif.

Le chronogramme 2.9 montre l'effet d'un reset asynchrone négatif. La sortie de la bascule est forcée à zéro dès que l'entrée n_reset passe à `0`. La bascule reste dans cet état jusqu'au front de l'horloge suivant le passage de l'entrée n_reset à `1`.

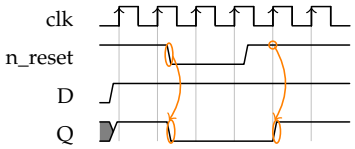


FIGURE 2.9: Chronogramme d'une remise à zéro asynchrone.

Description SystemVerilog

```
module dff (
             input clk,
             input n_reset,
             input D,
             output logic Q
             );
always @(posedge clk or negedge n_reset)
if(!n_reset)
    Q <= 1'b0;
else
    Q <= D;
endmodule
```

CODE 2.3: Bascule D avec remise à zéro asynchrone

Remarquez que dans l'exemple de code 2.3 deux évènements, (`posedge clk`) et (`negedge n_reset`), déclenchent l'évaluation du processus `always`. Ceci veut dire que le passage à zéro de l'entrée `n_reset` est pris en compte immédiatement.

Remise à zéro synchrone :

Une remise à zéro synchrone ou reset synchrone, est un forçage de l'état de la bascule qui prend effet au front d'horloge.

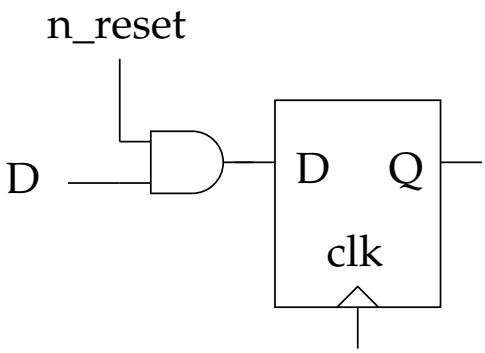


FIGURE 2.10: Construction d'une bascule D avec reset synchrone.

La figure 2.10 montre comment on peut construire une bascule avec un reset synchrone.

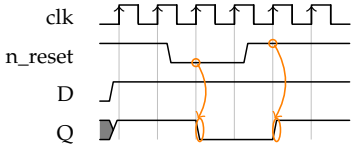


FIGURE 2.11: Chronogramme d'une remise à zéro synchrone.

Le chronogramme 2.11 montre comment cette remise à zéro synchrone se produit.

Le signal **n_reset** est un de polarité négative, il agit sur la bascule quand sa valeur est `0`. Au front de l'horloge, la bascule est forcée à zéro. Quand le signal **n_reset** repasse à `1`, la bascule retrouve son fonctionnement normal et la donnée en entrée est échantillonnée au front d'horloge.

Un reset synchrone doit respecter les mêmes règles que tout signal échantillonné par une bascule D.

En général, le reset synchrone est utilisé pour l'initialisation fonctionnelle d'une partie du circuit. Il est ainsi généré par une autre partie du circuit qui est elle aussi synchrone ce qui garanti qu'il a une durée d'au moins une période d'horloge.

Description SystemVerilog

CODE 2.4: Bascule D avec remise à zéro synchrone

```
module dff (
             input clk,
             input n_reset,
             input D,
             output logic Q
             );
always @(posedge clk)
if(!n_reset)
    Q <= 1'b0;
else
    Q <= D;
endmodule
```

Remarquez que dans l'exemple de code 2.4 seul l'évènement (`posedge clk`) déclenche l'évaluation du processus `always`. Ceci veut dire que l'état de l'entrée `n_reset` n'est testé qu'au front de l'horloge.

2.4 *Généralisation*

Un bloc générique de logique séquentielle est représenté en figure 2.12.

Dans un bloc de logique séquentielle on utilise la même horloge pour synchroniser l'ensemble des calculs. Tous les signaux venant du monde extérieur doivent être échantillonnés. Les sorties des blocs combinatoires doivent elles aussi être échantillonnées.

Ces sorties peuvent être :

— utilisées à l'extérieur (dans un autre bloc),

— redirigée vers les entrées de la logique combinatoire.

Dans le second cas nous parlons d'état interne. La valeur des sorties dépend alors de la valeur des entrées et de cet état interne.

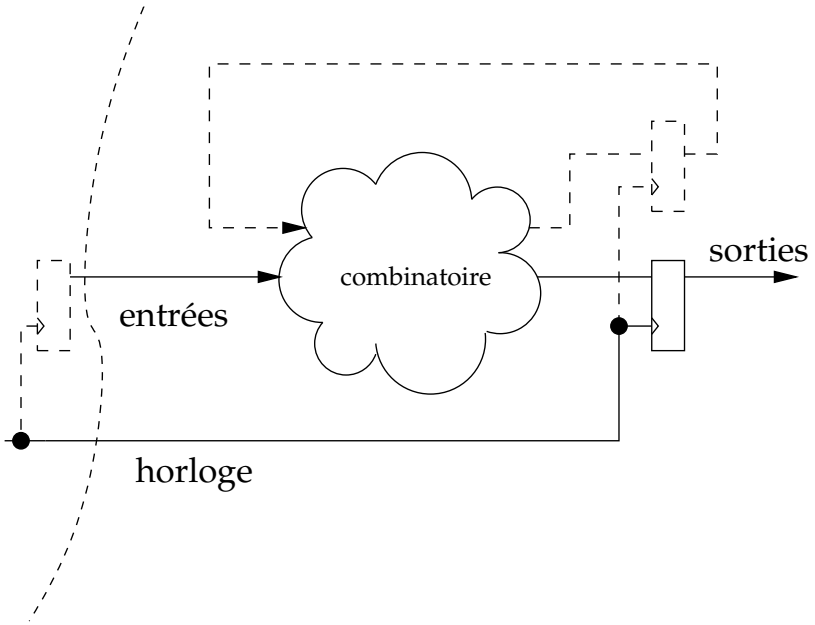


FIGURE 2.12: Schéma générique d'un bloc de logique séquentielle synchrone

Pour que les valeurs consécutives des sorties soient prédictibles, il faut pouvoir forcer l'état initial. Ce qui est fait grâce à un signal externe d'initialisation.

2.5 *Applications de la logique synchrone*

Vous trouverez dans la suite quelques exemples d'applications de la logique synchrone.

2.5.1 *Les registres à décalage*

La figure2.13 montre le schéma d'un registre à décalage composé de quatre bascules D mises en série. On peut aussi parler de registre à décalage d'une profondeur de 4.

Ce registre a comme entrée le signal E et comme sortie le signal S. Les signaux intermédiaires D_1, D_2 et D_1 servent à relier la sortie d'une bascule à l'entrée de la bascule qui la suit.

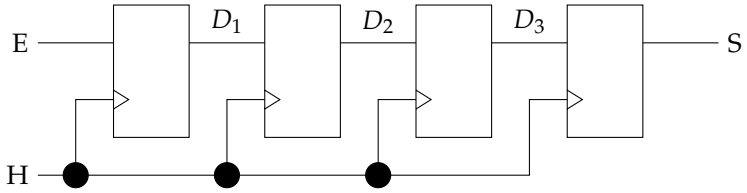
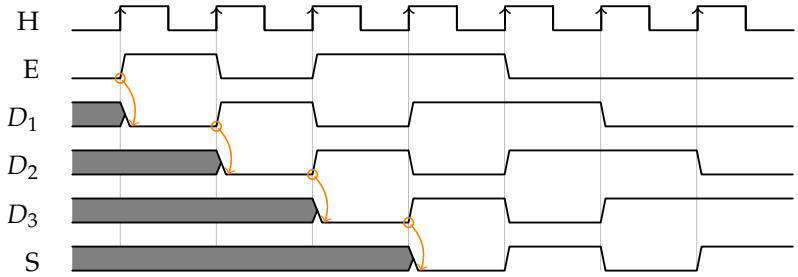


FIGURE 2.13: Shéma d'un registre à décalage composé de 4 bascules

Le chronogramme suivant montre le fonctionnement de ce registre à décalage.



Au premier coup d'horloge, l'entrée E est copiée en D_1. Puis aux coups d'horloge suivants, la valeur de E est décalée de proche en proche.

Utilisation d'un registre à décalage :

Un registre à décalage sert à retarder un signal d'un nombre entier de périodes d'horloge. Dans l'exemple précédent, la sortie S est une copie de l'entrée E avec quatre périodes d'horloge de retard.

Aussi, les sorties des bascules représentent l'histoire du signal E.

$$\begin{aligned} D_1 &= \mathrm{E}(t-1) \\ D_2 &= \mathrm{E}(t-2) \\ D_3 &= \mathrm{E}(t-3) \\ \mathrm{S} &= \mathrm{E}(t-4) \end{aligned}$$

Respect des contraintes temporelles :

Nous avons vu, en section 2.3.3, que pour qu'une bascule fonctionne correctement il faut que le signal en entrée soit stable au moment du front d'horloge.

Pour qu'un registre à décalage respecte cette contrainte, il faut que le temps de propagation de la bascule soit supérieur au temps de maintien.

C'est-à-dire :

$$t_{co} > t_h$$

Le respect de cette contrainte est garanti par construction par les concepteurs des bascules D.

Description SystemVerilog

CODE 2.5: Registre à décalage de profondeur 4

```
module shift (
             input clk,
             input E,
             output logic S
             );

logic D1,D2,D3;

always @(posedge clk)
begin
    D1 <= E;
    D2 <= D1;
    D3 <= D2;
    S  <= D3;
end
endmodule
```

Remarquez dans l'exemple de code 2.5 que les signaux internes sont déclarés à l'intérieur du module.

De plus, dans un processus (`always`) les expressions utilisées à droite des affectations (`<=`) sont évaluées au moment de l'évènement déclenchant (ici le front montant de l'horloge). Ainsi les quatre affectations (`<=`) sont effectuées en même temps à chaque front montant de l'horloge. Leur ordre d'écriture dans le processus n'a donc pas d'importance.

2.5.2 *Les compteurs*

Un compteur est un bloc de logique séquentielle synchrone de base. La sortie d'un compteur est incrémentée à chaque cycle d'horloge.

Pour construire un compteur il faut deux éléments :

— un registre, pour stocker la valeur (l'état) du compteur,

— un incrémenteur (additionneur avec 1) pour calculer les valeurs suivantes.

La figure 2.14 montre le schéma d'un tel compteur utilisant un registre et un addditionneur de 8 bits.

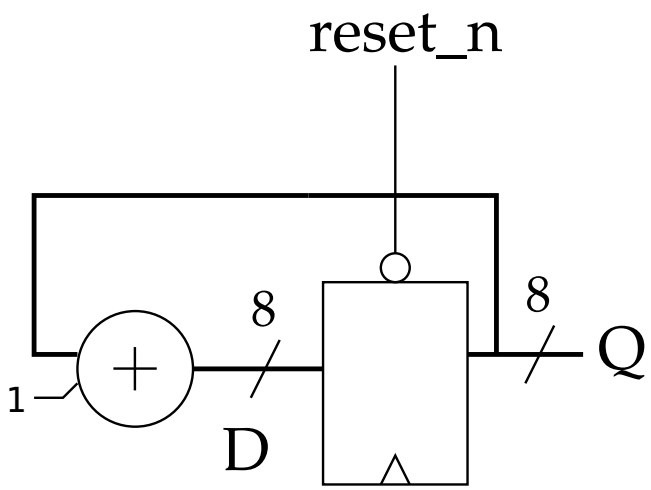
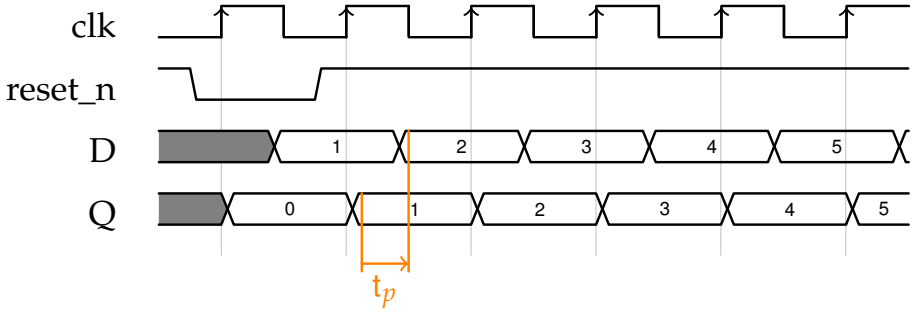


FIGURE 2.14: Schéma d'un compteur modulo 256

Le signal d'initialisation reset_n permet de forcer l'état initial et ainsi de commencer à compter à zéro. En suite, à chaque coup d'horloge, la valeur de Q est incrémentée.

Comme le nombre de bits du registre et de l'additionneur est de 8, une fois arrivé à 255 le compteur repassera naturellement à 0. On parle de compteur modulo.

Le chronogramme suivant montre l'évolution des signaux dans le compteur. Ici l'action du signal de remise à zéro est synchrone.



Remarquez que le signal de remise à zéro agit sur la sortie du registre. Remarquez aussi que la sortie de l'additionneur change après que la sortie du registre et que nous voyons apparaitre son temps de propagation.

Description SystemVerilog

Remarquez que le signal interne `D` n'a pas besoin d'apparaitre dans le code SystemVerilog. Quand nous écrivons `Q <= Q + 1` cela veut dire qu'au front d'horloge, la valeur du signal `Q` sera remplacée par son ancienne valeur plus **1**.

```
module cpt (
             input clk,
             input reset_n,
             output logic[7:0] Q
             );

always @(posedge clk)
 if (!reset_n)
    Q  <= '0;
 else
    Q  <= Q + 1;

endmodule
```

CODE 2.6: Compteur modulo 256 avec remise à zéro synchrone

2.5.3 *Le "pipeline"*

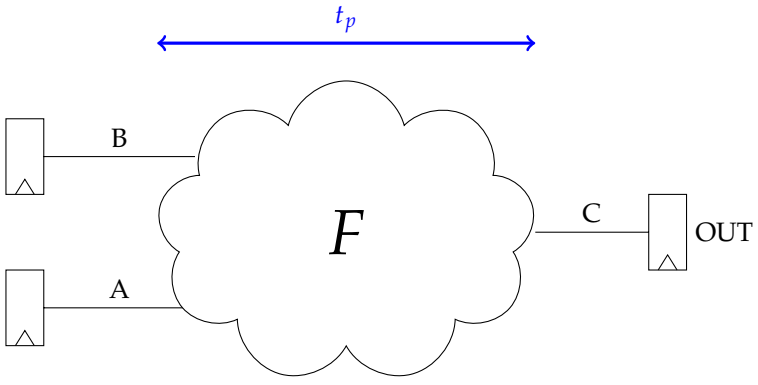
Le *pipeline* est une technique qui permet d'augmenter la fréquence de fonctionnement d'un bloc séquentiel. Elle est utile pour augmenter le débit de calcul sur un flux de données.

Considérons l'exemple suivant :

— Une fonction combinatoire F de temps de propagation t_p

— Les données sont présentées sur les entrées A et B à la cadence de l'horloge clk de période T_{clk}.

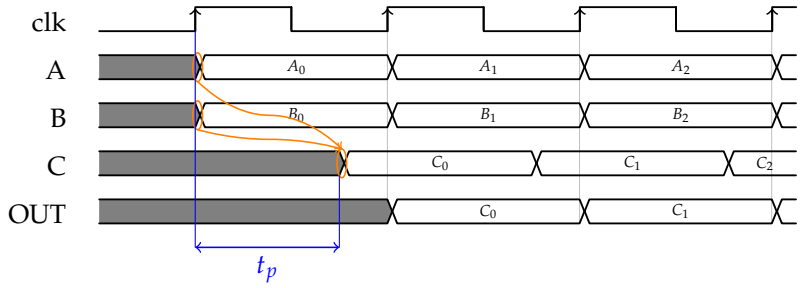
nb. Pour simplifier les expressions, nous négligeons dans cet exemple les temps de propagations dans les registres.



Le système fonctionne correctement tant que la relation suivante, entre la période de l'horloge et le temps de propagation, est respectée :

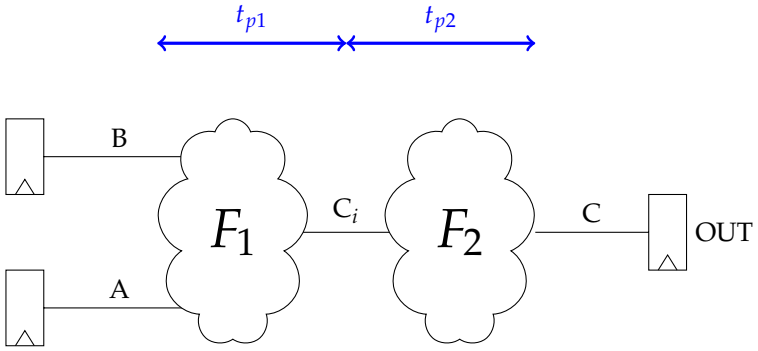
$$t_p < T_{clk}$$

La sortie du bloc combinatoire C est échantillonnée et on obtient un résultat sur la sortie OUT à chaque période T_{clk}. Ce que montre le chronogramme suivant :



On décompose F en deux fonctions combinatoires F_1 et F_2 de temps de propagation respectifs t_{p1} et t_{p2} .

— On s'arrange pour avoir $t_{p1} < t_p$ et $t_{p2} < t_p$

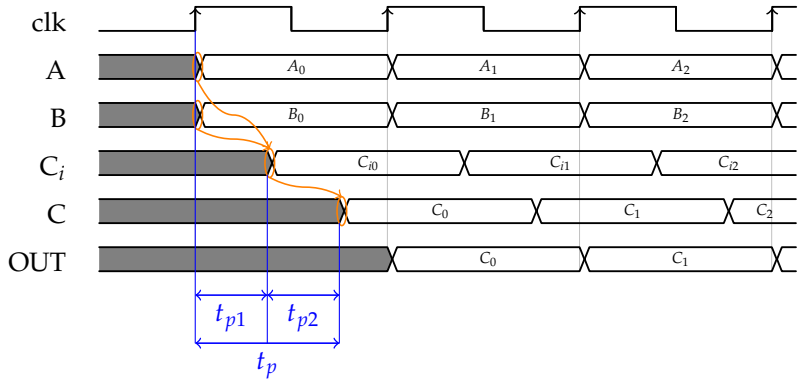


Le point C_i représente l'ensemble des signaux reliant les deux blocs combinatoires F_1 et F_2.

Le système fonctionne correctement tant que

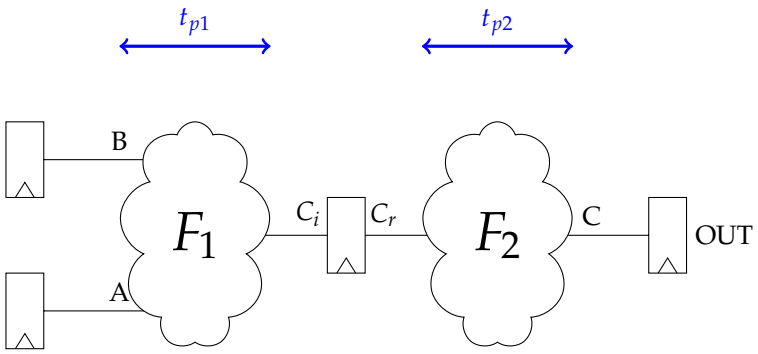
$$t_{p1} + t_{p2} < T_{clk}$$

Dans le chronogramme suivant, nous avons pris $t_{p1} + t_{p2} = t_p$ pour pouvoir le comparer au chronogramme précédent.



Ajoutons un registre entre les blocs combinatoires F_1 et F_2. Les deux blocs sont maintenant précédés et suivis de registres.

Dans le jargon, nous disons que nous avons ajouté une couche de *pipeline*.



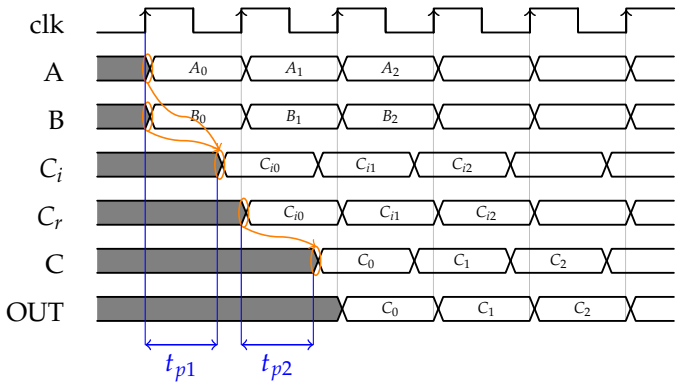
Le système fonctionne correctement si les deux conditions suivantes sont vérifiées :

$$\begin{cases} t_{p1} < T_{clk} \\ t_{p2} < T_{clk} \end{cases}$$

Mais comme les deux temps de propagation t_{p1} et t_{p2} sont inférieurs au temps de propagation initial t_p, nous pouvons réduire la période de l'horloge T_{clk}.

Ou augmenter ainsi la fréquence de fonctionnement et donc la cadence à laquelle les calculs sont faits. Ce qui est illustré dans le chronogramme suivant :

Remarquez que le premier résultat arrive sur la sortie OUT au bout de deux périodes de l'horloge.



Résumons :

— Le *pipeline* permet d'augmenter la fréquence de fonctionnement.

— La latence initiale augmente du nombre de couches de *pipeline*.

— On a augmenté la complexité et la taille du circuit en ajoutant des bascules et en modifiant le bloc combinatoire initial.

3
Les unités de contrôle

Les architectures classiques de traitement numérique sont traditionnellement découpées en ***Unités de Traitement*** *et* ***Unités de Contrôle****.*

Les ***Unités de Traitement*** *sont en charge du traitement proprement dit des données. L'****Unité Arithmétique et Logique*** *d'un microprocesseur est un exemple d'unité de traitement : Elle reçoit des données interprétées comme des entiers, et réalise des opérations arithmétiques simples (addition, soustraction,..) ainsi que des opérations booléennes sur ces données. Pour opérer convenablement, c'est-à-dire exécuter* ***la bonne opération au bon moment****, ces unités de traitement reçoivent des* ***ordres*** *par l'intermédiaire de* ***signaux de contrôle****.*

Le ***Unités de Contrôle*** *sont en charge de générer ces signaux de contrôle, en s'appuyant sur la connaissance du passé (l'****état*** *courant du système) et sur des sollicitations extérieures (les* ***commandes****). En clair, ce sont des* ***automates*** *matériels.*

Ce chapitre décrit donc les structures nécessaires à la réalisation d'automates matériels ainsi que les techniques de codage SystemVerilog associées.

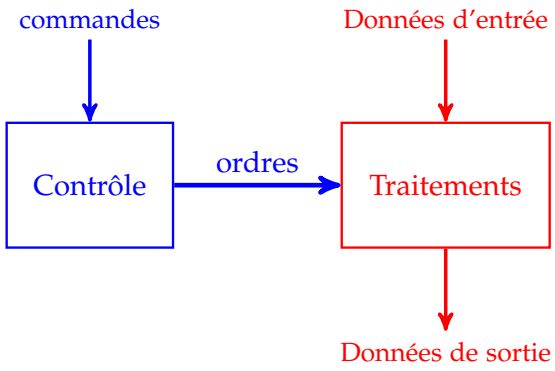


FIGURE 3.1: Système numérique partagé en unité de cotrole et unité de traitement

3.1 Automates matériels synchrones

3.1.1 Gestion de l'évolution des états

Les automates matériels synchrones peuvent être construits de la façon suivante :

— L'état courant de l'automate est codé dans un registre (ensemble de bascules D) synchrone.

— La taille du registre (nombre de bits) définit ainsi le nombre d'états maximum pouvant être codés.

— L'initialisation du registre permet d'imposer un état de départ à l'automate.

— En fonction de l'**état courant** et des entrées (les **commandes**) un **état futur** est calculé combinatoirement.

— Au front d'horloge, le registre est mis à jour : l'état futur devient l'état courant.

Le schéma de la figure 3.2 décrit la structure matérielle correspondante. Le nuage représente le bloc de calcul combinatoire. Remarquons que les signaux d'entrée des bascules codent **l'état futur (`n_state`)**, alors que les signaux de sortie des bascules codent **l'état courant (`state`)**.

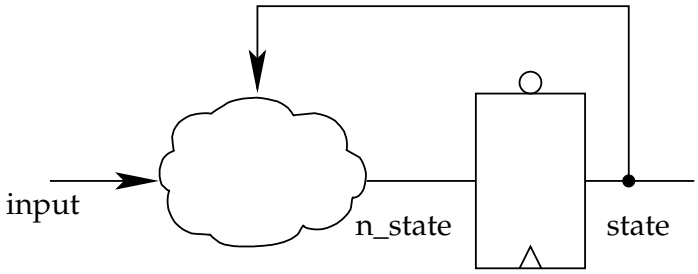


FIGURE 3.2: Structure matérielle pour l'évolution des états

3.1.2 Génération des signaux de contrôle

Un automate, n'a de sens que s'il sert à piloter une unité de traitement. Pour cela, un bloc de calcul combinatoire est en charge de créer les signaux de contrôle, soit simplement à partir de l'état courant (dans la littérature on appelle cela machine de Moore), soit à partir de l'état courant et des entrées du contrôleur (dans la littérature on appelle cela machine de Mealy). Les figures 3.3 et 3.4 représentent respectivement les schémas d'une machine de Moore et d'une machine de Mealy. Suivant la situation, l'une ou l'autre est plus adaptée.

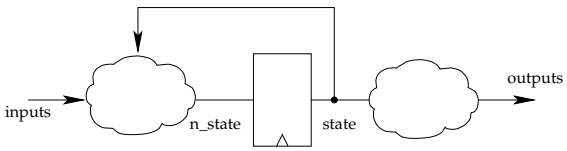


FIGURE 3.3: Structure matérielle pour le calcul des sorties (Moore)

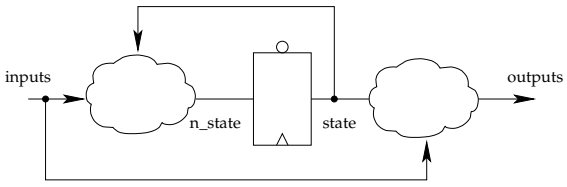


FIGURE 3.4: Structure matérielle pour le calcul des sorties (Mealy)

3.2 *Codage SystemVerilog des automates*

En s'appuyant sur les schémas précédents la traduction en code SystemVerilog est assez directe comme l'indique la figure 3.5 :

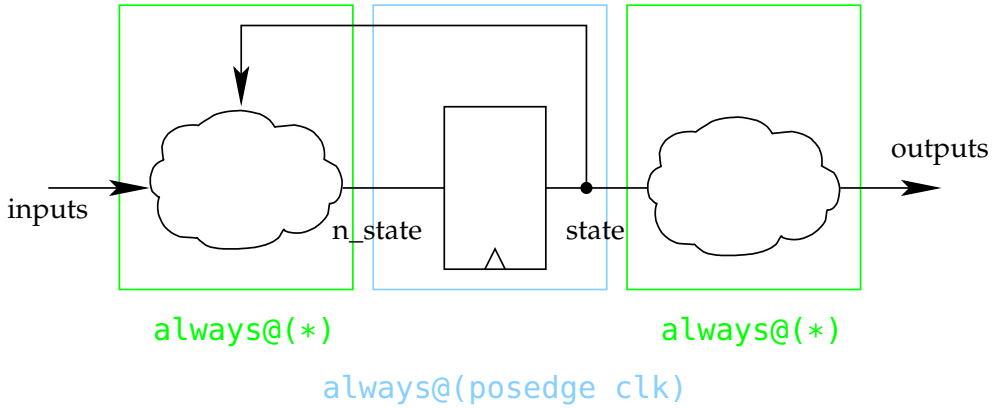


FIGURE 3.5: Processus SystemVerilog pour un automate synchrone

— Un processus combinatoire se charge de calculer les états futurs

— Un processus synchrone se charge de la mise à jour des états

— Un ou plusieurs processus combinatoires génére(nt) les signaux de contrôle.

3.2.1 *Utilisation de types énumérés pour décrire des états*

Il est possible, en SystemVerilog, de définir des vecteurs de booléens sous la forme de type énumérés par l'utilisation du mot-clé **enum**.

— Par défaut, les valeurs énumérées correspondent aux codes **0**, **1**, **2**, **3** ... en partant de la première valeur.

— La taille nécessaire au codage des différentes valeurs est définie dans la définition du vecteur **logic**.

Le code 3.1 décrit la définition de signaux **state** et **n_state** sous la forme de types énumérés. Ces signaux représentent respectivement l'état courant et l'état futur de notre automate. Il est possible ensuite d'utiliser directement les noms symboliques pour calculer des affectations ou faire des comparaisons.

```
...
// state et n_state doivent être déclarés conjointement
enum logic[2:0] {SWAIT, S1, S2, S3, S4, S5, S6} state, n_state ;
...
always@(*) begin
   n_state <= S1;
   if(state == SWAIT) begin
     ...
   end else begin
     ...
   end
```

CODE 3.1: Les états sous forme de type énuméré

3.2.2 *Codage de la mise à jour de l'état*

Le code 3.2 est un exemple de définition du registre d'état. Attention : Il ne faut pas oublier l'initialisation qui permet de définir l'état de départ de l'automate. Dans l'exemple suivant, on choisit de placer l'automate dans l'état **`SWAIT`** si le signal **`reset`** vaut **`1`** au front montant de l'horloge (initialisation synchrone). On aurait pu aussi choisir une initialisation asynchrone.

```
always@( posedge clk)
   if(reset)
      state <= SWAIT;
   else
      state <= n_state;
   end
```

CODE 3.2: Le registre d'états

3.2.3 *Codage de l'évolution d'un automate*

La construction **`case`** permet de traduire facilement la table d'évolution des états. Nous pouvons ainsi regrouper dans un même vecteur (les accolades servent à concaténer des éléments dans un vecteur) l'ensemble des signaux d'entrée et l'état courant de manière à définir la **table d'évolution des états** de manière très régulière. Remarquez, dans le code 3.3, l'usage du mot-clef **`default`** pour définir le cas par défaut, c'est-à-dire le fait de ne pas changer d'état.

```
always@(*)
  case ( {state, entree_1, entree_2} )
    {SWAIT,1'b0, 1'b1} : n_state <= S1;
    {S1   ,1'b1, 1'b1} : n_state <= S2;
    ...
    default            : n_state <= state;
  endcase
```

CODE 3.3: Le calcul de l'état futur

3.2.4 *Extension de la construction **`case`** : le mot-clef **`casez`***

Il peut parfois être long et fastidieux de donner explicitement tous les cas. Lorsque certains signaux ne sont pas utiles dans les équations de transition, il est possible de les représenter par le caractère «**`?`**». Dans ce cas le mot-clé **`case`** doit être remplacé par **`casez`** comme indiqué dans le code 3.4

Dans l'exemple précédent, on passe de l'état **`S1`** à l'état **`S2`** indépendamment de la valeur du signal **`entree_2`**.

```
always@(*)
  casez ( {state, entree_1, entree_2} )
    {WAIT,1'b0, 1'b1} : n_state <= S1;
    {S1,  1'b1, 1'b?} : n_state <= S2;
    ...
    default : n_state <= state;
  endcase
```

CODE 3.4: Exemple d'utilisation du mot-clé `casez`

3.2.5 *Utilisation des constructions `if` et les transitions par défaut*

Dans certains cas, il est plus simple et plus lisible d'exprimer les conditions de transitions en utilisant la construction `if`.

Une astuce de codage permet de réduire la complexité du code en prévoyant un cas par défaut dans lequel on reste dans l'état courant. Comme le montre l'extrait de code 3.6, les `if` ne servant à exprimer que les changements d'état.

Notez l'utilisation obligatoire de `begin...end` pour délimiter les blocs de code.

```
always@(*)
begin
  // cas par défaut, on reste dans l'état courant
  // n_state vaudra cette valeur si aucune autre condition n'est vérifiée
  n_state <= state;
  casez (state)
    WAIT : begin
        if (entree_1) n_state <= S1;
        if (entree_2) n_state <= S2;
        // si aucune condition n'est vérifiée, on sait que n_state
        // vaudra state et qu'on reste dans l'état courant
     end
    S1   : begin
        if (entree_3) n_state <= S3;
        ...
     end
    ...
    // ici le cas default n'est pas obligatoire
  endcase
end
```

CODE 3.5: Utilisation de `case` et `if` pour les transitions d'états

3.2.6 *Codage des signaux de sortie (signaux de contrôle).*

Pour des raisons de facilité de lecture, il est recommandé d'écrire un processus combinatoire par signal de sortie. Cela permet souvent d'éviter des codes trop complexes à base de `case` et de `if` imbriqués, difficiles à lire et à corriger. La règle étant de ne pas mélanger dans un même processus `always` des codes qui n'ont rien de commun.

Le code 3.6 est un exemple de génération de signaux de contrôle. Dans cet exemple, on imagine qu'une unité de traitement est capable de calculer des divisions et démarre un traitement si le signal **start_div** est égal à **1**.

Le contrôleur reçoit la commande **ask_div** mais ne donne l'ordre au diviseur de démarrer que lorsque il est dans l'état **START_COMPUTATION**.

Enfin, le contrôleur génère un signal **end_of_computation** dès qu'il atteint l'état **END_OF_COMPUTATION**.

Notez qu'en SystemVerilog, il y équivalence entre une expression booléenne et un bit.

```
always@(*) start_div <= (state==START_COMPUTATION)) && ask_div;

always@(*) end_of_computation <= (state==END_OF_COMPUTATION));
```

CODE 3.6: Exemple de codage de signaux de contrôle

4
Le nano processeur

Dans ce chapitre nous allons présenter comment concevoir un microprocesseur à partir d'un exemple simple et progressif. Nous introduisons la notion de programme (cf. section 4.1) ainsi que la mémoire servant à le stocker (cf. section 4.2). Dans la suite du chapitre nous faisons évoluer l'architecture du processeur pour permettre d'exécuter des programmes de plus en plus complexes.

4.1 Programme, instructions et données

Les processeurs ne sont rien d'autre que des machines à calculer *programmables*. Imaginez que vous êtes comptable, et que vous avez à effectuer une série d'opérations qu'on vous a gentiment inscrites sur une feuille de papier.

Voici un exemple d'instructions qu'on peut vous avoir donné :

```
1    faire 112 + 3
2    faire   4 + 5
3    faire   2 + 16
4    faire   ...
```

Vous pouvez avoir des enchaînements un peu plus complexes, par exemple :

```
1    faire 112 + 4
2    faire   "résultat précédent" + 1
3    ...
```

pour lesquels vous devez mémoriser des résultats intermédiaires.

Et si on vous demande vraiment de réfléchir, de raisonner et de prendre des décisions, vous vous retrouverez avec :

```
faire 112 + 3
faire "résultat précédent" - 4
faire si "le résultat est nul" alors "passer à l'étape" 6,
      sinon "continuer"
faire 3 * 4
faire "résultat précédent" + 9
faire "ouvrir la fenêtre"
faire "résultat de l'étape" 2 - 15
faire "passer à l'étape" 12
faire ...
```

Un microprocesseur est un dispositif électronique qui peut faire ce travail à votre place. Il ne reste qu'à rendre cette feuille de papier lisible par ce dispositif électronique.

Examinons un peu plus le texte de la feuille de papier pour essayer de trouver les éléments important qu'il faut faire « comprendre » au processeur.

Ce texte est une série [1] d'opérations ou d'actions à effectuer, c'est ce qu'on appelle un « **programme** ».

Dans ce programme on distingue deux types d'éléments :

- **les données :**
  - d'abord les **opérandes** proprement dits (3, 4, 112, ...),
  - et les opérandes implicites (« `résultat précédent` », « `résultat de l'étape 2` », ...);
- **les instructions :**
  - pour nous ce sont principalement les opérations arithmétiques (« + », « − », « ∗ », « / » ...),
  - il y a aussi des tests (« `si le résultat précédent est nul` ... »),
  - et des sauts (« `passer à l'étape 12` »), souvent conditionnés par un test (« `alors passer à l'étape 6` »),
  - ainsi que des instructions spéciales (« `ouvrir la fenêtre` »).

1. Notez que ces opérations sont numérotées : elles ont un **ordre**.
Dans le premier exemple, l'ordre n'a pas tellement d'importance, mais il en a une dans les deux suivants quand on parle de « résultat précédent », l'« étape 6 », ...

Résumons :

- Un programme est une suite ordonnée d'instructions.
- Une instruction permet :
  - d'agir sur des opérandes en utilisant des opérateurs,
  - d'agir sur le flot d'exécution du programme.

4.2 *La mémoire RAM*

Comme le processeur est un dispositif électronique numérique, il ne peut interpréter que des niveaux logiques (1 ou 0). Il faudra donc « **encoder** » les instructions et les données de notre programme en une série de **bits** (des 1 et 0 donc des nombres).

La feuille de papier doit être elle aussi remplacée par dispositif électronique permettant de stocker ces nombres et de les modifier pendant l'exécution du programme. Cette mémoire électronique doit aussi permettre d'accéder aux éléments qu'elle contient dans n'importe quel ordre (de façon aléatoire).

Ce type de mémoire est appelé « **RAM** » (Random Access Memory). La figure 4.1 montre le schéma de l'interface d'une RAM simple pouvant contenir 256 mots de 8 bits.

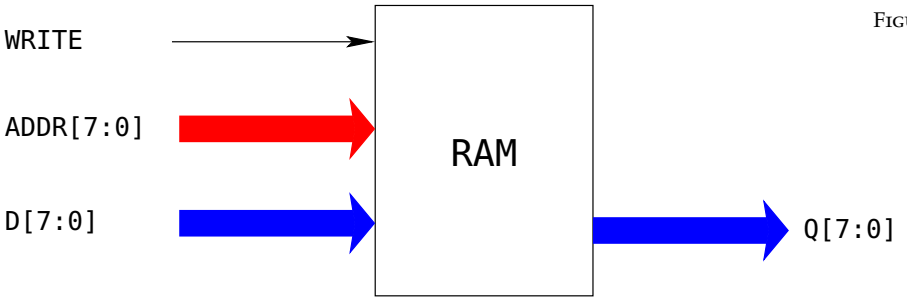


FIGURE 4.1: Une mémoire minimaliste

La RAM possède trois bus :

— un bus d'adresses, `ADDR[7:0]` indiquant l'emplacement en mémoire de la donnée à laquelle on accède,

— un bus de données d'entrée, `D[7:0]`, pour les données qu'on va écrire en RAM,

— un bus de données de sortie, `Q[7:0]`, pour les données qu'on lit en RAM,

ainsi que

— un signal de contrôle sur 1 bit, `WRITE`, indiquant si on est en train de faire une *lecture* dans la RAM (`WRITE==0`), ou une écriture (`WRITE==1`).

La figure 4.2 montre les chronogrammes d'une lecture et d'une écriture.

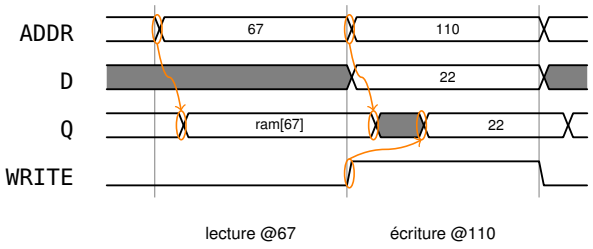


FIGURE 4.2: exemple d'accès à la RAM

Le fonctionnement de la RAM est le suivant :

— la RAM présente en permanence sur sa sortie `Q[]` la donnée stockée à l'adresse présente sur `ADDR[]`

  — *si cette donnée n'est pas utile, on l'ignore,*

— si `WRITE` est actif (égal à 1), la valeur présente sur `D[]` est écrite dans la mémoire à l'adresse présente sur `ADDR[]`,

— *cette écriture prend un certain temps*,

— si `WRITE` est inactif (égal à 0), l'entrée `D[]` est ignorée.

— Pendant que `WRITE` est actif, le bus `Q[]` prend comme valeur le contenu de la case pointée par l'adresse. Cela va donc être la copie de `D[]`, mais avec du retard.

Les données et les instructions de nos programmes sont alors « **encodés** » pour être stockées dans la RAM. L'encodage des données en binaire est quelque chose de naturel. Pour les instructions, il faudra choisir un encodage puis respecter cette convention.

4.3 *Le système de base*

Maintenant que nous avons notre feuille de papier électronique qui peut contenir notre programme, nous allons la connecter à notre processeur. Ceci nous permet de définir l'interface extérieure du processeur, qui ne changera plus, avant de concevoir son fonctionnement interne.

La figure 4.3 montre la structure de notre système. Il contient :

— notre microprocesseur

— la RAM, reliée au processeur,

— un buzzer qui servira à jouer de la musique

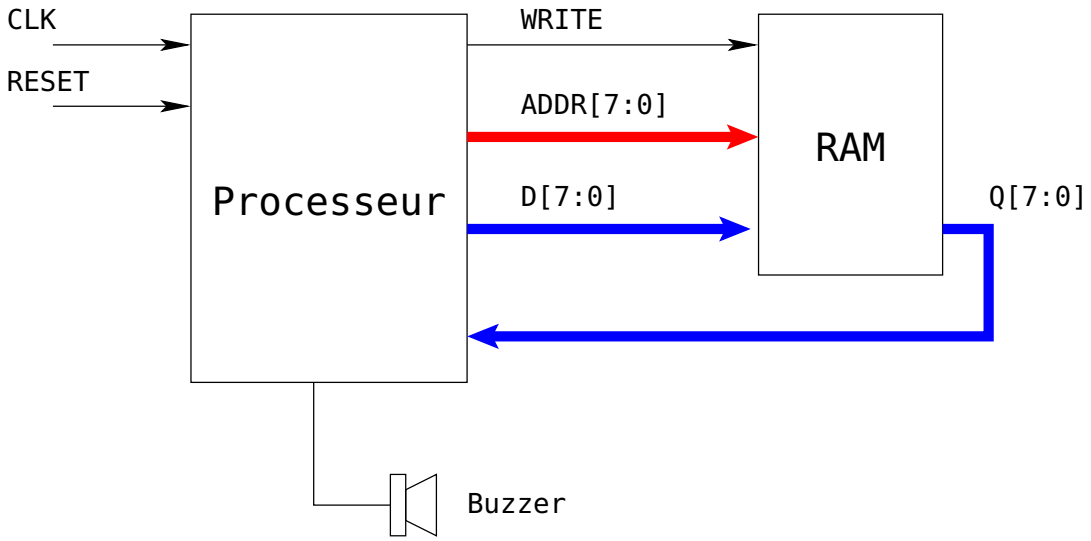


FIGURE 4.3: Architecture du système de complet

Nous connecterons notre microprocesseur à la RAM :

— Un bus pour les données `D[]` de 8 bits.

— Un bus pour les adresses `ADDR[]` de 8 bits [2].

— Un signal `WRITE` pour commander les écritures.

2. La RAM contient 256 mots, il y donc 256 valeurs potentielles d'adresse

4.4 *Première version du microprocesseur : l'automate linéaire*

Dans cette première étape nous allons définir l'architecture du processeur permettant d'exécuter simplement une série d'instructions indépendantes.

Aussi, nous nous limiterons à deux instructions, l'addition et la soustraction. Comme la mémoire que nous utilisons contient des mots de 8 bits, nous choisissons arbitrairement l'encodage suivant pour ces deux instructions [3].

3. Avec un encodage sur 8 bits nous pouvons avoir jusqu'à 256 instructions différentes.

Code		Instruction
`00000100`	(4)	`addition`
`00000110`	(6)	`soustraction`

Exemple de programme : Ceci est un exemple simple de programme que nous voulons exécuter :

```
1    3  + 4
2    12 - 8
```

Nous proposons d'organiser le programme en mémoire en respectant, pour chaque instruction, la structure suivante :

1. instruction
2. 1er opérande
3. 2nd opérande
4. emplacement pour le résultat

Les instructions et les données sont chargées de façon indifférenciée dans la mémoire, seule la position de ces dernières permet de savoir comment nous devons les interpréter.

Pour notre programme d'exemple, le contenu [4] suivant est chargé initialement dans la RAM :

4. Les X indiquent que nous ne connaissons pas la valeur se trouvant initialement dans cette case.

adresse	type du mot stocké	exemple
0	instruction	+
1	donnée (premier opérande)	3
2	donnée (deuxième opérande)	4
3	donnée (résultat)	X
4	instruction	-
5	donnée (premier opérande)	12
6	donnée (deuxième opérande)	8
7	donnée (résultat)	X

Après l'exécution du programme par le processeur, le contenu de la RAM est modifié [5] et contient le résultat des calculs demandés.

5. Les cases qui ont changé sont indiquées en rouge.

adresse	type du mot stocké	exemple
0	instruction	+
1	donnée (premier opérande)	3
2	donnée (deuxième opérande)	4
3	donnée (résultat)	7
4	instruction	-
5	donnée (premier opérande)	12
6	donnée (deuxième opérande)	8
7	donnée (résultat)	4

Fonctionnement de l'automate Vu l'organisation de la RAM qui a été choisie, le fonctionnement de l'automate est simple : à chaque coup d'horloge, il va chercher successivement une instruction, puis le premier opérande, puis le deuxième opérande, calcule le résultat et le stocke. Puis il recommence à l'adresse suivante.

En détail :

1. Premier coup d'horloge : le microprocesseur présente l'adresse 0 à la RAM.

   → La RAM lui présente sur son bus de sortie le contenu de l'adresse 0, qui est la première instruction.

2. Deuxième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM (1).

   → La RAM lui présente sur son bus de sortie le contenu de l'adresse 1, qui est le premier opérande.

3. Troisième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM (2).

   → La RAM lui présente sur son bus de sortie le contenu de l'adresse 2, qui est le deuxième opérande.

   A ce moment-là, le microprocesseur dispose de toutes les données nécessaires au calcul : l'instruction, et les deux opérandes. Il peut donc calculer le résultat.

4. Quatrième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM (3).

   → Il présente sur le bus de donnée en entrée de la RAM le résultat qu'il vient de calculer.

   → Il passe la ligne WRITE de la RAM à l'état haut, pour dire à la mémoire qu'il désire effectuer une écriture.

   → Le résultat du calcul est donc à ce moment-là écrit à l'adresse 3 de la mémoire.

5. Cinquième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM (4).

   → La RAM lui présente sur son bus de sortie le contenu de l'adresse 4, qui est la deuxième instruction.

6. Et on recommence ...

   La figure 4.4 montre ce séquencement temporel.

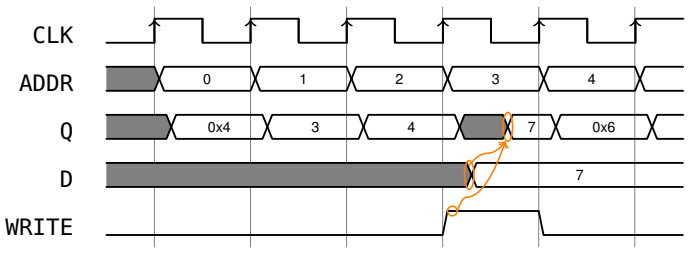


FIGURE 4.4: chronogramme des accès à la mémoire du processeur « linéaire »

4.4.1 *L'architecture du processeur :*

Les adresses : Pour effectuer un calcul, le processeur doit disposer de trois informations :

— l'instruction (l'opération à effectuer)

— l'opérande 1

— l'opérande 2

Il doit en disposer *en même temps*. Mais elles sont stockées en RAM, et ne peuvent être lues que l'une après l'autre. Il faudra donc prévoir de stocker ces trois informations à l'intérieur du processeur pour pouvoir effectuer le calcul (ce que nous verrons dans la suite).

Vu l'organisation de la mémoire, il semble logique de lire ces trois informations de la façon la plus simple possible, c'est-à-dire, dans l'ordre :

— tout d'abord l'instruction,

— puis l'opérande 1,

— puis l'opérande 2.

Ce qui correspond à un parcours linéaire de la mémoire.

De plus, le stockage du résultat s'effectue dans la RAM à l'adresse suivant celle de l'opérande 2. On peut donc doter le processeur d'un compteur qu'on appellera **compteur programme** ou **PC** (Program Counter), qui donnera l'adresse de la RAM à laquelle on est en train d'accéder. Ce compteur sera incrémenté à chaque coup d'horloge et pilotera directement le bus d'adresse de la RAM et qui servira pour les lectures et les écritures.

Les données : Le processeur a un fonctionnement linéaire et l'ordre des actions effectuées est toujours le même :

1. aller chercher une instruction,
2. aller chercher le premier opérande,
3. aller chercher le second opérande,
4. stocker le résultat du calcul.

On peut donc le concevoir comme un automate à quatre états, dont le fonctionnement est circulaire : état 1 $\rightarrow$ état 2 $\rightarrow$ état 3 $\rightarrow$ état 4 $\rightarrow$ état 1 $\rightarrow$ état 2 $\rightarrow \ldots$

État 1 : L'état **IF** (Instruction Fetch)

— le compteur est en train de présenter à la RAM une adresse correspondant à une instruction.
Le processeur récupère sur le bus `Q[7:0]` la contenu de la RAM à cette adresse, c'est-à-dire l'instruction à effectuer.

— il faut stocker cette instruction pour plus tard (quand on aura récupéré les opérandes).
On ajoute donc au processeur un registre de 8 bits disposant d'un enable (entrée d'activation).
L'entrée de ce registre est reliée au bus `Q[7:0]` (sortie de la RAM)
Le signal d'enable de ce registre est mis à l'état haut seulement pendant l'état 1.

État 2 : L'état **OP1F** (Operand 1 Fetch)

— le compteur est en train de présenter à la RAM une adresse correspondant au premier opérande. Le processeur récupère sur le bus `Q[7:0]` la contenu de la RAM à cette adresse, c'est-à-dire l'opérande 1...

— il faut stocker cet opérande, donc, on ajoute un registre 8 bits avec enable, relié à la sortie de la RAM. L'enable de ce registre est mis à l'état haut seulement pendant l'état 2.

État 3 : L'état **OP2F** (Operand 2 Fetch)

— le compteur est en train de présenter à la RAM une adresse correspondant au second opérande.
Le processeur récupère sur le bus `Q[7:0]` la contenu de la RAM à cette adresse, c'est-à-dire l'opérande 2...

— Comme précédemment on stocke cet opérande dans un registre[6] 8 bits, dont l'enable est à l'état haut uniquement durant ce cycle.

État 4 : L'état **WR** (Write back)

— le compteur est en train de présenter à la RAM une adresse correspondant au résultat à stocker.

— le processeur dispose dans ses trois registres de toutes les données pour effectuer le calcul. Il suffit d'ajouter une fonction combinatoire, pour produire le résultat[7]. La sortie de cette fonction combinatoire sera reliée au bus d'entrée de la RAM.

— Parallèlement, le processeur doit mettre le signal WRITE de la RAM à l'état haut, pour dire à la RAM de stocker à l'adresse courante la sortie de la fonction de calcul.

6. On doit stocker l'opérande dans un registre car l'écriture du résultat en RAM ne se fait qu'au cycle d'horloge suivant.
On aurait pu effectuer le calcul dans ce cycle et stocker le résultat pour le cycle suivant mais ça ne change rien au nombre de registres nécessaires. Ce ne sont que des considérations de chemin critique qui permettront de déterminer la meilleure des deux structures.

7. Le résultat dépend des deux opérateurs Op1 et Op2 ainsi que de l'instruction I que nous avons stockés dans les registres correspondants.

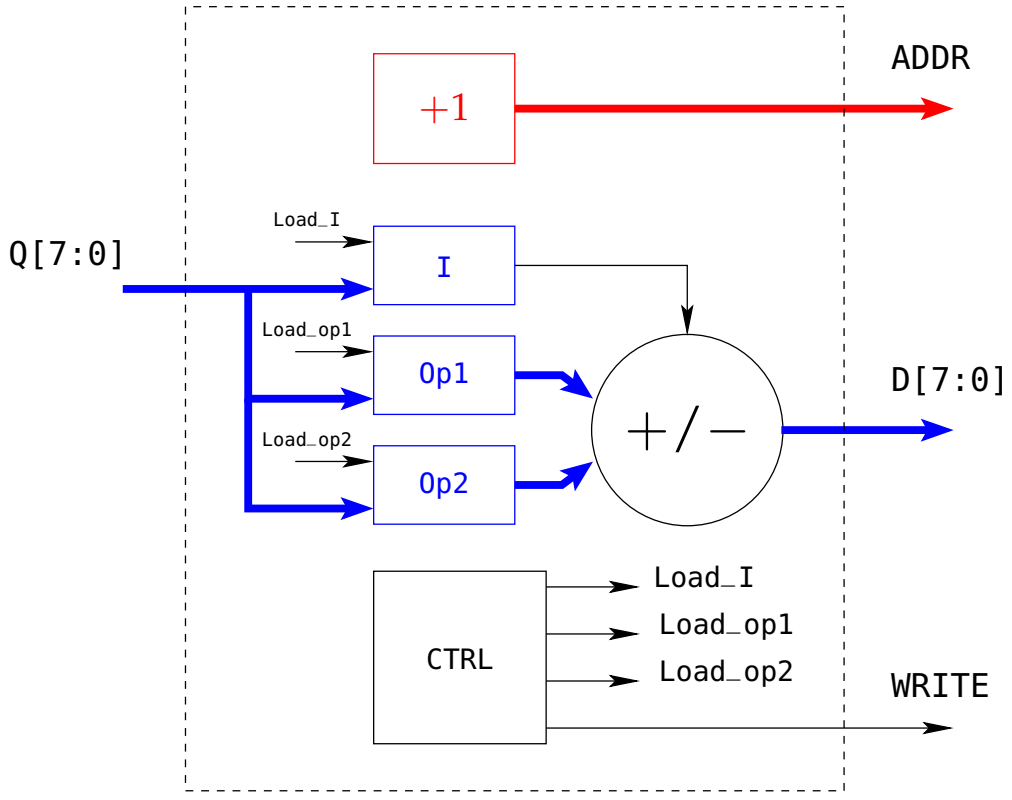


FIGURE 4.5: Architecture du processeur « linéaire »

On obtient donc l'architecture de la figure 4.5 pour notre processeur :

— En rouge : le compteur d'adresse courante.

— En bleu : les trois registres 8 bits, les signaux *load* sont les enable.

— En noir rond : la fonction combinatoire de calcul proprement dite (ALU : pour Arithmetic and Logic Unit).

— En noir carré : la machine à état qui séquence l'ensemble.

Le contrôleur (ou machine à état) « CTRL » est présentée en figure 4.6. Ce diagramme représente les transitions de l'état[8] (Etat) interne du processeur.

8. Ici, les deux bits de poids faible du compteur programme PC peuvent être utilisés pour définir cet état.

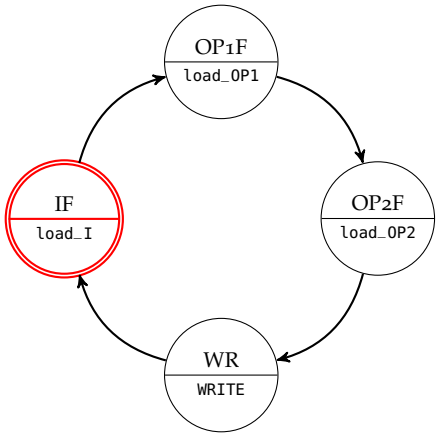


FIGURE 4.6: Séquencement des états du processeur « linéaire »

Les équations des différents signaux générés par ce contrôleur sont :

```
Load_I    <=  (Etat == IF)
Load_OP1  <=  (Etat == OP1F)
Load_OP1  <=  (Etat == OP2F)
WRITE     <=  (Etat == WR)
```

La figure 4.7 représente le chronogramme complet avec l'état des registres internes du processeur ainsi que leurs signaux d'activation (enable).

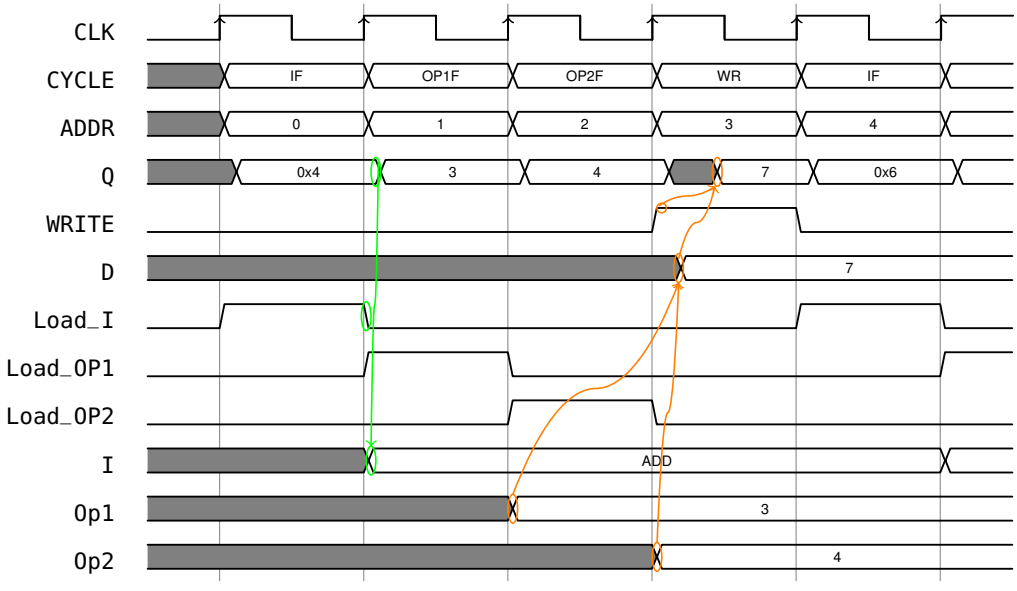


FIGURE 4.7: chronogramme complet pour le processeur « linéaire »

4.5 *Deuxième version du microprocesseur : l'automate avec accumulateur*

L'architecture actuelle ne permet pas de chaîner les calculs (exemple : $3 + 4 + 5$). Pour pouvoir le faire, il y a plusieurs possibilités, voici quelques exemples :

1. Garder le résultat de chaque opération en mémoire, et définir une nouvelle addition [9] qui opère sur un opérande en RAM et le résultat qu'on a gardé.
2. Définir des opérations de manipulation de la RAM avec lesquelles nous pourrions recopier le résultat en RAM à la position d'un des deux opérandes de la prochaine instruction. C'est bien compliqué car il faut connaitre les instructions à venir.
3. Définir une nouvelle addition qui opère sur un opérande à l'endroit habituel en RAM, et sur un autre opérande situé à l'adresse (instruction - 1)

9. On ajoute une instruction pour chaque type d'opérations. Cette nouvelle opération ne nécessitant qu'un seul opérande en RAM, pourra donc être effectuée en 3 ce qui va compliquer le séquencement des états car certaines opérations se feront en 3 cycles et d'autres en 4.

La solution que nous proposons d'implémenter est la suivante :

— Utiliser la première solution [10], mais pour simplifier les choses, et par cohérence, supprimer les opérations sur deux opérandes en RAM.

— Toutes les opérations se feront entre un opérande en RAM, et un gardé dans un registre interne au processeur.

— Et pour rendre cela possible [11], on définit deux nouvelles instructions : chargement de ce registre à partir d'une donnée en RAM et stockage du contenu de ce registre en RAM.

10. l'instruction opère sur un opérande en RAM

11. initialiser le registre interne et stocker le résultat final

4.5.1 *L'accumulateur*

Nous allons doter notre processeur d'un registre interne sur 8 bits, que nous appellerons accumulateur. Toutes les opérations arithmétiques à deux opérandes s'effectueront entre l'accumulateur et une donnée en RAM.

Plus précisément : pour effectuer « $3 + 4$ » et stocker le résultat en RAM, le processeur effectuera la séquence d'instructions suivante :

1. chargement de 3 dans l'accumulateur
2. addition de l'accumulateur avec un opérande en RAM (4)
3. stockage du contenu de l'accumulateur en RAM

Pour effectuer « $3 + 4 + 5$ » :

1. chargement de 3 dans l'accumulateur
2. addition de l'accumulateur avec un opérande en RAM (4)
3. addition de l'accumulateur avec un opérande en RAM (5)
4. stockage du contenu de l'accumulateur en RAM

On ajoute donc deux instructions à notre processeur :

— `LDA` (*load to accumulator*) : chargement de l'accumulateur à partir de la RAM

— `STA` (*store accumulator*) : stockage du contenu de l'accumulateur dans la RAM

Aussi, les instructions d'addition et de soustraction n'ont plus besoin que d'un seul opérande : le deuxième opérande est dans l'accumulateur.

Nous en profitons aussi pour ajouter des instructions pour la manipulation de bits (et, ou et ou-exclusif) qui, comme l'addition et la soustraction, opèrent sur l'accumulateur et un élément venant de la RAM.

La table suivante donne ce nouveau **jeu d'instructions** :

code (8 bits)		**mnémonique**	**fonction**
`00000001`	(1)	`XOR`	le ou-exclusif bit à bit
`00000010`	(2)	`AND`	le et bit à bit
`00000011`	(3)	`OR`	le ou bit à bit
`00000100`	(4)	`ADD`	l'addition
`00000110`	(6)	`SUB`	la soustraction
`00001010`	(10)	`LDA`	le chargement de l'accumulateur à partir de la mémoire
`00001011`	(11)	`STA`	la sauvegarde de l'accumulateur en mémoire

L'organisation en mémoire d'un programme permettant de calculer « $3 + 4 - 1$ » ressemblerait à :

adresse	**type du mot stocké**	**exemple**
`0`	instruction	`LDA`
`1`	donnée	3
`2`	instruction	`+`
`3`	donnée	4
`4`	instruction	`-`
`5`	donnée	1
`6`	instruction	`STA`
`7`	donnée	`X`

On remarque donc qu'une adresse sur deux contient une instruction, une sur deux contient une donnée, soit un opérande, soit un espace pour stocker le contenu de l'accumulateur. À la fin de l'exécution du programme, le résultat du calcul sera disponible à l'adresse `7`.

4.5.2 *L'architecture du processeur avec accumulateur :*

Commençons par observer une séquence d'accès à la mémoire pour l'exécution du programme précédent (voir figure 4.8).

Nous avons une séquence régulière avec, systématiquement, la lecture d'une instruction puis la lecture ou l'écriture d'une donnée. Nous pouvons nommer ces deux cycles :

IF (Instruction Fetch) : lecture de l'instruction, durant lequel :

— on présente l'adresse d'une instruction

— on autorise le chargement du registre instruction (`Load_I`)

DF (Data Fetch) : lecture ou écriture de la donnée

— on présente l'adresse d'une donnée

— on autorise le chargement du registre accumulateur (`Load_Acc`) sauf dans le cas d'un « store » où on autorise l'écriture en RAM (WRITE).

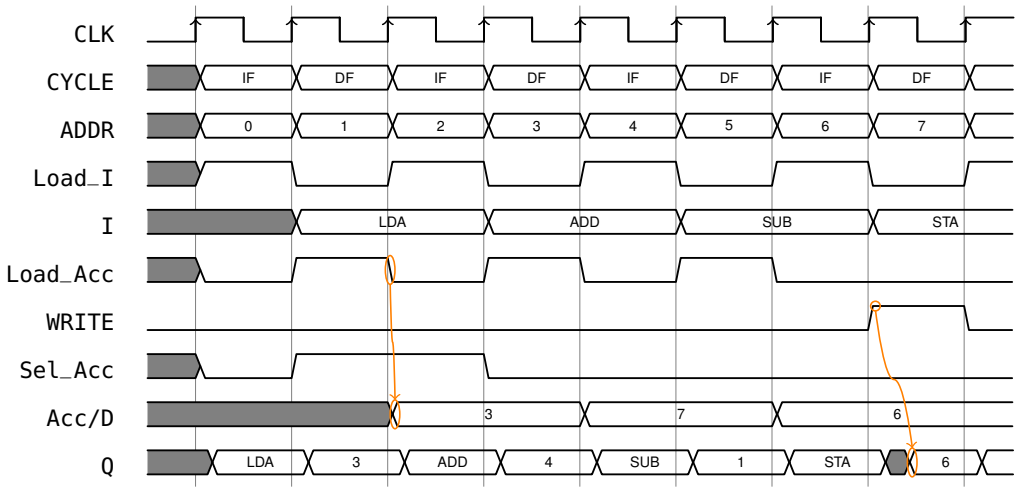


FIGURE 4.8: chronogramme pour le processeur avec « accumulateur »

L'architecture du processeur avec accumulateur est représentée par la figure 4.9. La séquence des adresses est toujours linéaire, nous avons toujours besoin d'un compteur programme (PC).

Les opérations arithmétiques et logiques se font dans l'ALU. L'opération à effectuer est choisie en fonction de la valeur du registre d'instruction [12].

L'accumulateur peut être soit chargé directement à partir de la RAM, soit modifié par le résultat d'un calcul venant de l'ALU. Son entrée est reliée aux deux par l'intermédiaire d'un multiplexeur qui permet, en fonction de l'instruction, de sélectionner la bonne entrée. Sa sortie est reliée directement à l'entrée de la RAM pour pouvoir sauvegarder le résultat [13], ainsi qu'à la seconde entrée de l'ALU.

12. Plusieurs fonctions combinatoires en parallèle avec un multiplexeur pour rediriger vers la sortie le bon calcul

13. dans le cas d'une instruction store (`STA`)

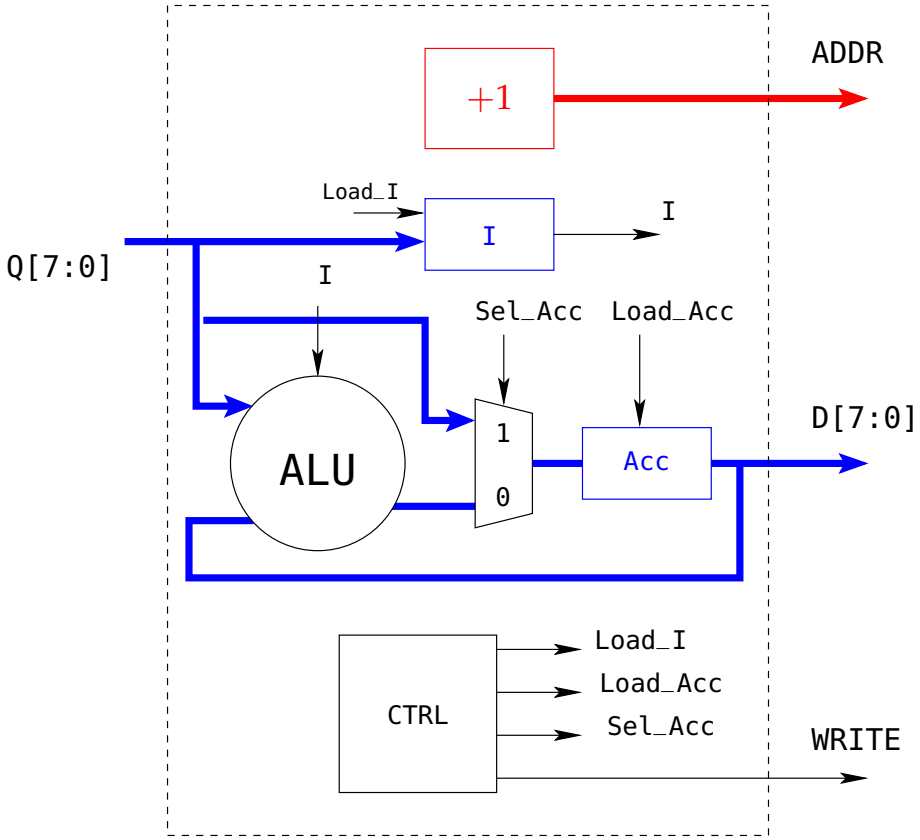


FIGURE 4.9: Architecture du processeur avec « accumulateur »

Le diagramme d'états [14] du contrôleur est représenté dans en figure 4.10. Les équations des différents signaux générés par ce contrôleur sont :

14. Ici, le bit de poids faible du PC peut être utilisé

```
Load_I    <= (Etat == IF )
Sel_Acc   <= (   I == LDA)
Load_Acc  <= (Etat == AF ) && (I !=  STA)
WRITE     <= (Etat == AF ) && (I ==  STA)
```

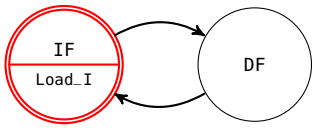


FIGURE 4.10: Séquencement des états du processeur avec « accumulateur »

4.6 *Troisième version du microprocesseur : l'automate avec accumulateur et indirection*

Séparation des données des instructions :

Imaginez qu'on souhaite séparer le code des données, pour :

— exécuter un même code sur des données différentes (sans le dupliquer pour chaque groupe de données)

— exécuter différents codes sur des mêmes données (sans dupliquer les groupes de données)

— exécuter un code sur des données qui ne sont pas connues avant l'exécution du programme (par exemple, au début du programme on demande à l'utilisateur d'entrer des valeurs)

Pour le moment, notre processeur ne sait pas faire : nous devons connaître les données au moment du pré-chargement de la RAM avec le code.

Il faudrait disposer d'instructions de manipulation du contenu de la RAM à des adresses arbitraires (on ne modifierait que des données, pas le code) Cela permettrait de modifier les zones de la mémoire dans lesquelles se trouvent les opérandes. Mais c'est peut-être un peu compliqué d'avoir à modifier plein de zones éparses.

Pour une meilleure organisation, on pourrait séparer le code des données. On aurait, en RAM, une zone avec les instructions et une zone avec les données. Il suffirait juste d'aller modifier la zone des données, et d'exécuter le code générique qui saurait, pour chaque instruction, où trouver les bons opérandes.

Dans l'état actuel, les instructions en RAM occupent de deux octets, un pour le code de l'instruction et l'autre pour l'opérande. Nous proposons de modifier toutes les instructions pour que le second octet soit l'adresse de l'opérande et non plus sa valeur.

Par exemple, pour effectuer « $3 + 4, 3 - 1$ » on pourra avoir une organisation en RAM comme suit :

adresse	type du mot stocké	exemple	zone
0	instruction	LDA	zone de code
1	adresse de l'opérande	100	
2	instruction	+	
3	adresse de l'opérande	101	
4	instruction	STA	
5	adresse de l'opérande	103	
6	instruction	LDA	
7	adresse de l'opérande	100	
8	instruction	-	
9	adresse de l'opérande	102	
10	instruction	STA	
11	adresse de l'opérande	104	
...	...	...	
100	donnée	3	zone de données
101	donnée	4	
102	donnée	1	
103	donnée	X	
104	donnée	X	
...	...	...	

Comme vous pouvez le voir le programme est mis dans la partie basse de la RAM (commençant à l'adresse 0 [15]) alors que les données

15. Le fait que le programme commence à l'adresse 0 simplifie le démarrage après une remise à zéro

sont mises dans la partie haute de la RAM (commençant à l'adresse 100 [16]).

16. le choix de cette adresse est arbitraire

Après l'exécution du programme la zone des données est modifiée et contient les résultats comme suit.

adresse	type du mot stocké	exemple	zone
...	...	...	
100	donnée	3	zone de données
101	donnée	4	
102	donnée	1	
103	donnée	7	
104	donnée	1	
...	...	...	

Cette séparation entre une zone contenant les données et une zone contenant les instructions est une séparation simple. Dans un programme plus complexe (et plus réaliste) on aurait plusieurs zones pour les données en fonction du fait qu'elles soient constantes, modifiables, non définies au début du programme...

Fonctionnement de l'automate : Vu la nouvelle structure du programme en mémoire RAM, dans le processeur, nous devons suivre les trois étapes suivantes :

1. lire l'instruction (IF : Instruction Fetch),
2. lire l'adresse de l'opérande [17] (AF : Adress Fetch),
3. présenter l'adresse de l'opérande à la RAM pour exécuter l'instruction (EX : Execution).

17. Nous devons aussi stocker cette adresse dans un registre du processeur car elle est utilisée au cycle suivant

Cette séquence est présentée en figure 4.11. Les instructions s'exécutent donc maintenant en trois cycles.

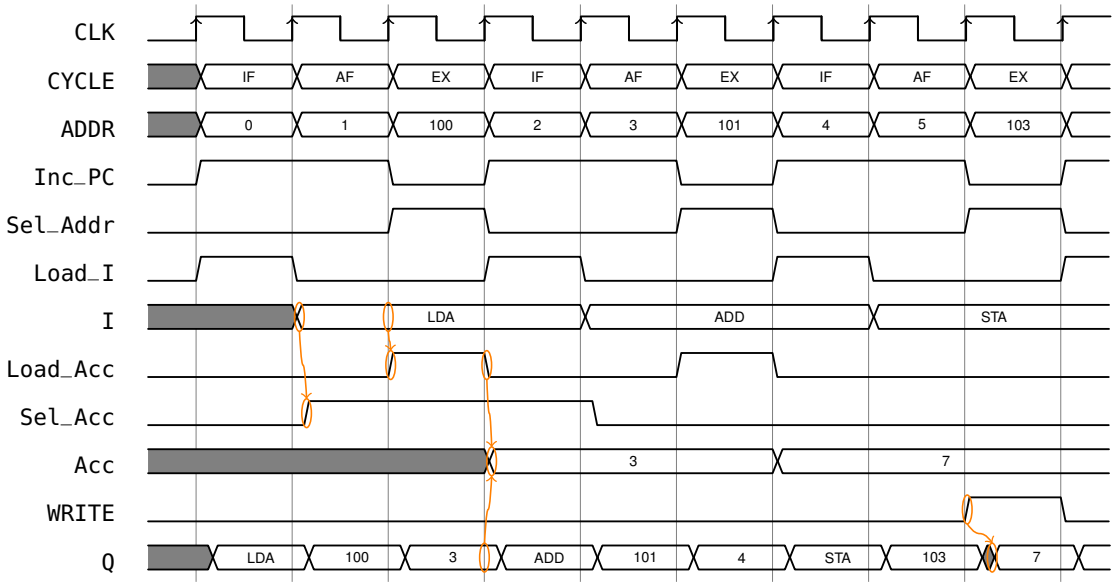


FIGURE 4.11: chronogramme pour le processeur avec « indirection »

Les accès à la RAM ne sont donc plus simplement linéaires. Dans l'exemple de programme nous avons la séquence d'adresses suivante :

1	`0`	*(adresse de l'instruction)*
2	`1`	*(adresse de l'adresse de l'opérande)*
3	`100`	*(adresse de l'opérande)*
4	`2`	*(adresse de l'instruction)*
5	`3`	*(adresse de l'adresse de l'opérande)*
6	`101`	*(adresse de l'opérande)*
7	`4`	*(adresse de l'instruction)*
8	`5`	*(adresse de l'adresse de l'opérande)*
9	`103`	*(adresse de l'opérande)*
10	`...`	*(adresse de l'instruction)*

Les adresses du code sont globalement linéaires (0, 1, 2, 3...), celles des données ne le sont plus (elles sont arbitraires). Nous pouvons donc garder le compteur programme, tel que défini pour les versions précédentes du processeur en prenant soin de présenter sur le bus d'adresse RAM :

— le compteur programme durant les deux premiers cycles (que l'on incrémente à chaque fois)

— puis le contenu du registre d'adresse (adresse de l'opérande à aller chercher) pendant le troisième cycle (et ici le compteur d'adresse ne doit pas être incrémenté)

Pour cela, le compteur programme doit pouvoir être stoppé. Un signal ide contrôle supplémenta tire `INCR_PC` permet d'autoriser le fait qu'il soit incrémenté. La figure 4.12 montre comment ce signal de contrôle peut être ajouté.

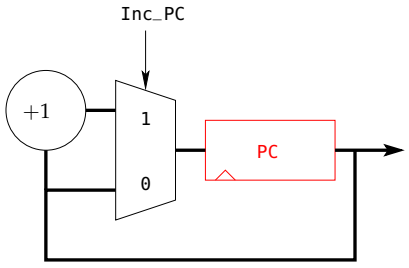


FIGURE 4.12: Structure du compteur programme du processeur avec indirection

Le registre d'adresse est chargé au cycle numéro 2 (AF) et son contenu n'est utile qu'au cycle numéro 3 (EX). Il n'est donc pas nécessaire de le piloter avec un signal enable. Il peut rester tout le temps actif : son contenu sera indéterminé pendant les cycles 1 (IF) et 2 (AF), mais ce n'est pas grave, il n'est pas utilisé pendant ces cycles-là.

Un multiplexeur supplémentaire permet durant le cycle numéro 3 (EX) de présenter sur le bus d'adresse de la RAM le contenu du registre d'adresse à la place de PC.

La figure 4.13 montre l'architecture interne du processeur avec cette sélection d'adresse.

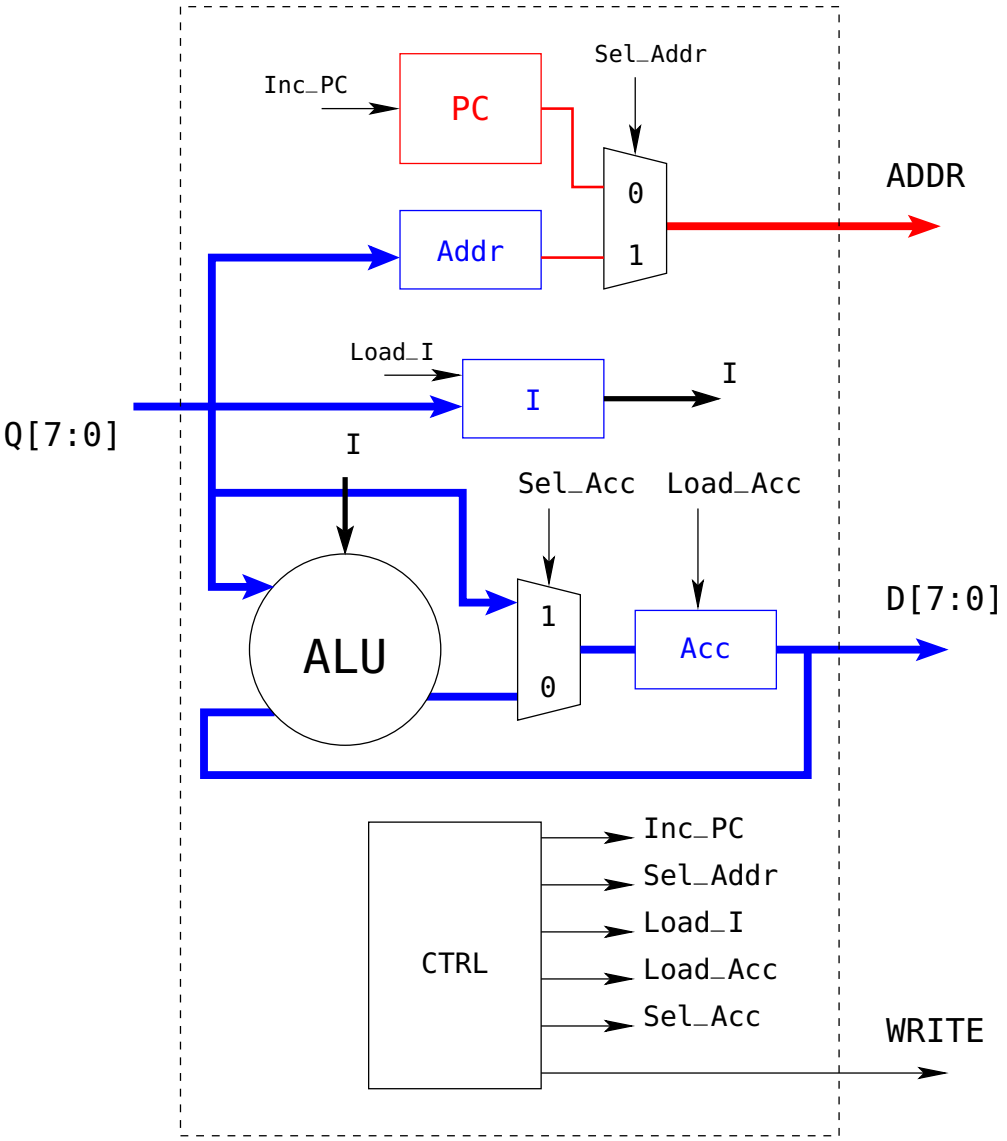


FIGURE 4.13: Architecture du processeur avec « indiretion »

Le diagramme d'états du contrôleur est représenté en figure 4.14. Ici, les bits de poids faible du PC ne peuvent plus être utilisés.

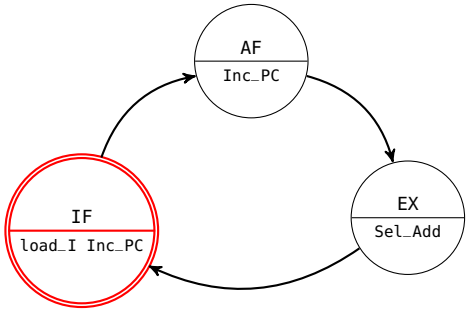


FIGURE 4.14: Séquencement des états du processeur avec « indirection »

4.7 *Quatrième version du microprocesseur : le processeur RISC*

Dans cette partie nous allons ajouter des fonctionnalités au processeur pour pouvoir exécuter des programmes plus réalistes.

Exécution conditionnelle : Jusqu'ici, l'architecture du processeur ne permet que d'effectuer que des calculs linéaires sur une suite fixe d'instructions. Les données sont potentiellement inconnues mais leurs adresses de stockage sont connues.

Nous allons maintenant lui ajouter des instructions de saut conditionnels et, tant qu'on y est, inconditionnels.

Les « Flags » :

Pour cela, chaque opération (logique ou arithmétique) va positionner deux signaux sur l'état d'un résultat de calcul. Ces indicateurs (**flags**) seront mémorisés pour l'instruction suivante et ne devront être modifiés que si le contenu de l'accumulateur est modifié.

— *C* pour la retenue (*Carry*) :
  — mis à 1 si l'opération courante est une opération arithmétique et donne lieu à une retenue,
  — mis à 0 si l'opération courante est une opération arithmétique et ne donne pas lieu à une retenue,
  — mis à 0 si on charge l'accumulateur avec une nouvelle donnée.
— *Z* pour zéro :
  — mis à 1 si ce qui est chargé dans l'accumulateur est nul.
  — mis à 0 dans tous les autres cas.

La génération de deux signaux C et Z est combinatoire et peut être effectuée dans l'ALU.

Il suffit pour cela d'ajouter deux registres 1 bits pour stocker ces deux signaux, pilotés par le même *enable* que l'accumulateur `LOAD_ACC`, qu'on appellera maintenant `LOAD_AZC`. Nous pouvons considérer que Z et C font partie de l'accumulateur qui devient donc un registre sur 10 bits : 8 de donnée, 1 pour Z, un pour C.

Ces *flags* permettront de changer le flot d'exécution en fonction du résultat d'un calcul.

Remarque : Comme nous mémorisons la retenue C, nous pouvons nous en servir pour ajouter deux instructions permettant d'enchaîner les opérations d'addition (ou de soustraction), en prenant en compte la retenue précédente. Il suffit pour cela de refaire entrer la sortie du registre C sur la retenue entrante de l'additionneur/soustracteur de l'ALU. Nous aurons donc deux instructions supplémentaires `ADDC` et `SUBC` dans notre jeu d'instructions.

Les sauts :[18]

18. *jump* en anglais.

Nous ajoutons les trois instructions de saut (ou branchement) suivantes :

— *JMP* : saut inconditionnel :
  → L'exécution de cette instruction fait sauter l'exécution du programme directement à une adresse passée comme opérande.
— *JNC* : saut si pas de retenue (*Jump if No Carry*) :
  → Idem à JMP, mais seulement si C est nul. Sinon, on continue à l'adresse suivante
— *JNZ* : saut si non nul (*Jump if No Carry*) :
  → Idem à JMP, mais seulement si Z est nul. Sinon, on continue à l'adresse suivante

Pour implémenter les sauts, il suffit de se donner la possibilité de remplacer le contenu de PC par la valeur de l'adresse de destination lue en RAM.

Le PC devient donc un peu plus complexe. C'est globalement un compteur, mais il :

— est incrémenté si son signal de commande `INCR_PC = 1`,

— est chargé avec une nouvelle valeur si un signal de chargement `LOAD_PC = 1`,

— si `LOAD_PC` et `INCR_PC` valent 1, c'est `LOAD_PC` qui prime.

Ceci peut être implémenté comme montré sur la figure 4.15.

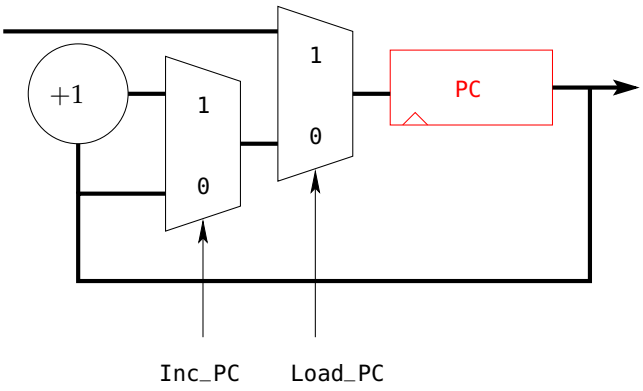


FIGURE 4.15: Structure du compteur programme du processeur complet

Les instructions de sauts ainsi que l'instruction d'écriture en mémoire `STA`, ne changent jamais l'état de l'accumulateur ni celui des flags. Tant qu'on y est nous pouvons définir une instruction explicite `NOP` (*No OPeration*) qui ne fait rien. Cette instruction permettra d'avoir des pauses durant lesquelles l'état du processeur n'est pas modifié.

Elle n'a pas besoin d'opérande, et pourrait donc être stockée sur un seul octet (au lieu de deux pour les autres). Mais cela casserait la régularité de l'organisation du programme en mémoire et compliquerait la gestion de la machine à états pour générer les signaux `LOAD_PC` et `INCR_PC`.

Nous acceptons donc de perdre un octet de mémoire pour ne pas casser l'organisation de la mémoire. L'instruction `NOP` sera accompagnée d'un opérande qui ne servira pas.

Une instruction sera toujours exécutée en trois cycles. La seule modification de la machine à état sera dans le calcul du signal qui autorise la modification du PC (`LOAD_ACC`).

Nous ajoutons aussi deux instructions de rotation de données (vers la droite ou vers la gauche en incluant le bit de retenue) :

— `ROL` : `ACC[7:0]` devient {`ACC[6:0], C`} et `C` devient `ACC[7]`.

— `ROR` : `ACC[7:0]` devient {`C, ACC[7:1]`} et `C` devient `ACC[0]`.

Ces opérations sont combinatoires et seront donc implémentées dans l'ALU.

Le jeu d'instructions plus complet devient donc :

code (8 bits)	mnémonique	fonction	effet
00000000 (0)	NOP	Pas d'opération	Aucun
00000001 (1)	XOR	ou-exclusifle bit à bit	Acc = Acc XOR (AD)
00000010 (2)	AND	et bit à bit	Acc = Acc AND (AD)
00000011 (3)	OR	ou bit à bit	Acc = Acc OR (AD)
00000100 (4)	ADD	addition	Acc = Acc + (AD)
00000101 (5)	ADC	addition avec retenue entrante	Acc = Acc + (AD) + C
00000110 (6)	SUB	soustraction	Acc = Acc - (AD)
00000111 (7)	SBC	soustraction avec retenue entrante	Acc = Acc - (AD) - C
00001000 (8)	ROL	rotation de l'accumulateur vers la gauche	{C, Acc} = {Acc[7 :0], C }
00001001 (9)	ROR	rotation de l'accumulateur vers la droite	{C, Acc} = {Acc[0], C, Acc[7 :1] }
00001010 (10)	LDA	chargement de l'accumulateur à partir de la mémoire	Acc = (AD)
00001011 (11)	STA	sauvegarde de l'accumulateur en mémoire	(AD) = Acc
...	...		
00001101 (13)	JMP	saut inconditionnel	PC = AD
00001110 (14)	JNC	saut si pas de retenue	PC = AD si C=0
00001111 (15)	JNZ	saut si resultat non nul	PC = AD si Z=0

Remarques :

— AD est le deuxième octet (en RAM) de l'instruction

— (AD) est la valeur en RAM stockée à l'adresse AD

Avec toutes ces modifications, l'architecture du processeur évolue comme le montre la figure 4.16.

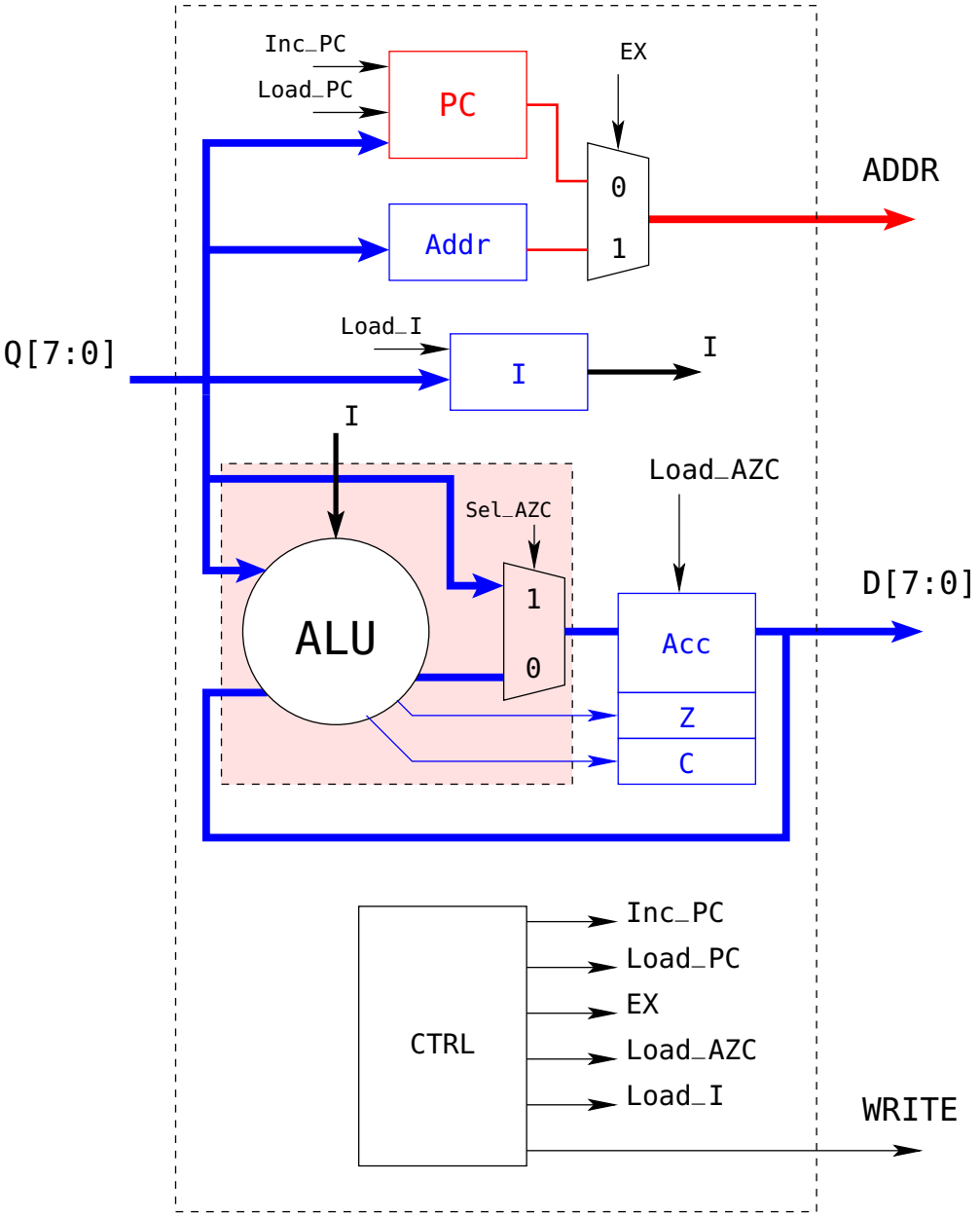


FIGURE 4.16: Architecture du processeur complet

La figure 4.18 montre comme sont générés les signaux de contrôle des différents registres du processeur. Le fonctionnement se décompose en trois phases (`IF`,`AF` et `EX`) comme pour la version précédente.

On peut noter que certains signaux (`load_I` par exemple) ont systématiquement la même allure (on ira toujours chercher l'instruction au 1e cycle) alors que d'autres signaux (`WRITE`) dépendent de l'instruction en cours (on écrit en mémoire que si l'instruction est `STA`).

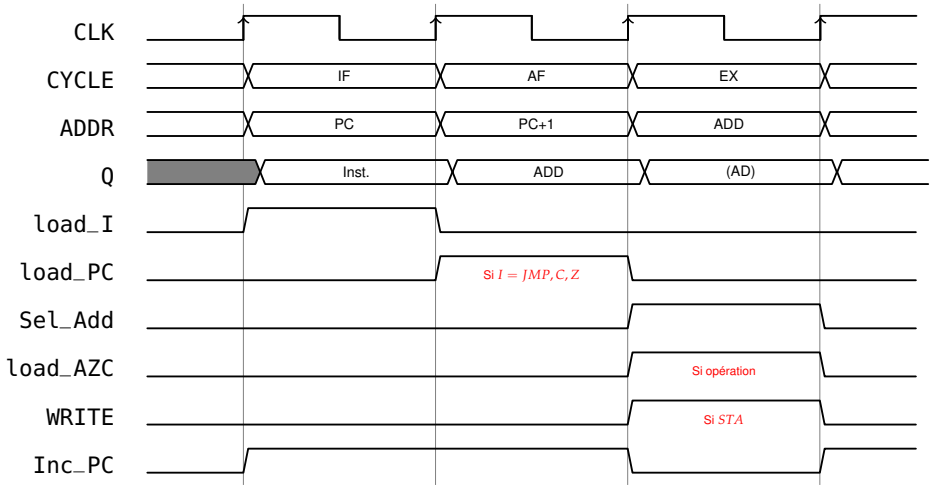


FIGURE 4.17: chronogramme pour le processer RISC

La figure 4.18 montre le séquencement des états du contrôleur. Les équations des différents signaux générés par ce contrôleur sont :

```
Load_I    <= (Etat == IF )
Inc_PC    <= (Etat == IF ) || (Etat == AF)
Load_PC   <= (Etat == AF ) && ( I == JMP || ...)
Sel_Add   <= (Etat == EX )
Sel_AZC   <= (   I == LDA)
Load_AZC  <= (Etat == EX ) && (I != NOP || ...)
WRITE     <= (Etat == EX ) && (I ==  STA)
```

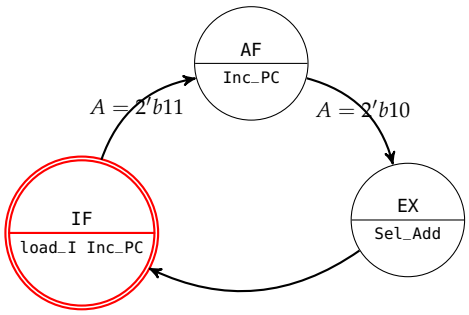


FIGURE 4.18: Séquencement des états du processeur complet

4.8 *Les périphériques*

Une instruction pour le buzzer : Rajouter une instruction pour commander le buzzer. Cette instruction permet de commander ce qui est appelé un **port**.

Pour commander le buzzer un seul bit suffit mais comme dans notre processeur toutes les données manipulées font 8 bits de large, nous pouvons avoir un port faisant aussi 8 bits. Le bit 0 serait connecté au buzzer tandis que les autres pourraient être utilisés pour commander d'autres éléments (de leds par exemple).

L'instruction que nous ajoutons (OUT) va permettre de charger un registre connecté à ce port de sortie avec une valeur lue en mémoire.

code (8 bits)		**instruction**	**effet**
...		...	
00001100	(12)	OUT	PORT = (AD) (ou BZ = (AD)[0])
...		...	

Fonctionnellement cette instruction n'est pas bien différente de l'instruction `LDA`, sauf qu'ici ce n'est pas l'accumulateur qui est chargé. La figure 4.19 montre comment ce registre pourrait être ajouté.

Le contrôleur doit générer un signal de commande supplémentaire pour le registre `PORT`. Ce signal vaut 1 dans le cycle d'exécution si l'instruction est `OUT`.

```
Load_AZC  <= (Etat == EX ) && (I == OUT)
```

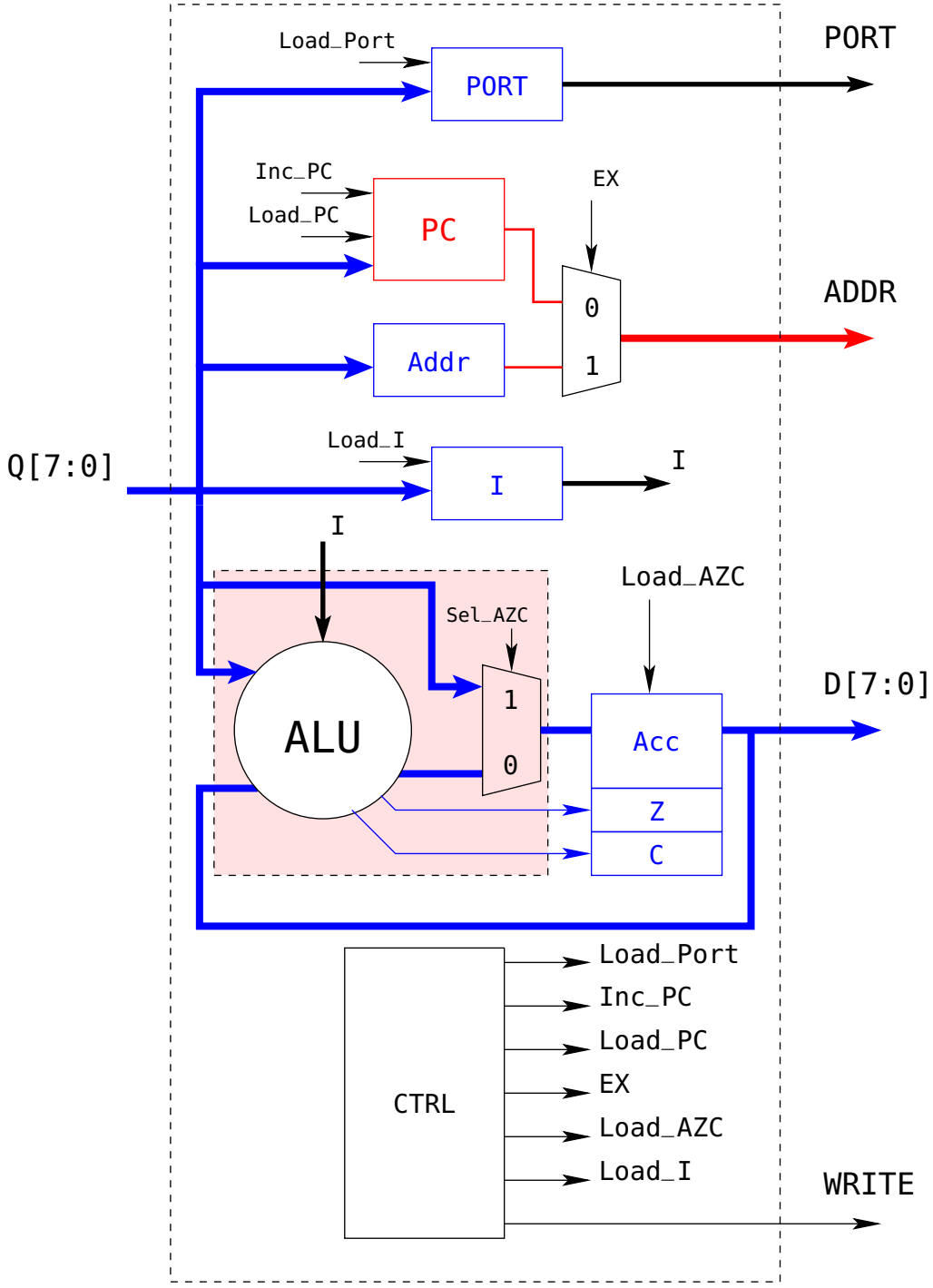


FIGURE 4.19: Architecture du processeur complet avec un port en sortie

5
La logique CMOS

Dans ce chapitre, notre objectif est de faire le lien entre le monde « virtuel » du traitement numérique et la réalisation physique des composants qui le supportent. Il s'agit, sans entrer dans les détails subtils du fonctionnement de ces composants, d'établir les grandes lois technologiques et économiques régissant l'industrie micro-électronique. Ainsi, il sera possible, d'une part de comprendre les enjeux de performance des composants réalisés (vitesse de traitement, consommation) et d'autre part d'intégrer ces enjeux dans un contexte économique plus global.

5.1 Construisons des fonctions logiques

5.1.1 Construisons un inverseur

La première étape de la réalisation « physique » de fonctions booléennes consiste à définir arbitrairement une convention liant un état booléen (0 ou 1) à une grandeur physique. Nous pouvons choisir, par exemple, la valeur d'une tension [1] aux bornes d'un dipôle. Le plus simple, pour cela est de disposer d'une source de tension fixe servant de référence pour la génération des différents états des variables. Nous supposons dans la suite du cours que nous disposons d'une telle source d'alimentation [2] connectée entre V_{dd} (borne positive) et V_{ss} (référence de masse). Par convention, un signal électrique s :

— représentera la valeur booléenne 0 si $V(s) = V_{ss}$,

— représentera la valeur booléenne 1 si $V(s) = V_{dd}$.

Supposons, maintenant, que nous disposons d'un composant se comportant comme un interrupteur idéal piloté par une tension. En nommant « V_g » la tension de commande de cet interrupteur, mesurée par rapport à V_{ss}, les états de l'interrupteur sont les suivants :

— $V_g = 0$: interrupteur ouvert,

— $V_g = V_{dd}$: interrupteur fermé.

Nous disposons enfin d'une charge résistive R_{load}. Nous construisons un schéma de référence composé de la source d'alimentation, de l'interrupteur commandé et de la charge résistive (figure 5.1).

Nommons « e » le signal de commande de l'interrupteur dans le schéma de référence, et nommons « y » le signal commun entre la résistance de charge et l'interrupteur.

1. Tout phénomène physique facilement réductible à 2 états pourrait être utilisé. L'aspect fondamental du choix est la possibilité d'effectuer des « calculs » directement au moyen de ces grandeurs physiques.
2. Attention : la présence de cette source d'alimentation implique que la réalisation d'un calcul nécessite forcément la consommation d'une certaine quantité d'énergie

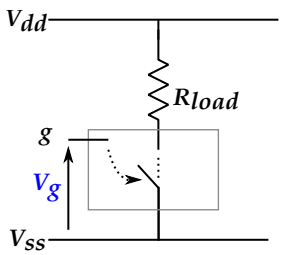


FIGURE 5.1: Schéma de référence utilisant un interrupteur commandé

Dans une première étape, (figure 5.2) nous fixons la tension V_e du signal e à la valeur 0 : l'interrupteur est ouvert. Au repos, aucun courant ne traverse la charge R_{load} donc la chute de potentiel à ses bornes est nulle : La tension en sortie V_y est égale à V_{dd}.

Si maintenant, (figure 5.3) nous fixons la tension V_e à la valeur V_{dd}, alors l'interrupteur est fermé. L'interrupteur étant idéal, la tension V_y est égale à V_{ss}.

En utilisant notre convention liant « booléen » à « tension » nous avons réalisé la fonction booléenne inverseuse ($y = \overline{e}$).

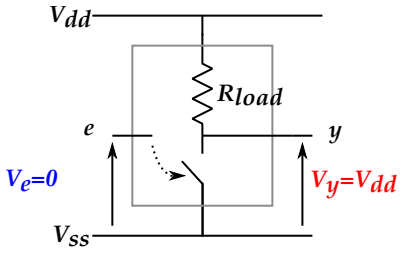


FIGURE 5.2: Tension de commande égale à 0

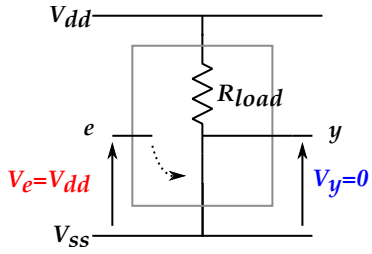
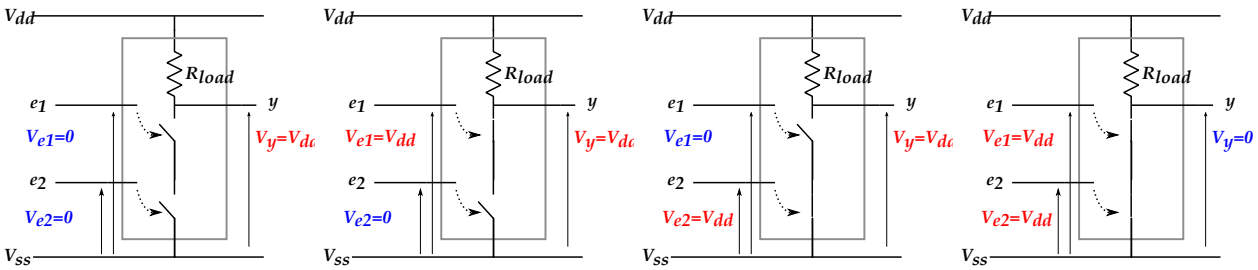


FIGURE 5.3: Tension de commande égale à V_{dd}

5.1.2 *Construisons des fonctions booléennes plus élaborées*

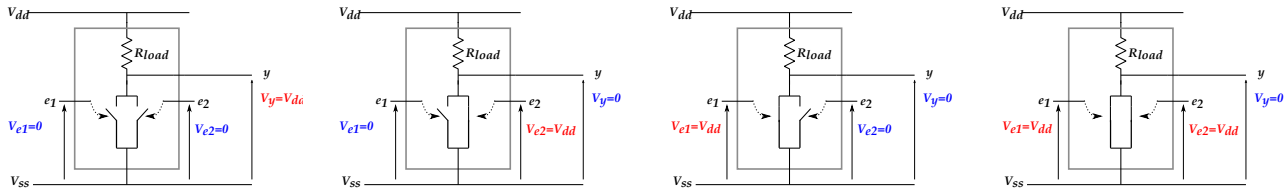
En jouant sur des assemblages d'interrupteurs en série, nous pouvons créer la fonction non-et à 2 entrées ($y = \overline{e_1 \cdot e_2}$). Les quatre figures suivantes représentent l'état des interrupteurs et de la sortie pour les quatre entrées possibles.



(a) Entrées 00 (b) Entrées 01 (c) Entrées 10 (d) Entrées 11

FIGURE 5.4: La fonction non-et à deux entrées

En jouant sur des assemblages d'interrupteurs en parallèle, nous pouvons de même créer la fonction non-ou à 2 entrées ($y = \overline{e_1 + e_2}$) :



(a) Entrées 00 (b) Entrées 01 (c) Entrées 10 (d) Entrées 11

FIGURE 5.5: La fonction non-ou à deux entrées

5.1.3 *En conclusion c'est simple, mais...*

Cette première approche prouve que nous pouvons créer « physiquement » des fonctions booléennes simples. Dans le chapitre 1, nous avons déjà montré comment passer de la logique à l'arithmétique : nous pouvons donc virtuellement réaliser n'importe quelle fonction

de traitement combinatoire. La réalisation de fonctions de logique séquentielle n'est guère plus compliquée.

Cela dit, nos fonctions logiques ne sont guère réalistes.

D'une part ces fonctions consomment de l'énergie en permanence lorsque leur valeur de sortie est 0. En effet, dans cette situation, un courant permanent s'établit dans la résistance de charge (ce problème existait dans les premiers microprocesseurs réalisés au siècle dernier).

Nous aimerions réaliser des microprocesseurs qui ne consomment de l'énergie que lorsqu'ils effectuent des calculs, c'est-à-dire lorsque les signaux d'entrée et de sortie des portes logiques changent d'état.

D'autre part les physiciens ne savent pas réaliser d'interrupteur idéal (à des températures de fonctionnement raisonnables), ainsi le niveau électrique en sortie n'atteint pas exactement V_{ss} (lorsque la sortie est au 0 logique). Il est alors difficile de garantir la bonne ouverture des interrupteurs des portes situées en aval de la porte à l'état 0.

L'objectif de la section suivante est de s'appuyer sur les composants « réalisables » par les « technologues » pour construire la logique robuste universellement utilisée dans les circuits électroniques depuis plusieurs décennies : la logique **CMOS**.

5.2 *La logique CMOS*

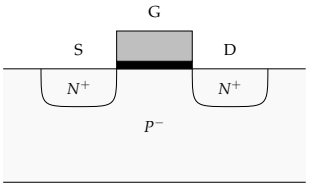
5.2.1 *Transistors MOS complémentaires*

L'interrupteur commandé en tension mis à disposition par les technologues est le transistor MOS (acronyme de « Metal Oxyde Semiconductor »). Ce composant exploite la possibilité d'établir un courant dans un matériau semi-conducteur (le silicium) entre deux électrodes que nous nommerons la « Source » et le « Drain » du transistor.

La zone ou s'établit le courant (le « Canal » du transistor) est située sous une électrode de commande appelée la « Grille » du transistor. Le canal est isolé de la grille par une fine couche d'isolant (l'oxyde). Cette structure verticale Grille/Oxyde/Canal est à l'origine du nom du transistor. Enfin, le dopage (implantation d'impuretés donnant ou acceptant des électrons) permet de spécialiser le comportement du transistor à la fabrication : on peut ainsi créer des courants de charge négatives (électrons) ou de charges positives (trous).

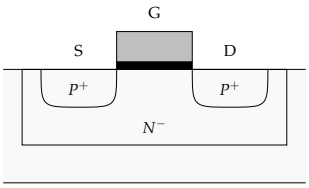
Dans le cas du transistor nMOS (ou MOS à canal N) , une différence de potentiel V_{gs} suffisamment positive ($V_{gs} > V_t$) établie entre la Grille et la Source permet de faire circuler un courant allant du Drain vers la Source. La constante V_t est appelée tension de seuil du transistor. On peut, pour simplifier, considérer le transistor nMOS comme un interrupteur fermé si ($V_{gs} > V_t$) et ouvert dans le cas contraire.

Dans le cas du transistor pMOS (ou MOS à canal P) , une différence de potentiel V_{gs} suffisamment négative ($V_{gs} < -|V_t|$) établie



— Canal N
— Courant d'électrons
— Passant si $V_{gs} > V_T$

FIGURE 5.6: Transistor nMOS



— Canal P
— Courant de trous
— Passant si $V_{gs} < |V_T|$

FIGURE 5.7: Transistor pMOS

entre la Grille et la Source permet de faire circuler un courant allant de la Source vers le Drain. La constante V_t est appelée tension de seuil du transistor. On peut, pour simplifier, considérer le transistor pMOS comme un interrupteur fermé si ($V_{gs} < -|V_t|$) et ouvert dans le cas contraire.

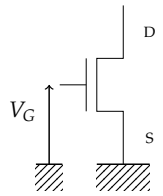


FIGURE 5.8: Transistor nMOS, interrupteur connecté à la masse

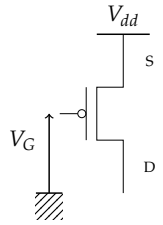


FIGURE 5.9: Transistor pMOS, interrupteur connecté à l'alimentation

5.2.2 *L'inverseur CMOS*

Nous allons exploiter les propriétés duales des deux transistors nMOS et pMOS pour réaliser des fonctions booléennes « efficaces ».

Dans une première étape, nous choisissons une tension d'alimentation V_{dd} dont la valeur est supérieure à la tension de seuil V_t des transistors nMOS et pMOS. Dans la pratique, cette tension d'alimentation est 3 à 4 fois plus élevée que la tension de seuil, ce qui garantit la bonne ouverture ou fermeture des interrupteurs.

Nous spécialisons les transistors nMOS pour réaliser des interrupteurs connectés à la masse. Ainsi, en reprenant la convention logique de la section précédente :

— si V_g, tension de grille du transistor nMOS, est égale à V_{ss} alors l'interrupteur est ouvert.

— si V_g, tension de grille du transistor nMOS, est égale à V_{dd} alors l'interrupteur est fermé.

Nous spécialisons les transistors pMOS pour réaliser des interrupteurs connectés à la tension d'alimentation. Ainsi, en reprenant la convention logique de la section précédente :

— si V_g, tension de grille du transistor pMOS, est égale à V_{ss} alors l'interrupteur est fermé.

— si V_g, tension de grille du transistor pMOS, est égale à V_{dd} alors l'interrupteur est ouvert.

Nous pouvons maintenant joindre les deux schémas précédents pour obtenir le schéma de la figure 5.10). Les deux transistors nMOS et pMOS sont pilotés par un même signal e. Lorsque V_e est égal à V_{ss} alors l'interrupteur nMOS est ouvert et l'interrupteur pMOS est fermé. La tension en sortie est donc V_{dd}. Inversement lorsque V_e est égal à V_{dd} alors l'interrupteur pMOS est ouvert et l'interrupteur nMOS est fermé. La tension en sortie est donc V_{ss}.

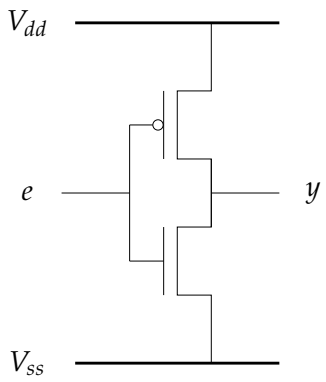


FIGURE 5.10: L'inverseur CMOS

Nous avons bien réalisé la fonction inverseuse, mais dans ce cas nous avons la garantie qu'il n y a consommation d'énergie qu'au moment des transitions entre deux états différents de la fonction. A l'état stable, il n'y a aucun courant permanent entre l'alimentation et la masse.

5.2.3 *Généralisation à une fonction logique complexe*

La structure de l'inverseur CMOS peut être généralisée (figure 5.11) à certaines fonctions logiques à N entrées $(e_1, e_2, ...)$ et une sortie S. Un réseau d'interrupteurs nMOS est connecté à la masse du montage, les grilles des différents transistors sont connectées aux entrées e_i.

Par le biais des associations de transistors nMOS montés en série et en parallèle il est possible de rendre ce réseau « passant » pour certaines combinaisons des entrées e_i et « bloqué » pour les autres combinaisons des entrées.

De même, un réseau de transistors pMOS est connecté à l'alimentation du montage, et ce réseau peut lui-même être construit de façon à être bloqué lorsque le réseau nMOS est passant et réciproquement [3].

Ainsi, la sortie S est « électriquement » fixée à V_{dd} ou V_{ss} quelles que soient les combinaisons d'entrées : nous avons bien réalisé une fonction booléenne.

La figure 5.12 présente le schéma de la porte logique CMOS « non-et à deux entrées » construite en respectant les règles précédentes. Nous laissons au lecteur le soin de vérifier la valeur de la tension sur la sortie *Out* de ce schéma pour les quatre configurations des tensions sur les entrées A et B.

Cette méthode ne permet de construire que les fonctions booléennes pouvant s'exprimer sous la forme $S = \overline{\sum \prod e_i}$, c'est-à-dire les fonctions booléennes résultant de l'assemblage de « et » et de « ou » des entrées e_i et dont la sortie est inversée [4].

Nous détaillons, ici, quelques cas de figure :

— $S = \overline{a + b}$: réalisable en une seule porte logique

— $S = \overline{a + b \cdot c}$: réalisable en une seule porte logique

— $S = \overline{a + \overline{b}}$: réalisable en deux portes logiques (il faut un inverseur pour obtenir $\overline{b}$

— $S = \overline{a \cdot b + c \cdot d}$: réalisable en une seule porte logique

— $S = a \oplus b$ réalisable à partir de 3 portes logiques (dont deux inverseurs)

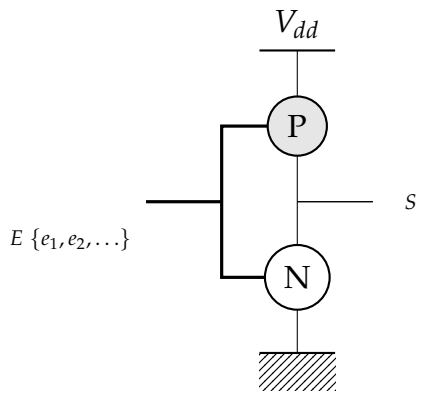


FIGURE 5.11: Logique MOS complémentaire

3. On peut démontrer qu'il suffit de disposer d'un réseau pMOS « dual » du réseau nMOS, c'est-à-dire où les assemblages en parallèle et en série sont échangés pour obtenir ce comportement, cependant, dans certains cas, ce n'est pas la solution optimale

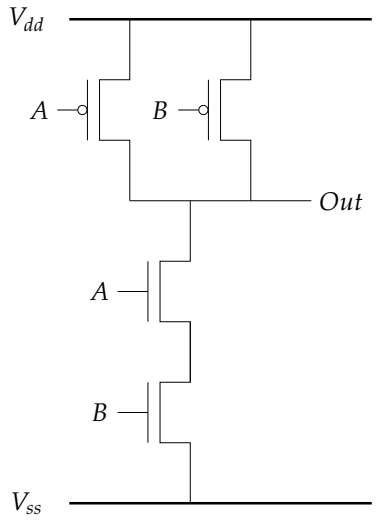


FIGURE 5.12: La porte CMOS « non-et à deux entrées »

4. Dans la pratique, seules des fonctions ayant 2 à 6 entrées peuvent être « raisonnablement » construites ainsi, au-delà, notre modèle « interrupteur presque parfait » n'est plus valable.

5.3 *Performances de la logique CMOS*

5.3.1 *Un peu d'histoire*

En 1965, Gordon Moore, co-fondateur d'Intel a émis la constatation que la complexité (le nombre de transistors) des circuits à semiconducteurs doublait tous les deux ans à coûts constants (coûts de fabrication). Cette constatation s'accompagnait de la prédiction de la poursuite de cette tendance. Cette constatation, dénommée depuis "Loi de Moore", est devenue par la suite « auto-prédictive » , en ce sens qu'elle est devenue un facteur d'ajustement économique pour l'ensemble de l'industrie micro-électronique, du fabriquant d'équipements aux fondeurs de circuits. Plus précisément, les grands acteurs du domaine, ont ajusté leurs efforts de recherche et développement ainsi que leurs investissements dans les usines de fabrication pour « tenir » le rythme imposé par cette prédiction.

En termes économiques, nous sommes dans une situation de « Technology push » : les fabricants de semi-conducteurs tablent sur une offre constamment renouvelée pour créer de nouveaux besoins. Les

analyses simplistes de l'innovation dans le numérique se réfèrent souvent au « Market pull » (l'innovation tirée par le marché), en oubliant totalement l'impact de cette industrie entièrement tournée vers la fourniture de composants sans cesse moins chers et plus performants.

La loi de Moore a depuis été extrapolée à d'autres aspects des composants, liés à leurs performances, on trouvera dans la littérature des choses comme :

— La fréquence de fonctionnement des microprocesseurs double tous les ... mois

— La consommation des circuits est divisée par deux tous les ... mois

Les fabricants tentent d'améliorer en permanence les « performances » de leur technologie, mais qu'entend-on par performances ?

5.3.2 *Critères de performance*

Le premier critère, conforme à l'énoncé de Gordon Moore est la **surface** du circuit intégré. En effet, comme pour tout procédé de fabrication le rendement de fabrication (le nombre de circuits opérationnels divisé par le nombre de circuits fabriqués) n'est pas de 100%. Le procédé de fabrication des circuits intégrés s'accompagne de défauts (transistors non opérationnels), dont la probabilité d'apparition croît avec la surface du circuit réalisé. On peut donc tenter de réduire les coûts de fabrication, en réduisant la taille des transistors.

Le deuxième critère est la **vitesse** de fonctionnement. Plus le temps de propagation des portes logique est faible, plus on peut augmenter la fréquence d'horloge des processeurs et donc plus on augmente la puissance de calcul. Nous verrons plus loin, qu'ici aussi, la diminution de la taille des transistors est un critère prépondérant.

Le troisième critère est la **consommation**. Nous avons déjà vu que le calcul implique une consommation d'énergie. On désire évidemment minimiser cette consommation comme, par exemple, dans le cas des objets connectés, mais aussi évacuer la chaleur dissipée comme, par exemple, pour les serveurs dans le « cloud ». Cette fois encore, la taille des transistors a un impact direct sur l'énergie consommée par une porte logique.

5.3.3 *Le transistor MOS : une vision plus précise*

Pour tenter de modéliser les performances de la logique CMOS, il nous faut revenir à un modèle un peu plus précis du transistor MOS, en intégrant, entre autre, ses caractéristiques géométriques.

La figure 5.13 présente la vue en coupe d'un transistor nMOS. Dans une technologie donnée, le concepteur de portes logiques peut utiliser deux paramètres géométriques pour « tailler » le transistor.

Intuitivement, le canal du transistor situé entre la source et le drain (les deux zones dopées N^+) peut être assimilé à un barreau conducteur : plus le barreau est court (longueur L du canal) et plus

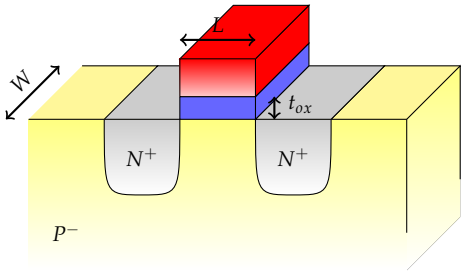


FIGURE 5.13: Un transistor nMOS de largeur W et de longueur L

le barreau est large (largeur W du canal) moins le barreau est résistif. Enfin, la commande du transistor (la grille) étant séparée du canal par l'oxyde, sera d'autant plus efficace que l'épaisseur de cet oxyde sera faible.

Tout en conservant le modèle interrupteur des paragraphes précédents, nous pouvons étudier un modèle simple du courant maximal traversant le transistor utilisé pour réaliser des portes en logique CMOS, lorsqu'il est « passant » :

$$I_{DS_{max}} = K_n \cdot (V_{dd} - V_{TN})^2$$

avec

$$K_n = \frac{1}{2}\mu_{0N} \cdot C'_{ox}\frac{W_N}{L_N}$$

Dans ces équations, nous observons :

— Que le courant croît comme le carré de la différence entre la tension d'alimentation V_{dd} et la tension de seuil V_{TN} du transistor

— Que le courant croît avec la mobilité μ_{0N} des porteurs dans le matériau

— Que le courant croît avec la capacité surfacique de l'oxyde de grille C'_{ox}, c'est-a-dire avec la diminution de l'épaisseur de cet oxyde.

— Enfin, que les géométries W et L ont bien l'impact prédit précédemment.

De plus, nous avons vu que la grille des transistors est utilisée pour connecter les entrées des portes en logique CMOS. Cette grille, séparée du canal par l'oxyde de grille, forme une capacité (condensateur) parasite avec ce canal. Cette capacité est tout simplement proportionnelle à la surface du canal. Nous pouvons modéliser cette capacité par la formule :

$$C_{ox} = C'_{ox}W_N \cdot L_N$$

5.3.4 Temps de calcul d'une porte en logique CMOS

Nous avons déjà vu dans le chapitre 1 que l'on pouvait s'appuyer sur le temps de propagation des portes logiques pour estimer la vitesse de fonctionnement maximale d'opérateurs de traitements. Nous allons, ici, élaborer un modèle de ce temps de propagation en nous appuyant sur notre connaissance des caractéristiques du transistor MOS et en nous limitant à l'exemple simple de l'inverseur CMOS.

La sortie d'un inverseur est physiquement reliée par des fils de connexions métalliques et à des entrées de portes logiques situées en aval. Ces fils de connexions forment des capacités parasites par rapport au substrat de silicium du circuit. Nous avons vu, d'autre part, que les entrées des portes logiques sont connectées à des grilles de transistors qui forment elles aussi des capacités parasites.

Toutes ces capacités peuvent être assimilées à une unique capacité parasite C_{par}, connectée à la sortie de notre inverseur.

À chaque transition montante de l'inverseur le transistor pMOS fournit le courant nécessaire à la charge de cette capacité parasite V_{ss} à V_{dd}.

À chaque transition descendante de l'inverseur le transistor nMOS fournit le courant nécessaire à la décharge de cette capacité parasite V_{dd} à V_{ss}.

Le temps de calcul d'une porte logique est directement lié au temps nécessaire à ces charges et décharges.

Considérons la figure 5.14. Le signal d'entrée E passe brutalement de V_{ss} à V_{dd}. Au début de la transition, le transistor pMOS est passant et le transistor nMOS est bloqué : la sortie S vaut V_{dd}. Pendant la transition, le transistor pMOS se bloque et le transistor nMOS devient passant. Ce dernier fourni le courant pour décharger la capacité C_{par} et établir la sortie S à V_{ss}.

Nous pouvons exploiter la relation entre le courant et la tension aux bornes de la capacité parasite :

$$I_{C_{par}} = C_{par}\frac{dV_{C_{par}}}{dt}$$

Le courant de décharge peut être approximativement considéré comme constant et identique à celui du transistor nMOS :

$$I_{C_{par}} \approx I_{DSmax} = K_n \cdot (V_{dd} - V_{tn})^2$$

Le temps de calcul de l'inverseur, c'est-à-dire le temps de décharge de V_{dd} à V_{ss}, est donc :

$$t_{calc} = C_{par}\frac{\Delta V}{I_{DSmax}} = C_{par}\frac{V_{dd}}{K_n \cdot (V_{dd} - V_{tn})^2}$$

Nous disposons donc d'un modèle simple du temps de calcul de l'inverseur dans lequel apparait l'influence de la capacité parasite chargée (C_{par}), de la tension d'alimentation (V_{dd}) de la logique[5], et enfin de la technologie et des dimensions du transistor (K_n et V_{tn}).

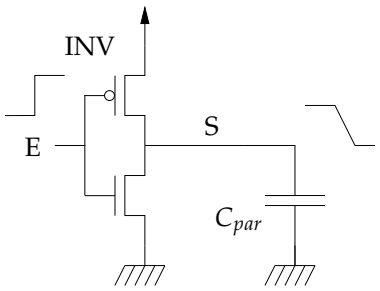


FIGURE 5.14: Décharge d'une capacité parasite pour une transition descendante de l'inverseur CMOS

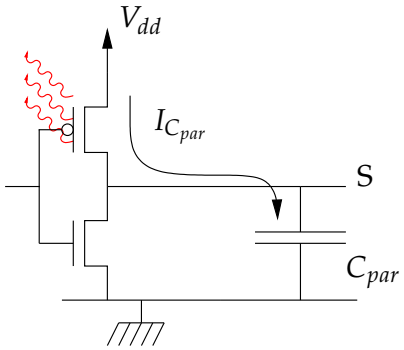


FIGURE 5.15: Consommation pour une transition montante de l'inverseur

5. On pourrait être tenté d'augmenter la tension d'alimentation de la logique pour augmenter sa vitesse de calcul. C'est d'ailleurs ce qui est fait pour les PC dédiés aux jeux pour lesquels on pratique l' « Overclocking ». Cependant cela nécessite de maîtriser parfaitement le refroidissement des processeurs ce que nous comprendrons dans le paragraphe traitant de la consommation

5.3.5 *Consommation d'une porte en logique CMOS*

Pendant une transition montante de la sortie de l'inverseur (figure 5.15), la source d'alimentation fournit le courant traversant le transistor pMOS servant à la charge de la capacité parasite. L'énergie fournie par l'alimentation constante V_{dd} est :

$$E_{Vdd} = C_{par}\int_0^{V_{dd}} V_{dd}\mathrm{d}V_s = C_{par}V_{dd}^2$$

L'énergie stockée dans la capacité C_{par} est :

$$E_{C_{par}} = C_{par}\int_0^{V_{dd}} V_s\mathrm{d}V_s = C_{par}\frac{V_{dd}^2}{2}$$

La moitié de l'énergie fournie par l'alimentation a été dissipée par effet Joule dans le transistor pMOS, l'autre moitié a été stockée dans la capacité parasite.

Pendant une transition descendante de la sortie de l'inverseur (figure 5.16), l'énergie stockée dans la capacité parasite est restituée et dissipée par effet Joule dans le transistor nMOS.

En moyenne, on peut considérer que $C_{par}\frac{V_{dd}^2}{2}$ est dissipée (ou consommée) à chaque transition de la sortie de la porte logique.

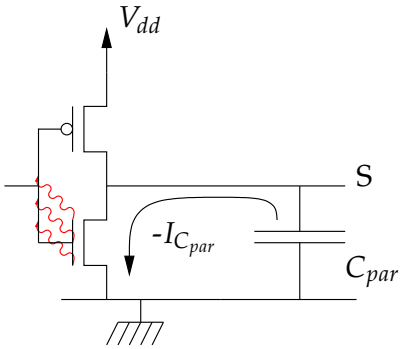


FIGURE 5.16: Consommation pour une transition descendante de l'inverseur

5.3.6 *Consommation d'un circuit intégré en logique synchrone CMOS*

Nous pouvons extrapoler le calcul précédent à un circuit complet. Pour cela, nous nous intéresserons plutôt à la puissance consommée qu'à l'énergie. Nous considérons un circuit réalisé en logique synchrone :

Soit F_h la fréquence de fonctionnement du circuit.

À chaque cycle d'horloge, un nouveau calcul sera effectué, certaines portes logiques changeront d'état à l'occasion de ce calcul, d'autres ne changeront pas d'état. Pour un circuit donné on peut estimer la probabilité de transition des portes à chaque cycle d'horloge. L'ordre de grandeur communément admis de cette probabilité est d'environ $0,3$ [6], bien que cela soit très variable d'un circuit à l'autre.

Soit T_{act} la probabilité de transition des portes du circuit à chaque front d'horloge.

Soit enfin C_{total} la capacité parasite totale du circuit.

La puissance consommée (donc dissipée) par le circuit est égale au produit du nombre de transitions par seconde des portes par l'énergie consommée par ces mêmes portes. D'où la formule évaluant la consommation du circuit :

$$P_{circuit} \approx T_{act} F_h C_{total} V_{dd}^2$$

Nous constatons donc que la consommation du circuit est directement proportionnelle à sa fréquence de fonctionnement et croît comme le carré de sa tension d'alimentation [7].

6. Une probabilité de 1 correspond à une horloge de fréquence $F_h/2$. Une horloge de fréquence $F_h/4$ correspond à une probabilité de $0,5$. Quand il y a calcul, c'est-à-dire traitement de l'information, cette probabilité est forcément inférieure...

7. En ce qui concerne l'« Overclocking » déjà évoqué dans une note précédente, nous percevons la difficulté : si nous augmentons la tension d'alimentation pour augmenter la fréquence de fonctionnement, la consommation doit subir à la fois une augmentation due à la fréquence et une augmentation due à la tension d'alimentation. C'est ce qui explique l'usage de systèmes de refroidissement monstrueux dans les PC pour joueurs...

5.4 *Évolutions technologiques et lois de Moore*

Nous avons déjà évoqué les lois de Moore dans ce chapitre, avec notamment cette course à la réduction des dimensions des transistors.

Les fabricants de circuits intégrés travaillent en utilisant la notion de « génération technologique » : ils visent une **réduction d'un facteur 2 des surfaces des circuits intégrés** à chaque nouvelle génération. Cela nécessite à chaque génération des investissements colossaux d'abord en recherche et développement et ensuite en construction de nouvelles usines.

Une génération technologique est caractérisée par la longueur de grille L caractéristique des transistors utilisés. Nous parlons ainsi de technologies 90 nm, 65 nm, 40 nm, 28 nm...

Nous allons utiliser les formules de performance calculées précédemment pour évaluer l'impact de la réduction des dimensions sur ces performances.

Attention, les hypothèses que nous allons faire sont simplificatrices et ne correspondent pas à la réalité d'aujourd'hui. Cependant

elles permettent de bien comprendre les conséquences qualitatives de ces évolutions.

Pour réduire la surface d'un facteur 2, nous utilisons un facteur $\beta = \sqrt{2}$ de la façon suivante :

— division par β de la largeur W et la longueur L des transistors [8]

— division par β de l'épaisseur d'oxyde de grille T_{OX} [9]

— division par β de la tension d'alimentation V_{dd} des circuits [10]

— division par β de la tension de seuil V_T des transistors [11]

8. **d'où la réduction en surface d'un facteur β^2**
9. pour mieux piloter le courant du transistor
10. pour ne pas "claquer" les transistors à plus faible épaisseur de grille
11. pour garder de la marge par rapport à V_{dd}

5.4.1 *Évolution du temps de calcul en fonction de β*

En reprenant l'expression du temps de calcul de l'inverseur et en exprimant chaque paramètre en fonction de β, nous avons :

$$\begin{aligned} t_{calc}(\beta) &= C_{par}(\beta)\frac{V_{dd}/\beta}{K_n(\beta)\cdot((V_{dd}-V_{tn})/\beta)^2} \\ &= C_{par}(\beta)\frac{\beta\cdot V_{dd}}{K_n(\beta)\cdot(V_{dd}-V_{tn})^2} \end{aligned}$$

Nous devons donc évaluer l'évolution de la capacité parasite C_{par}. Nous supposons qu'elle provient essentiellement des capacités de grilles des transistors, donc interviennent les dimensions et l'épaisseur de l'oxyde :

$$C_{par}(\beta) = (W/\beta)(L/\beta)(\beta C'_{ox}) = \frac{C_{par}}{\beta}$$

Nous devons de même évaluer l'évolution du paramètre K_n :

$$K_n(\beta) = \frac{1}{2}\mu_{0N}\cdot(\beta C'_{ox})\frac{W_N/\beta}{L_N/\beta} = K_n\cdot\beta$$

D'où l'expression finale du temps de calcul :

$$t_{calc}(\beta) = (C_{par}/\beta)\frac{\beta\cdot V_{dd}}{K_n\cdot\beta\cdot(V_{dd}-V_{tn})^2} = \frac{t_{calc}}{\beta}$$

La vitesse de fonctionnement de la logique (donc la fréquence maximale pour un circuit synchrone) est donc multipliée par β.

5.4.2 *Évolution de la consommation en fonction de β*

Reprenons l'expression de l'énergie fournie à la porte par l'alimentation :

$$E_{porte}(\beta) = (\frac{C_{par}}{\beta})(\frac{V_{dd}}{\beta})^2 = \frac{E_{porte}}{\beta^3}$$

L'énergie consommée par chaque porte au moment des transitions est donc diminuée d'un facteur β^3 !

5.4.3 *Changer de génération technologique pour diminuer les coûts et la consommation*

Une première stratégie d'exploitation du changement de technologie consiste à **ne pas exploiter le gain théorique en vitesse** pour

se concentrer essentiellement sur les coûts. La nouvelle génération technologique permet de créer des circuits de surface β^2 plus petits (d'où le gain en coût), ayant les mêmes performances que la génération précédente et ayant une consommation globale qui diminue. En effet, si nous reprenons l'expression de la puissance consommée globalement par le circuit, nous avons :

$$P_{circuit}(\beta) = T_{act} \cdot F_h \frac{E_{circuit}}{\beta^3} = \frac{P_{circuit}}{\beta^3}$$

Cette stratégie est particulièrement intéressante dans les systèmes embarqués. Elle peut être utilisée pour gérer des transitions de produits « haut de gamme » à faible diffusion vers des produits « bas de gamme » à forte diffusion. L'exemple le plus frappant est celui des smartphones pour lesquels les produits à faible coût intègrent petit à petit les fonctionnalités des produits plus chers.

On peut aussi mettre à profit la diminution de consommation pour créer de nouveaux produits avec de nouvelles utilisations, l'exemple majeur étant ici l'apparition des objets connectés.

5.4.4 *Changer de génération technologique pour augmenter la puissance de calcul à coût constant*

L'idée est d'une part d'utiliser le gain potentiel en vitesse (un facteur β) tout en gardant un coût de fabrication des circuits. La surface du circuit ne diminuant pas nous multiplions donc par β^2 le nombre de transistors du circuit, donc sa complexité.

La consommation du circuit ne change pas [12] :

$$P_{circuit}(\beta) \approx T_{act}(\beta F_h)(\beta^2(C_{total}/\beta))(V_{dd}/\beta)^2 = P_{circuit}$$

12. La capacité des transistors est divisée par β mais le nombre de transistors est multiplié par β^2

Nous pouvons donc réaliser un circuit β fois plus rapide, β^2 fois plus complexe, ayant la même consommation que le circuit de la génération précédente et au même coût de fabrication. Cette stratégie est particulièrement intéressante pour les processeurs de stations de travail « Haut de gamme » ou des serveurs de calcul, puisque la puissance totale de calcul de ces processeurs bénéficie à la fois de la montée en fréquence et de l'augmentation de parallélisme.

5.4.5 *Changer de génération technologique : la fin de la loi de Moore ?*

Dans la pratique, les vitesses de fonctionnement maximales stagnent depuis le début des années 2000 (3 à 5 Ghz). Les processeurs sont limités en vitesse à cause de la résistance et des capacités des interconnexions entre portes dont l'impact est de plus en plus important. Pour compenser cela, les architectes utilisent une stratégie de parallélisme (processeurs multi-coeurs). Mais ils sont cette fois limités par la consommation, ainsi les performances des processeurs pour serveurs "progressent" mais à un rythme relativement modéré.

On ne peut diminuer sans cesse la tension d'alimentation sans s'éloigner du modèle d'interrupteur idéal. Les circuits ont des courants de fuite de moins en moins négligeables ce qui est un problème pour les applications embarquées avec des modes de veille.

Les technologues doivent jongler avec des procédés de fabrication de plus en plus complexes pour continuer à suivre la loi de Moore. À chaque nouvelle génération il faut inventer de nouvelles structures physiques dont on est de moins en moins sûr de la faisabilité. Il faut d'autre part créer des équipements de fabrication aptes à gérer des dimensions physiques de plus en plus petites, une fois de plus ces équipements sont à la limite de ce que l'on sait réaliser.

On a plusieurs fois prédit la fin de la loi de Moore pour des raisons "scientifiques" dues à la physique du transistor mais il semble, en 2015, que le plus grave problème soit économique, en tout cas la controverse est clairement présente.

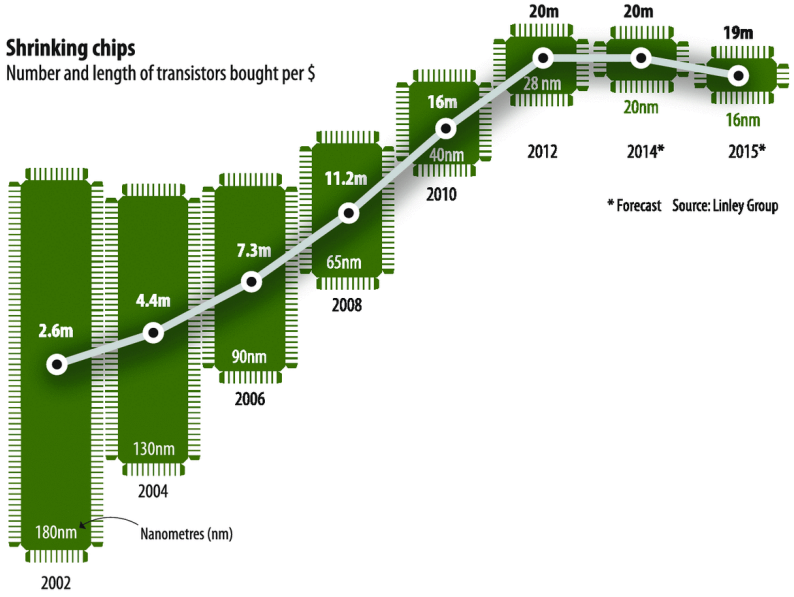


FIGURE 5.17: La fin de la loi de Moore ?

La figure 5.17 présente une étude de l'évolution des coûts des technologies les plus récentes et à venir. La figure indique, pour chaque génération technologique, l'année d'apparition, la longueur de grille [13] des transistors et le nombre de transistors achetables par dollar. Cette étude tend à montrer que les coûts n'évoluent plus depuis 2012. C'est-à-dire que le seul gain des nouvelles technologies concerne les performances en vitesse et en consommation. Ce genre de projection peut avoir un impact sérieux sur les décisions d'investissement dans la micro-électronique et en conséquence sur l'évolution de l'industrie du numérique au sens large.

Cette projection est contestée par des acteurs majeurs comme la

13. Attention, la dénomination des technologies par référence à la longueur L de grille des transistors n'a plus beaucoup de sens depuis la technologie 65 nm. Les fondeurs utilisent une longueur équivalente correspondant à la racine carrée du rapport de surface des circuits en passant d'une technologie à la suivante…

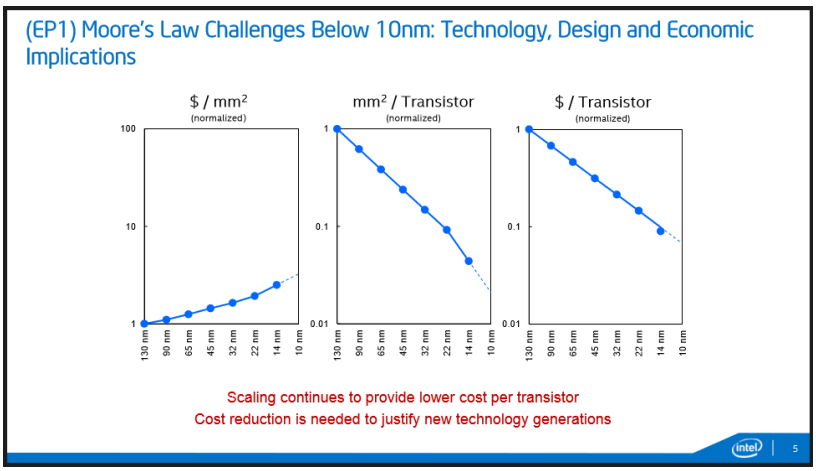


FIGURE 5.18: Pour Intel, la loi de Moore continue

société Intel. Ainsi le document présenté en figure 5.18, extrait d'une présentation de la société Intel, faite en novembre 2014 et à destination d'investisseurs, présente une diminution des coûts se poursuivant au moins sur les 4 à 5 années à venir.

Toujours est-il que nous ne connaissons pas, à l'heure actuelle, les solutions technologiques qui permettrons de poursuivre cette évolution dans les dix années à venir.

Annexes

A
SystemVerilog

SystemVerilog est un langage de description de matériel ou HDL (*Hardware Description Language*). Il permet de décrire des circuits de deux façons :

Description d'une structure : on décrit de quoi est composé un circuit en termes de blocs et comment ces blocs sont reliés entre eux. Les blocs peuvent aussi à leur tour contenir des blocs plus petits jusqu'aux portes élémentaires.

Description d'un comportement : plutôt que de décrire un circuit en termes de sous-blocs, on peut décrire son comportement (ce qu'il fait, sa fonctionnalité).

La syntaxe de SystemVerilog est proche de celle de C, cependant il faut faire attention au fait que :

— Un programme en C ou en Java est une succession d'ordres (instructions) qu'un processeur exécute successivement les uns après les autres, alors qu'une description SystemVerilog décrit le comportement d'un circuit composé de plusieurs entités fonctionnant simultanément en parallèle

— SystemVerilog sert à décrire des opérations entre des signaux (au sens physique du terme) et non pas des variables (au sens informatique du terme)

A.1 Quelques règles générales de syntaxe

L'extension des fichiers SystemVerilog est `.sv`. Ce sont de simples fichiers textes, vous pouvez donc utiliser l'éditeur de texte de votre choix.

Les commentaires sont comme en C :

— `//` pour commenter une ligne

— `/* .... */` pour commenter un bloc

Le `;` joue le rôle de séparateur.

Les blocs sont délimités par les mots clés `begin` et `end`. Ils remplacent les accolades {} utilisées en C ou en Java.

Il ne faut pas mettre de séparateur `;` après les mots-clés `end`, `endmodule` et `endcase`.

A.2 Les modules

En SystemVerilog, les blocs sont appelés modules. Toute description comportementale ou structurelle doit **impérativement** être faite dans un module.

La déclaration d'un module se fait de la façon suivante :

```
module nom_du_module ( declaration_des_entrees_sorties );
   contenu_du_module
endmodule
```

CODE A.1: Déclaration d'un module

Les signaux d'entrée/sortie sont déclarés dans l'entête de déclaration du module. Il s'agit d'une liste de déclarations individuelles séparées par des **virgules**.

Chaque déclaration individuelle doit comprendre :

— La déclaration du type d'entrée-sortie : `input` pour une entrée, `output` pour une sortie.

— Le type de signal : nous utiliseront exclusivement le type `logic`.

— La largeur éventuelle du signal : la notation `[j:i]` indique un vecteur de `(j-i+1)` bits indexés de `i` à `j`.

Attention : `i` est le bit de poids faible et `j` est le bit de poids fort.

```
(
  input  logic clk,      // Un signal entrant de largeur 1 bit appelé "clk"
  output logic aaa,      // Un signal sortant de largeur 1 bit appelé "aaa"
  input  logic [7:0] d   // Un bus entrant de largeur 8 bits appelé "d"
);
```

CODE A.2: Exemple de déclaration d'entrées/sorties

Les modules contiennent des signaux internes (qui ne sont ni des entrées ni des sorties). Ces signaux internes peuvent être déclarés de façon similaire aux entrées/sorties.

```
...
logic mon_signal ;              // Un signal de largeur 1 bit appelé "mon_signal"
logic [3:0] mon_autre_signal ;  // Un bus de largeur 4 bits appelé "mon_autre_signal"
...
```

CODE A.3: Exemple de déclaration de signaux internes

A.3 Contenu des modules : Structures et instanciation de modules

La figure A.1 présente l'exemple d'un module `M_a` contenant deux instances `Inst1` et `Inst2` d'un même module `M_b`.

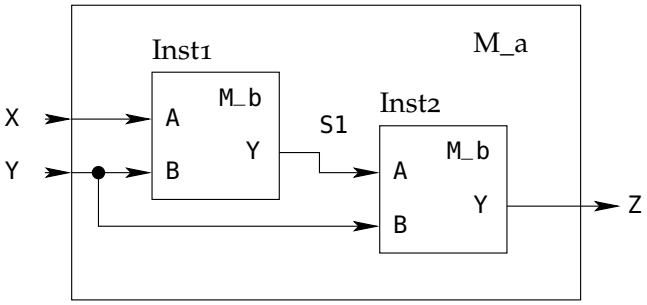


FIGURE A.1: Exemple d'instanciation de sous-modules

Remarquez le signal nommé `S1` qui connecte la sortie `Y` d'un sous-module à l'entrée `B` d'un autre sous-module.

Le code correspondant à cette description est le suivant :

CODE A.4: Exemple d'instanciation de sous-modules

```
module M_a( input logic X,
            input logic Y,
            output logic Z );

  logic S1; // Déclaration du signal S1 interne à M_a

  M_b Inst1(.A(X),.B(Y),.Y(S1)) ; // Instance Inst1 de  M_b
  M_b Inst2(.A(S1),.B(Y),.Y(Z)) ; // Instance Inst2 de  M_b
endmodule
```

— La déclaration d'un signal interne à un module ce fait en précisant :
  — Le type de signal ; pour nous exclusivement **`logic`**
  — La largeur éventuelle du signal : la notation `[i:j]` indique un vecteur de `i-j+1` bits indexés de `j` à `i`
  — Le nom du signal
— L'instanciation d'un sous-module ce fait en précisant :
  — Le nom du sous-module
  — Le nom de l'instance particulière du sous-module
  — Entre parenthèse, une liste de connexions des entrées/sorties.
    La syntaxe pour chaque Entrée/Sortie est :
    `.nom_de_l'entree_sortie_du_sous_module ( nom_du_signal_du_module )`

Exemples de connexions :

```
logic [1:0]C ;
logic [1:0]D ;

xxx inst_xxx(.E(C),...    ); // Le bus C est relié à l'entrée E de xxx.
                             // Ils ont le même nombre de bits.
xxx inst_xxx(.F(C[0]),...  ); // Le bit 0 de C est connecté à l'E/S F de xxx.
                             // (F doit donc normalement avoir une largeur de 1 bit)
xxx inst_xxx(.G({C,D}),... ); // Les signaux C et D sont regroupés en un vecteur
                             // de largeur 4 bits, connecté à l'E/S G de xxx.
                             // (G doit donc ici normalement avoir une largeur de 4 bits)
...
```

A.4 *Contenu des modules : Description d'un comportement combinatoire*

A.4.1 *Logique combinatoire*

On décrit un comportement en spécifiant les valeurs affectées aux sorties ou à des signaux internes d'un module. Pour l'instant nous nous limitons à la description de comportements de type calcul combinatoire.

La syntaxe générale utilisée est la suivante :

```
always@(*)
     nom_du_signal <= fonction_combinatoire ;
```

CODE A.5: Forme générale pour décrire de la logique combinatoire

Il faut interpréter cette syntaxe comme :

— Le signal `nom_du_signal` est la sortie de la fonction `fonction_combinatoire`.

— À chaque fois qu'une entrée de cette fonction est modifiée, le signal `nom_du_signal` est remis à jour.

Voici quelques exemples de description de calculs combinatoires.

```
logic a, b, c ;          // Déclaration de 3 signaux internes sur 1 bit
logic [1:0] Z ;          // Déclaration d'un signal interne sur 2 bits

always@(*) a <= b & c ;  // Il y a une porte logique AND dont les entrées
                         // sont b et c et dont la sortie est a

always@(*) a <= c ;      // le signal a est identique au signal c (on a relié les fils)

                         // Le signale a est défini par une fonction tabulée de Z
always@(*)
 case (Z)
  2'b00   : a <= 1'b1;
  2'b01   : a <= 1'b0;
  2'b10   : a <= 1'b0;
  2'b11   : a <= 1'b1;
endcase
                         // Le signal a est défini par une
always@(*)               // fonction complexe de Z, b c et d...
 case (Z)
  2'b00   : a <= b ;     // cas où Z vaut 0
  2'b01   : a <= c & d ; // cas où Z vaut 1
  default : a <= d ;     // tous les autres cas
endcase
```

CODE A.6: Exemple de comportement combinatoire

Comme vous le remarquez dans cet exemple, nous vous proposons deux modes de calcul des fonctions combinatoires :

— une expression algébrique directe,

— une expression tabulée grâce à l'usage de la syntaxe case.

Les expressions algébriques peuvent utiliser les opérateurs suivants :

Fonction	Symbole
Négation	~
Et	&
Ou	|
Xor	^

TABLE A.1: Opérateurs logiques

La syntaxe que nous utiliserons pour le `case` est celle du bloc de code A.7.
Attention : lorsque l'on utilise l'expression `case` il faut veiller à définir complètement la table de vérité, soit explicitement, soit implicitement via l'expression `default`.

Les valeurs indiquées dans le `case` sont des constantes.

La base `10` est utilisée par défaut. Vous pouvez aussi préciser explicitement la base utilisée (décimal : `'d`, hexadécimal : `'h` ou binaire : `'b`) ainsi que la taille de la constante de la façon suivante :

```
case (signal_de_selection)
     valeur_1 : mon_signal <= expression_algebrique_1 ;
     valeur_2 : mon_signal <= expression_algebrique_2 ;
     ...
     default :  mon_signal <= expression_algebrique_default ;
endcase
```

CODE A.7: Syntaxe du `case`

```
25     : ...   // constante (par défaut sur 32 bits) valant 25 (en base 10)
8'd25  : ...   // constante sur 8 bits valant 25 (en base 10)
3'b101 : ...   // constante sur 3 bits valant 101 en base 2 (donc 5 en base 10)
4'ha   : ...   // constante sur 4 bits valant A en hexadécimal (donc 10 en base 10)
```

CODE A.8: Exemple de constantes

A.4.2 Manipulation des vecteurs (bus)

Les entrées/sorties ainsi que les signaux internes peuvent être des vecteurs. Il est possible de sélectionner un élément ou une partie d'un vecteur. Les exemples du bloc A.9 vous montrent quelques cas d'utilisation.

```
logic [7:0] A;          // Déclaration d'un vecteur de dimension 8
logic [3:0] B, C, D;    // Déclaration de 3 vecteurs de dimension 4
logic Z;                // Élement sur 1 bit

always@(*) B <= A[7:4]; // Le quartet B est identique à la moitié de poids fort de l'octet A

always@(*) Z <= A[3];   // Z est identique au bit n° 3 de A

always@(*) A <= {C,D};  // A est la concaténation de C et D
                        // équivalent à A[7:4] <= C et
                        //              A[3:0] <= D
```

CODE A.9: Exemple d'opérations sur les bus

A.4.3 Arithmétique entière et expressions

En SystemVerilog, les vecteurs de bits de type `logic[i:0]` sont implicitement interprétés comme des entiers non signés de `(i+1)` bits. Vous pouvez donc utiliser les opérateurs arithmétiques standards : `+` et `-` ...

SystemVerilog gère automatiquement les opérations d'extension de taille et de troncature lorsque les opérandes sont de tailles différentes :

```
logic [3:0] a ;
logic [4:0] b ;
logic c ;
logic y[5:0] ;
logic z[3:0] ;
logic t[2:0] ;

always@(*) b <= 5'b01110 ;        // b est un entier de 5 bits valant 14.
always@(*) a <= 2'b11 ;           // la valeur constante 3 est affectée à l'entier a
                                  // codé sur 4 bits (a vaut 4'b0011)
always@(*) c <= 3 ;               // c est un entier de 1 bit,
                                  // on lui affecte la valeur 3 (2'b11),
                                  // après troncature c vaut la valeur 1
                                  // (bit de poids faible de la constante)
always@(*) y <= a + b + c ;       // y est codé sur un nombre de bits suffisant
                                  // quelles que soient les valeurs de a, b et c
                                  // (ici y = 18)  010010 = 000011 + 001110 + 000011

always@(*) z <= a + b + c ;       // z prends les 4 bits de poids faible du résultat
                                  //  (z = 2) 0010 = 0011 + 1110 + 0011

always@(*) {t,z} <= a + b + c ;   // z prends les 4 bits de poids faible du résultat : z=2
                                  // voir la remarque 1
                                  // t prends les 3 bits de poids fort du résultat : t=1
always@(*) z <= {4{c}} ^ a ;      // Le bus z est identique au bus a si c est égal a 0,
                                  // sinon le bus z est le complémentaire du bus a.
                                  // voir la remarque 2
```

CODE A.10: Exemple d'opérations arithmétiques

1. Remarquez l'usage des accolades pour concaténer deux bus.
2. Remarquez l'usage des accolades pour générer un bus de 4 bits identiques.

A.4.4 Nombres signés

Par défaut, les vecteurs de bits déclaré comme `logic[i:0]` sont considérés comme des nombres non signés. Les opérations arithmétiques ainsi que les opérations de comparaison arithmétique les interpréteront donc comme des nombres non signés.

Pour pouvoir faire de l'arithmétique sur des nombres signés il faut utiliser le mot-clé `signed` au moment de la déclaration.

```
logic signed [3:0] A,B;
logic c;
always@(*) A <= 4'b1111;
always@(*) B <= 4'b0000;
always@(*) c <= (A>B);  // c vaut 0 car A est interprété comme un nombre signé
                        // ie. A vaut -1 et B vaut 0
```

CODE A.11: Exemple d'utilisation des nombres signés

A.4.5 *Expressions booléennes*

En SystemVerilog, un signal de type `logic` est équivalent à un booléen :

— vrai est représenté par un `1`

— faux est représenté par un `0`

Les opérateurs utilisés dans les expressions booléennes sont semblables à ceux du langage C

L'opérateur ternaire `<condition> ? <valeur si vrai> : <valeur si faux>` peut être utilisé de la même façon qu'en langage C ou Java.

Fonction	Symbole
et	&&
ou	||
égalité	==
négation logique	!
non égalité	!=
inférieur	<
...	...

TABLE A.2: Opérateurs logiques booléens

```
logic a, c ,d ;
logic [7:0] b ;

always@(*) a <= (b == 8) ;          // a prend la valeur 1'b1 lorque b est égal à 8

always@(*) a <= (!(b == 8)) && c ; // a prend la valeur 1'b1 lorsque b
                                    // est différent de 8 et que c est égal à 1'b1
always@(*) a <= (b == 8) ? c : d ; // a prend la valeur c si b est égal à 8,
```

CODE A.12: Exemple d'utilisation des opérateurs logiques

A.5 Contenu des modules : Description d'un comportement séquentiel synchrone

La syntaxe `always@(*)` permet de décrire des fonctions combinatoires dans le langage SystemVerilog. Dans le jargon des HDL, nous appelons cela un processus.

Une description HDL d'un matériel est donc un ensemble de processus s'exécutant parallèlement. Pour décrire des bascules `D` et des registres ou plus généralement tout élément synchrone à une horloge, nous devons disposer d'une syntaxe permettant de déclencher l'évaluation d'un processus sur des évènements correspondant à la transition d'un signal. La transition particulière qui nous intéresse ici est le front de l'horloge.

Pour cela, en SystemVerilog, nous pouvons utiliser le processus `always@(posedge clk)`.

La syntaxe générale pour décrire de la logique séquentielle synchrone est :

Code A.13: Forme générale pour décrire de la logique séquentielle

```
always@( evenement_declenchant )
   action_a_realiser ;
```

A.5.1 Une bascule D

Le code suivant décrit une bascule D sensible au front montant d'une horloge `CLK` :

Code A.14: La description d'une bascule D

```
logic D,Q ;
...
always@( posedge clk )
  Q <= D ;
```

Vous pouvez combiner la bascule D avec de la logique combinatoire à son entrée. Le code suivant décrit une bascule D dont l'entrée est le calcul du AND entre deux signaux `A` et`B`.

```
logic Q, A, B;

always@( posedge clk )
  Q <= A & B;
```

Vous pouvez décrire un groupe de bascules D (un registre) en utilisant la notation vectorielle vue précédemment. Le code suivant décrit un registre 8 bits.

```
logic [7:0] Q, D;

always@( posedge clk )
  Q <= D;
```

Pour définir une bascule fonctionnant sur front descendant il suffit de remplacer le mot-clé **posedge** par **negedge**.

A.5.2 Une bascule D avec initialisation asynchrone

Le processus **always@(posedge** clk**)** doit aussi pouvoir être déclenché par le signal d'initialisation ou de remise à zéro. De plus ce signal doit être prioritaire sur l'horloge.

La syntaxe suivante permet de décrire une bascule D avec une initialisation asynchrone active dès le passage à l'état bas d'un signal `init` et dont l'action est de mettre à **0** la sortie de la bascule.

CODE A.15: Bascule D avec remise à zéro asynchrone

```
logic  Q, D, init;

always@( posedge clk or negedge init )
  if ( ~init )
    Q <= 0;
  else
    Q <= D;
```

Vous pouvez interpréter cette syntaxe de la façon suivante :

— S'il y a un évènement (front montant) sur l'horloge, ou un évènement sur `init` (passage à l'état bas) alors il faut réévaluer le contenu de la bascule `D`.

— S'il faut réévaluer et que `init` **==** **0** on passe la sortie de la bascule à **0**.

— S'il faut réévaluer et que `init` **==** **1** alors c'est le front montant d'horloge qui a déclenché le processus et donc `Q` reçoit `D`.

Vous êtes évidemment libres de choisir :

— l'état actif de l'initialisation :
  — **posedge** pour un signal actif à l'état haut **(1)**, associé au test **if(**init**)**
  — **negedge** pour un signal actif à l'état bas **(0)**, associé au test **if(~**init**)**

— la valeur initiale : **0** ou **1**

A.6 Généralisation des processus `always`

Jusqu'à maintenant, vous n'avez décrit que des processus (`always`) n'ayant qu'une seule action à réaliser. Il est possible de regrouper plusieurs actions d'un processus dans un bloc.

La syntaxe est la suivante :

```
begin
  premiere_action ;
  deuxieme_action ;
  ...
end
```

```
logic [7:0] Q;
logic [4:0] D;
...
always@( posedge clk )
begin
  Q[7:4] <= Q[3:0];
  Q[3:0] <= D;
end
```

Code A.16: Exemple de bloc dans un processus `always`

Les mots clés `begin` et `end` jouent le rôle des accolades ({ }) dans des langages comme le C ou le Java.

Les affectations d'un même bloc sont effectuées en parallèle. Dans les deux blocs suivant sont donc équivalents.

```
always@( posedge clk )
begin
  b <= a;
  c <= b;
end

always@( posedge clk )
begin
  c <= b;
  b <= a;
end
```

A.7 Codage des états des automates finis

A.7.1 Définition de types énumérés pour décrire des états

SystemVerilog supporte la définition de signaux de type énuméré de la façon suivante :

```
// sig est un signal pouvant prendre 7 valeurs :
//              "SWAIT", "S1", "S2", "S3", "S4", "S5" ou "S6".
enum logic[2:0] {SWAIT, S1, S2, S3, S4, S5, S6} sig;
   // On peut alors écrire les choses suivantes :
   always@(*)
      sig <= S1;

enum logic {V1, V2} sig1, sig2; // sig1 et sig2 sont deux signaux
                                //pouvant prendre 2 valeurs "V1" ou "V2".
// On peut alors écrire :
   always@(*)
      if (sig1 == sig2)
         sig1 <= V2;
      else
         ...
```

CODE A.17: Exemple de déclaration d'un type énuméré

— Par défaut, les valeurs énumérées correspondent aux codes `0`, `1`, `2`, `3` ... en partant de la première valeur.

— La taille nécessaire au codage des différentes valeurs est définie dans la définition du type.

A.7.2 Codage de l'évolution d'un automate fini.

L'usage de la syntaxe `case` permet de traduire facilement la table d'évolution des états. Pour cela vous pouvez regrouper dans un même vecteur l'ensemble des signaux d'entrée et l'état courant de manière à définir la table sous la forme :

```
 always@(*)
   case ({etat_courant, entree_1, entree_2})
     {WAIT,1'b0, 1'b1} : next_state <= S1;
     {S1,  1'b1, 1'b1} : next_state <= S2;
     ...
     default : next_state <= current_state;
   endcase
```

CODE A.18: Exemple de transition d'états

A.7.3 Extension de l'usage du `case` : le `casez`

Il peut parfois être long et fastidieux de donner explicitement tous les cas. Lorsque certains signaux ne sont pas utiles dans les équations de transition, il est possible de les représenter par un "`?`". Dans ce cas le mot-clé `case` doit être remplacé par `casez`.

Code A.19: Exemple d'utilisation du `casez`

```
always@(*)
  casez ({etat_courant, entree_1, entree_2})
    {WAIT,1'b0, 1'b1} : next_state <= S1;
    {S1,  1'b1, 1'b?} : next_state <= S2;
    ...
    default : next_state <= current_state;
  endcase
```

Dans l'exemple précédent, on passe de l'état `S1` à l'état `S2` indépendamment de la valeur de `entree_2`.

B
Sujets de travaux dirigés

Ce chapitre, comprend les sujets des trois séances de travaux dirigés du module ELECINF102.

— *Le **TD1** nécessite la compréhension des chapitres 1 (page 5) et 2 (page 25) ainsi que des annexes A.1 (page 87) à A.5 (page 95).*

— *Le **TD2** nécessite la compréhension des chapitres 1 (page 5) à 2 (page 25) ainsi que des annexes A.1 (page 87) à A.5 (page 95).*

— *Le **TD3** nécessite la compression des chapitres 1 (page 5) à 3 (page 41) ainsi que de l'annexe A (page 87) en entier.*

B.1 TD1 : Logique séquentielle synchrone

B.1.1 Bascule D avec enable

— Nous voulons construire une bascule D munie d'un signal **reset**, **synchrone** actif à l'état **haut**

— Cette bascule sera munie d'un signal supplémentaire **enable**

— Si le signal **enable** vaut **1**, la bascule fonctionne normalement.

— Si le signal **enable** vaut **0**, la bascule est gelée, la sortie **Q** garde sa valeur même après un front montant de l'horloge.

1. Faites un schéma a base de portes simples et d'une bascule D.
2. Ecrivez le code SystemVerilog équivalent

B.1.2 Parallélisation

— Nous recevons, de manière synchrone, une séquence de données **data_in** codées sur 1 bit.

— À chaque cycle d'horloge un signal **en_in** indique si le bit **data_in** est une donnée valide.

— Nous voulons transformer cette séquence en séquences de données **data_out** de 4 bits, accompagnée d'un signal **en_out** indiquant si la donnée **data_out** est valide.

1. Réalisez un chronogramme montrant un exemple de transmission.
2. Déterminez le ou les signaux suplémentaires nécessaires au fonctionnement du dispositif.
3. Déterminez les éléments logiques nécessaires à la réalisation du dispositif.
4. Faites un schéma.
5. Ecrivez le code SystemVerilog équivalent.

B.1.3 Sérialisation

— Nous recevons, de manière synchrone, une séquence de données **data_in** codées sur 4 bits.

— À chaque cycle d'horloge un signal **en_in** indique si la donnée **data_in** est une donnée valide.

— Transformer cette séquence en séquences de données **data_out** de 1 bit, accompagnée d'un signal **en_out** indiquant si la donnée **data_out** est valide.

1. Déterminez les conditions de bon fonctionnement du dispositif (oubliées dans l'énoncé).
2. Réalisez un chronogramme montrant un exemple de transmission.
3. Déterminez le ou les signaux suplémentaires nécessaires au fonctionnement du dispositif.
4. Déterminez les éléments logiques nécessaires à la réalisation du dispositif.
5. Faites un schéma.
6. Ecrivez le code SystemVerilog équivalent.

B.1.4 Détecteur de front montant

L'horloge, signal global prédéfini, est l'unique signal pouvant être utilisé comme entrée de synchronisation sur une bascule. Il arrive cependant de devoir détecter le passage de 0 à 1 ou de 1 à 0 d'un signal synchrone d'un cycle d'horloge au suivant.

A chaque étape, construire un schéma, puis un code SystemVerilog.

1. Comment peut-on faire pour détecter d'un cycle à l'autre de l'horloge, qu'un signal est passé de l'état 0 à l'état 1 ?
2. Si le signal entrant n'est pas synchrone de l'horloge, comment garantir que le signal généré dure exactement un cycle ?
3. Comment modifier le montage pour détecter indifféremment un changement d'état de 0 à 1 ou de 1 à 0 ?
4. À quoi peut servir ce genre de dispositif ?

B.1.5 Filtrage à moyenne glissante

Nous recevons une séquence de données codées sur 8 bits.

1. Réalisez un filtre calculant la somme des 4 derniers échantillons reçus.
2. Réalisez un filtre calculant la moyenne des 4 derniers échantillons reçus.
3. Optimisez la structure pour limiter le nombre d'additionneurs nécessaires à moins de 3.

B.2 TD2 : Représentation des nombres, opérateurs de calcul séquentiels et combinatoires

B.2.1 Un multiplieur de nombres représentés en virgule fixe

Nous disposons d'un module de calcul **Multiplieur** calculant le produit **P** de deux nombres **A** et **B**. Les nombres **A** et **B** sont deux entiers **signés** codés sur **8 bits** ;

Question 1 : Déterminez les valeurs minimales et maximales pouvant être atteintes par **A** et **B**.

Question 2 : Déterminez les valeurs minimales et maximales pouvant être atteintes par P. En déduire le nombre de bits nécessaires au codage de **P**.

Nous voulons utiliser ce module pour traiter une représentation en virgule fixe de nombres réels. Pour cela les nombres **A** et **B** seront interprétés de la façon suivante :

— Les 4 bits de poids fort représentent la partie entière du nombre.

— Les 4 bits de poids faible représentent la partie fractionnaire du nombre.

— Nous nommerons cette représentation `FIX4.4`

Question 3 : Déterminez sous la forme de fraction rationnelle les valeurs maximales et minimales pouvant être atteintes par **A** et **B**.

Question 4 : Faut-il modifier le multiplieur pour calculer la multiplication ?

Question 5 : Où se situe la virgule dans le résultat **P** ?

Nous voulons coder le résultat en représentation `FIX4.4`, cela nécessite de tronquer la sortie **P**.

Question 6 : Quels bits de P devons nous conserver ?

Nous voulons mettre en place un dispositif de saturation si le nombre P n'est pas représentable en `FIX4.4`.

Question 7 : Déterminez une condition simple permettant de détecter un dépassement de capacité sur un nombre positif, et permettant de choisir la valeur maximale atteignable dans ce cas.

Question 8 : Déterminez une condition simple permettant de détecter un dépassement de capacité sur un nombre négatif, et permettant de choisir la valeur minimale dans ce cas.

B.2.2 Architecture de multiplieur combinatoire

Nous voulons construire un module de calcul **Multiplieur** calculant le produit **P** de deux nombres **A** et **B**. Les nombres **A** et **B** sont deux entiers **non signés** codés sur **4 bits**. Le multiolieur sera une structure de traitement combinatoire.

Question 1 : En utisant les expressions analytiques $\mathbf{A} = \sum_{i=0}^{3} A_i.2^i$ et $\mathbf{B} = \sum_{i=0}^{3} B_i.2^i$ exprimez le résultat P sous la forme de deux sommes imbriquées.

Question 2 : Posez le calcul de la multiplication de **A** par **B** comme vous l'avez appris à l'école primaire en base 10 et en utilisant les produits partiels exprimés dans la question **1**

Question 3 : Quelle fonction logique simple permet de calculer chaque produit partiel ?

Nous supposons que nous disposons des portes logiques suivantes :

— additioneur 1 bit : $\{cout, s\} <= a + b + Cin$

— fonctions logiques élémentataires à 2 entrées (et, ou, inverseur,...)

Question 4 : Faites le schéma d'une architecture de traitement combinatoire réalisant la multiplication en utilisant ces portes logiques.

Question 5 : En supposant que le temps de calcul de chacune des portes logiques est de 1 (dans une unité arbitraire), calculez le temps de calcul du multiplieur.

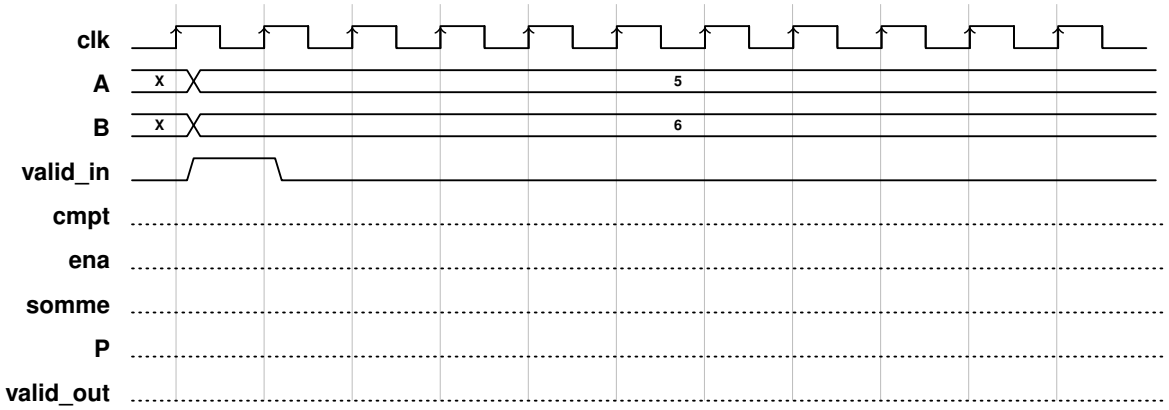
Question 6 : Généralisez le précédent résultat pour un multiplieur NxN.

B.2.3 Architecture de multiplieur séquentiel

Nous voulons maintenant calculer la multiplication de manière itérative et séquentielle. Un code SystemVerilog est proposé (voir B.1)

Question 1 : Etudiez le code proposé, et tentez d'en comprendre le principe.

Question 2 : Complétez le chronogramme suivant.



Question 3 : Quel est le rôle des signaux **valid_in** et **valid_out** ?

Question 4 : Faites un schéma de la structure de calcul utilisée.

Question 5 : Déterminez approximativement le temps de calcul de la structure pour un multiplieur NxN.

Code B.1: Code SystemVerilog d'un multiplieur itératif

```
module multiplieur2( input  logic [3:0] A,
                     input  logic [3:0] B,
                     output logic [7:0] P ,
                     input  logic valid_in,
                     output logic valid_out,
                     input  logic clk) ;

logic [1:0] cmpt ;
logic       ena;
logic [4:0] somme ;

always @(*) somme <= P[7:4] + (B[cmpt] ? A : 4'd0) ;

always @(posedge clk)
    if(valid_in) begin
        P    <= 0 ;
        cmpt <= 0 ;
        ena  <= 1 ;
        valid_out <= 1'b0 ;
    end else begin
        P <= {somme,P[3:1]} ;
        valid_out <= 1'b0;
        if (ena) begin
           cmpt <= cmpt + 1 ;
           if(cmpt == 3) begin
              ena       <= 1'b0 ;
              valid_out <= 1'b1;
           end
        end
    end
endmodule
```

B.3 TD3 : Automates matériels

B.3.1 Qu'est-ce qu'un bus de communication ?

Lorsque, au sein d'un système complexe, plusieurs dispositifs électroniques doivent communiquer entre eux on peut imaginer de relier chaque élément à tous les autres. Cette situation, illustrée par la figure B.1, est probablement la première qui vient à l'esprit. C'est aussi la plus puissante car elle permet un nombre très important de communications simultanées.

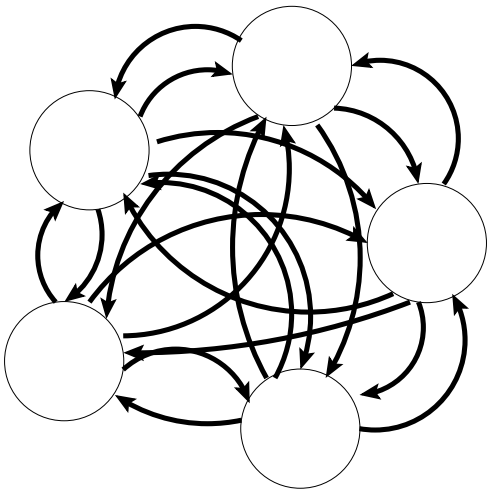


FIGURE B.1: Liaisons point à point

Malheureusement elle est aussi très coûteuse car le nombre de connexions nécessaires est très important. Il suffit d'imaginer pour s'en convaincre que les arcs du schéma ci-dessus véhiculent des informations codées sur 32 bits. En outre elle n'offre pas une grande flexibilité car il n'est pas possible d'ajouter des éléments à notre réseau (le nombre d'entrées et de sorties de chaque élément est fixé à la construction). Ce système n'est pas très *plug and play*. C'est dommage car le *plug and play* est justement très à la mode. Une autre solution, plus raisonnable et aussi plus courante, est le bus central comme illustré dans la figure B.2

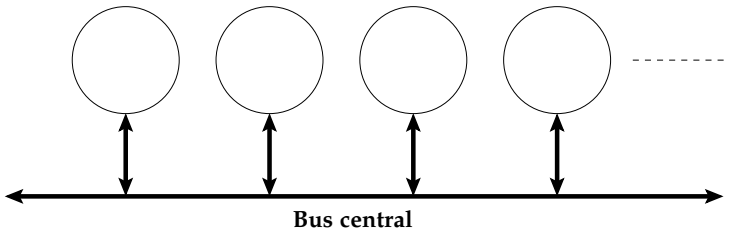


FIGURE B.2: Bus central

Les possibilités d'échanges sont limitées mais chaque élément peut tout de même communiquer avec n'importe quel autre et le nombre de connexions est considérablement réduit. Il est en outre théoriquement possible d'ajouter à l'infini de nouveaux éléments au système. La gestion d'une telle organisation des communications nous servira de thème tout au long de ce TD.

B.3.2 Le contrôleur de bus simple.

Nous nous proposons de concevoir un contrôleur de bus de communication. Le système au sein duquel notre contrôleur doit s'intégrer comporte un arbitre de bus et un nombre indéterminé mais potentiellement très grand de points d'accès au bus. Chaque point d'accès est composé d'un contrôleur et d'un client. La figure B.3 représente le système de communication complet :

L'arbitre est chargé de répartir la ressource de communication (le bus) entre les différents points d'accès. En effet, le système n'admet pas que plusieurs points d'accès émettent simultanément des informations

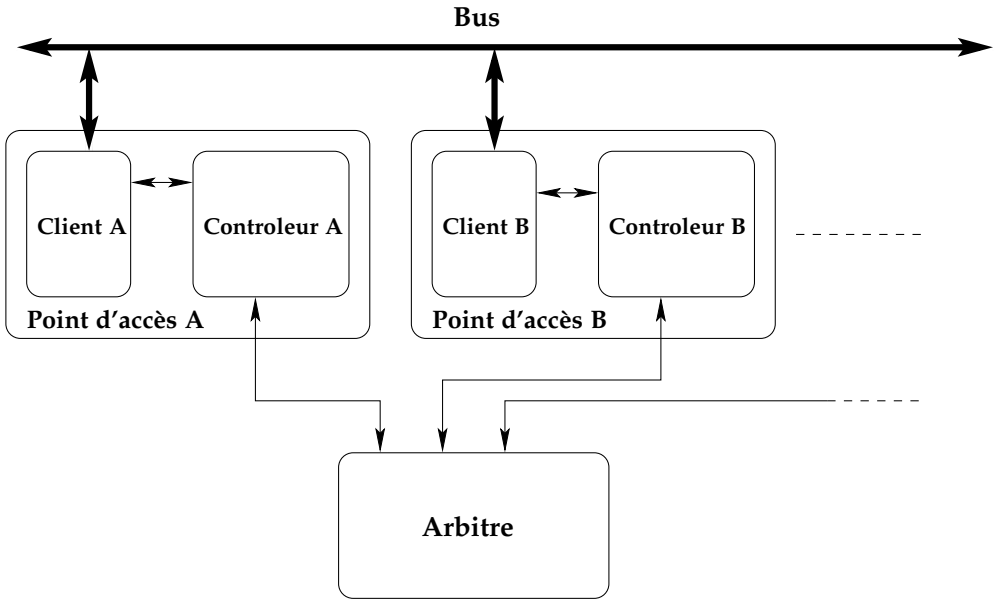


FIGURE B.3: Système de communication

sur le bus. Si cela se produisait il y aurait conflit et perte d'informations. La présence d'un arbitre est donc nécessaire. C'est lui qui autorise successivement les points d'accès à écrire sur le bus en leur attribuant un "jeton". Le point d'accès possesseur du jeton peut écrire sur le bus. Les autres ne peuvent que lire. Lorsque le point d'accès a terminé sa transaction il rend le jeton à l'arbitre qui peut alors l'attribuer à un autre point d'accès. L'absence de conflit est garantie par l'unicité du jeton.

Les clients sont les utilisateurs du bus. Lorsqu'un client désire écrire sur le bus il en informe son contrôleur associé et attend que celui-ci obtienne le jeton et lui donne le feu vert.

Les contrôleurs servent d'interface entre l'arbitre et leur client. C'est l'un de ces contrôleurs que nous allons concevoir. Ses entrées - sorties sont décrites dans le schéma illustré en figure B.4 et la table qui suit.

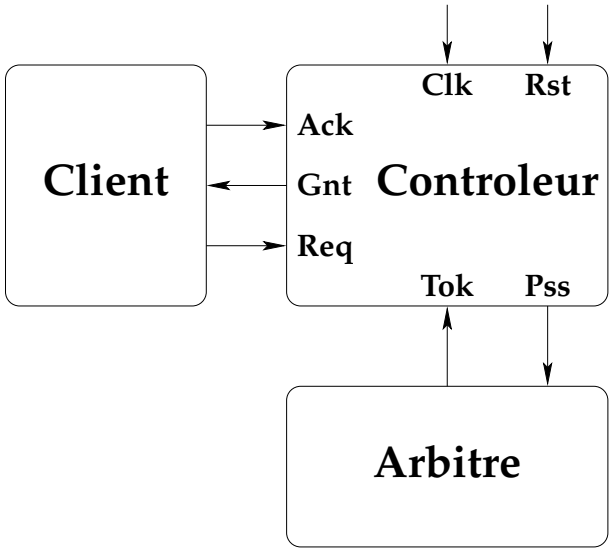


FIGURE B.4: Contrôleur de communication

Nom	Direction	Description
CLK	**Entrée**	Horloge pour la synchronisation du contrôleur.
RST	**Entrée**	Signal d'initialisation **asynchrone**, actif à '1'. Lorsque ce signal est à état haut ('1') le contrôleur est entièrement réinitialisé.
TOK	**Entrée**	Ce signal provient de l'arbitre et indique que le contrôleur peut disposer du bus. Il signifie donc que l'arbitre offre le jeton au contrôleur. Il n'est actif que pendant une période d'horloge. Si le contrôleur n'a pas besoin du jeton il le rend (voir le signal **PSS**). Sinon il le garde jusqu'à ce qu'il n'en ait plus l'utilité.
REQ	**Entrée**	Ce signal est émis par le client et indique que ce dernier demande à disposer du bus. Le client maintient ce signal jusqu'à la fin de sa transaction sur le bus. Il ne le relache que lorsqu'il n'a plus besoin du bus.
ACK	**Entrée**	Ce signal provient du client et indique que le client a pris le bus et commence sa transaction. Il n'est actif que pendant une période d'horloge.
PSS	**Sortie**	Ce signal est destiné à l'arbitre et l'informe que le contrôleur rend le bus, soit parce que l'arbitre le lui a proposé alors qu'il n'en a pas besoin, soit parce que la transaction du client est terminée. Il signifie donc que le contrôleur rend le jeton à l'arbitre qui pourra ensuite en disposer et l'attribuer à un autre contrôleur, voire au même. Il n'est actif que pendant une période d'horloge.
GNT	**Sortie**	Ce signal est destiné au client et l'informe qu'il peut disposer du bus. Il est maintenu tant que le client n'a pas répondu (par le signal **ACK**) qu'il a pris le bus.

Le graphe d'états et l'automate

Question 1 : Dessinez un chronogramme représentant une ou plusieurs transactions complètes entre un contrôleur, son client et l'arbitre.

Question 2 : Le contrôleur est un automate synchrone. Imaginez et dessinez son graphe.

Question 3 : Vérifiez la cohérence du graphe (complet, non contradictoire).

Question 4 : Ecrivez le code SystemVerilog de votre automate.

Une optimisation possible.

Les échanges entre l'arbitre et le contrôleur (signaux **TOK** et **PSS**) présentent l'inconvénient de ralentir inutilement les opérations et donc de gaspiller des cycles d'utilisation du bus. En effet, un cycle est perdu lorsqu'un contrôleur se voit proposer le jeton alors qu'il n'en a pas l'usage. Le chronogramme de la figure illustre B.5 ce phénomène :

TOKA et **TOKB** sont les signaux **TOK** destinés à deux contrôleurs, A et B. **PSSA** est le signal **PSS** émis par le contrôleur A et indiquant qu'il rend le jeton que l'arbitre vient de lui confier et dont il n'a pas l'usage. On voit que l'arbitre, lui aussi synchrone sur front montant de l'horloge, ne peut pas proposer immédiatement le jeton à un autre contrôleur.

Pour améliorer les performances du système nous voudrions obtenir le chronogramme illustré en figure B.6 :

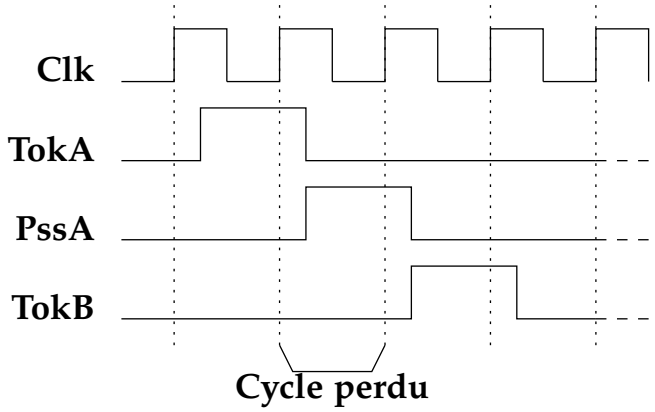


FIGURE B.5: Illustration de la perte d'un cycle

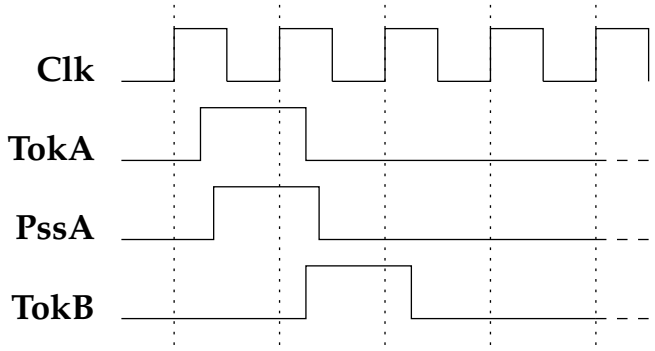


FIGURE B.6: Chronogramme optimisé

Question 5 : Proposez des modifications du contrôleur permettant d'obtenir ce nouveau comportement.
Question 6 : Décrivez l'automate en langage SystemVerilog.

B.3.3 Le problème de l'équité.

Le contrôleur équitable.

Le contrôleur que nous venons de concevoir n'est pas entièrement satisfaisant car il n'est pas équitable. En d'autres termes, il ne garantit pas qu'un client n'accaparera pas le bus au détriment des autres. Il ne garantit même pas qu'un client, après avoir obtenu l'accès au bus, l'utilisera effectivement puis le relâchera. Il est en effet possible qu'un client ne réponde jamais au signal **GNT** de son contrôleur (ce qu'il est censé faire à l'aide du signal **ACK**). Le système complet serait alors bloqué par un "mauvais" client qui monopolise une ressource dont il n'a pas l'usage. Pour remédier à cet inconvénient il faut à nouveau modifier le contrôleur.

Question 7 : Imaginez des solutions afin de rendre équitable le contrôleur optimisé du premier exercice.

L'arbitre équitable.

Pour obtenir que l'ensemble du système soit équitable, la modification du contrôleur seul ne suffit pas. L'arbitre doit, lui aussi, adopter un comportement particulier. **Question 8** : Pourquoi ? Donnez un exemple de comportement non équitable possible de l'arbitre et ses conséquences.

Question 9 : Imaginez et décrivez des comportements possibles de l'arbitre équitable.

C
Exemples de constructions synchrones

Ce chapitre contient des exemples d'utilisation de la logique séquentielles synchrone. Vous y trouverez des schémas ainsi que le code SystemVerilog correspondant.

C.1 Les bascules

C.1.1 Bascule D avec enable

Ici nous voulons construire une bascule D qui conserve son état si une condition d'activation n'est pas réalisée. Dans la réalisation suivante, cette condition est matérialisée par l'entrée **en** de notre module synchrone.

Comme le signal d'horloge a un rôle particulier, **il ne faut pas ajouter de logique sur l'entrée d'horloge**.

La solution qu'il faut mettre en œuvre est l'ajout d'un multiplexeur commandé par l'entrée **en** et qui permet, si **en** est à l'état logique **0** de rediriger la sortie de la bascule vers son entrée et ainsi la conserver au cycle suivant. Si **en** est à l'état logique **1**, alors le signal d'entrée est dirigé vers l'entrée de la bascule pour capturer une nouvelle valeur au cycle suivant.

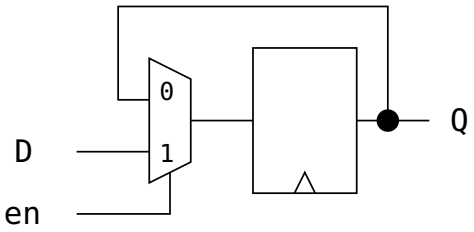


FIGURE C.1: Schéma d'une bascule D avec entrée d'activation

Le code SystemVerilog correspondant est donné en C.1. Il exprime le fait qu'à chaque front de l'horloge on ne modifie la sortie **Q** que si **en** est vrai. Exprimer la condition duale (le **else**) n'est pas obligatoire car dans ce cas, la sortie **Q** conserve bien son état.

CODE C.1: Description SystemVerilog d'une bascule D avec entrée d'activation

```
module dff_en ( input  clk,
                input  en,
                input  D,
                output logic Q );

 always@(posedge clk)
    if (en)
      Q <= D;
   // Sinon Q garde sa valeur précédente

endmodule
```

C.2 Les compteurs

C.2.1 Compteur modulo 256

Ici nous voulons réaliser un compteur libre modulo 256 (qui est une puissance de 2).

Il suffit de reboucher la sortie d'un registre de 8 bits (car $256 = 2^8$) sur l'entrée en utilisant un incrémenteur (additionneur avec la seconde entrée forcée à 1).

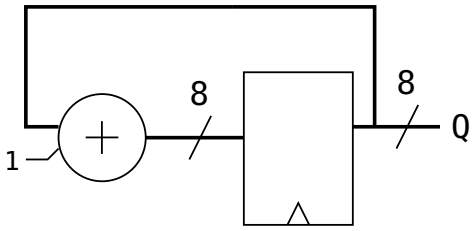


FIGURE C.2: Schéma d'un compteur modulo

Naturellement, comme on ne conserve que 8 bits en sortie de l'additionneur, nous obtiendront en sortie, cycliquement, toutes les valeurs entre 0 et 255.

i**Attention :** Avec cette construction, nous ne maitrisons pas l'état initial de la séquence. Il nous manque pour cela un mécanisme de remise à zéro (reset).

Le code SystemVerilog représentant ce comportement est donné en C.2.

CODE C.2: Description SystemVerilog d'un compteur modulo

```
module cpt_mod ( input  clk,
                 output logic [7:0] Q );

 always@(posedge clk)
      Q <= Q + 1;

endmodule
```

C.2.2 *Compteur modulo 256 avec remise à zéro*

Pour maitriser l'état initial du compteur il faut ajouter un mécanisme de remise à zéro (reset).

Grâce à un signal externe, on peut forcer l'état du registre du compteur et garantir qu'il démarre d'une valeur connue.

Remise à zéro asynchrone Un signal de remise à zéro asynchrone et un signal qui va agir directement sur la bascule. Son action est immédiate et n'est pas liée à l'état de l'horloge.

Les concepteurs des registres (bascules) prévoient donc une entrée supplémentaire pour cet effet.

Dans le schéma C.3 nous avons un signal de remise à zéro asynchrone actif sur niveau bas (schématisé par le cercle sur l'entrée **`nrst`**). La sortie du registre est mise à zéro **dès** que le signal **`nrst`** passe à **`0`**. Pour que la sortie change, il faut attendre le premier cycle d'horloge après le passage de **`nrst`** à **`1`**.

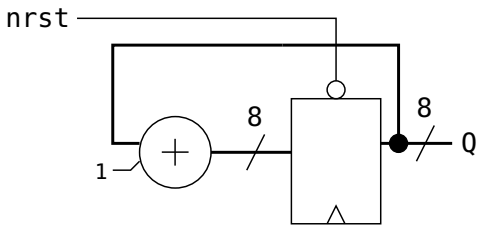


FIGURE C.3: Schéma d'un compteur modulo avec remise à zéro asynchrone

En SystemVerilog, pour exprimer une remise à zéro asynchrone, il faut faire apparaitre le signal de reset dans la liste de sensibilité du processus (**`always`**) pour indiquer qu'il doit être pris en compte dès qu'il devient actif.

```
module cpt_arst ( input  clk,
                  input  nrst,
                  output logic [7:0] Q );

 always@(posedge clk or negedge nrst)
   if (!nrst)
      Q <= '0;
   else
      Q <= Q + 1;

endmodule
```

CODE C.3: Description SystemVerilog d'un compteur modulo avec remise à zéro asynchrone

Remise à zéro synchrone La remise à zéro peut aussi être provoquée par un signal "normal" venant d'un autre bloc de logique synchrone. Dans ce cas, on peut agit directement sur l'entrée du registre. L'effet de ce signal de remise à zéro ne se produit alors qu'au front d'horloge.

La figure C.4 montre comment on peut implémenter une remise à zéro synchrone. Si le signal **nrst** passe à **0**, alors la sortie de la porte ET (en réalité 8 portes en parallèle) passe à zéro quelle que soit la sortie de l'aditionneur. Au front d'horloge suivant, la sortie du registre passera donc à zéro.

On voit qu'il faut garantir que le signal **nrst** reste à **0** durant au moins un cycle d'horloge. C'est forcement le cas si ce signal provient lui-même d'un bloc de logique synchrone.

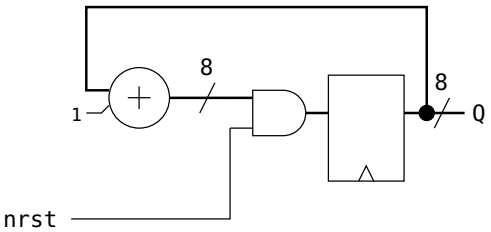


FIGURE C.4: Schéma d'un compteur modulo avec remise à zéro synchrone

Le code SystemVerilog diffère peu de la version précédente. Seul le fait que l'entrée **nrst** n'apparait pas dans la liste de sensibilité du processus ce qui fait que son état n'est pris en compte qu'au front de l'horloge.

CODE C.4: Description SystemVerilog d'un compteur modulo avec remise à zéro synchrone

```
module cpt_srt ( input  clk,
                 input  nrst,
                 output logic [7:0] Q ); // La sortie

 always@(posedge clk)
   if (!nrst)
      Q <= '0;
   else
      Q <= Q + 1;

endmodule
```

C.2.3 *Compteur/Décompteur avec entrée d'activation*

Ici nous construisons un compteur/décompteur contrôlé par le signal **sv** qui contrôle le sens d'évolution du compteur.

Un signal de remise à zéro synchrone **nrst** permet de forcer l'état initial et un signal d'activation **en** permet de contrôler son avancement.

Le code SystemVerilog C.5 décrit ce comportement. Notez que les actions des signaux de contrôle ont les priorités suivantes :

1. **nrst** : la remise à zéro doit être la plus prioritaire,
2. **en** : on évolue ou pas,
3. **up** : vers le haut ou le bas.

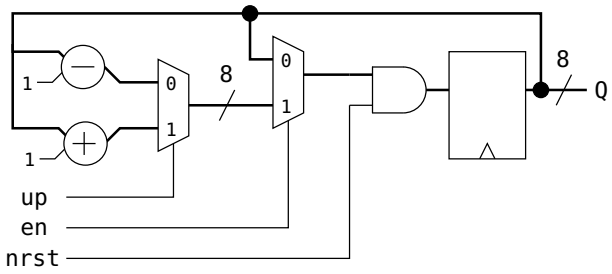


FIGURE C.5: Schéma d'un compteur/décompteur

Cette même priorité se retrouve sur le schéma C.5. Plus le signal est prioritaire, plus son action est proche du registre.

CODE C.5: Description SystemVerilog d'un compteur/décompteur

```
module cpt_updown ( input  clk,
                    input  nrst,
                    input  en,
                    input  up,
                    output logic [7:0] Q );

 always@(posedge clk)
   if (!nrst)
      Q <= '0;
   else
   if (en)
    begin
      if (up)
        Q <= Q + 1;
      else
        Q <= Q - 1;
    end
    // sinon on ne change pas d'état

endmodule
```

C.2.4 *Compteur jusqu'à 13*

Ici nous voulons que le compteur ne dépasse pas une valeur arbitraire, qui n'est pas uns puissance de deux, 13 par exemple.

. . .puis on re-part à zéro

Pour faire un compteur modulo une valeur arbitraire, qui n'est pas une puissance de deux il faut ajouter de la logique combinatoire pour comparer la valeur que registre à la valeur maximale désirée.

La sortie de ce comparateur agit alors comme un signal de remise à zéro synchrone. C'est ce qui est représenté sur le schéma C.6. Notez que la sortie du comparateur est combinée au signal de remise à zéro externe, lui aussi synchrone.

Ce comportement est décrit par le code SystemVerilog C.6. [1]

1. Ici le registre fait encore 8 bits de large mais comme la sortie ne dépasse jamais la valeur 13, nous aurions pu se suffire de 4 bits.

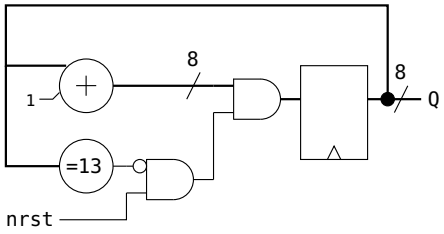


FIGURE C.6: Schéma d'un compteur modulo 13

CODE C.6: Description SystemVerilog d'un compteur modulo 13

```
module cpt_13_zero ( input  clk,
                     input  nrst,
                     output logic [7:0] Q );

 always@(posedge clk)
   if (!nrst)
      Q <= '0;
   else
     if(Q!=13)
      Q <= Q + 1;
     else
      Q <= 0;

endmodule
```

. . .et on s'arrête

Si nous voulons une séquence unique qui s'arrête à une valeur arbitraire il faut utilise la sortie du comparateur comme entrée d'activation du compteur. Dès que la sortie du compteur atteint la valeur max désirée, il est forcé à conserver sa valeur jusqu'à la prochaine remise à zéro.

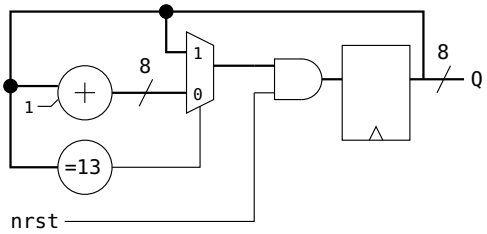


FIGURE C.7: Schéma d'un compteur qui s'arrête à 13

```
module cpt_13 (  input  clk,
                 input  nrst,
                 output logic [7:0] Q );

 always@(posedge clk)
   if (!nrst)
      Q <= '0;
   else
     if(Q!=13)
      Q <= Q + 1;
      // Sinon on reste à 13
endmodule
```

CODE C.7: Description SystemVerilog d'un compteur qui s'arrête à 13

C.3 *Détecteur de fronts montants*

Un détecteur de front est un module synchrone qui produit une impulsion dont a durée est exactement une période d'horloge quand l'état d'un signal passe de **0** à **1**.

Il ne faut pas utiliser le signal extérieur comme horloge ! I faut observer son état à chaque front de l'horloge et comparer deux valeurs consécutives de son état.

Dans le schéma C.8, nous voulons détecter les fronts du signal **key**. Pour cela, nous capturons sa valeur (nous l'échantillonnons) grâce à une première bascule. La seconde bascule permet de conserver l'état précédemment échantillonné.

Nous avons donc :

— `key_s[0]` le dernier état de **key**,

— `key_s[1]` l'état précédent de **key**.

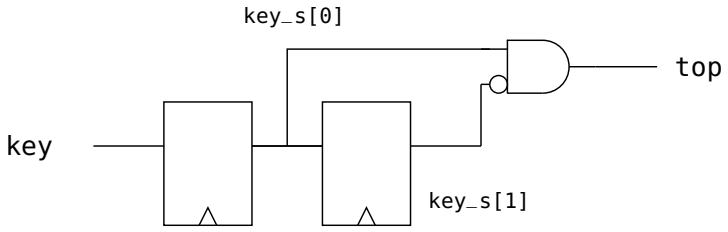


FIGURE C.8: Schéma d'un détecteur de fronts montant

La porte Et permet de comparer ces deux états et la sortie **top** passera à un si l'état est à **1** et l'état précédent est à **0**. Comme nous comparons des états (sortant de bascules) nous avons aussi la garantie que cette impulsion aura exactement une durée de 1 cycle de l'horloge.

Le code SystemVerilog C.8 contient deux processus (**always**). Le premier donnant le comportement du registre à décalage composé des deux bascules. Le second, la comparaison combinatoire des états.

```
module front( input  clk,
              input  key,
              output logic top
              );

   // on sauvegarde l'état de key
   // dans un registre à décallage
   logic [1:0]key_s;

   always@(posedge clk)
   begin
     key_s[0] <= key;
     key_s[1] <= key_s[0];
   end

   // On a un front montant si l'état actuel (0) est à 1
   // et que l'état précédent (1) est encore à 0
   always@(*)
     top <= key_s[0] & !key_s[1];
endmodule
```

CODE C.8: Description SystemVerilog d'un détecteur de fronts montant

C.3.1 *Utilisation d'un détecteur de fronts*

Ici nous utilisons le détecteur de fronts précédemment décrit pour contrôler un l'avancement d'un compteur modulo 256.

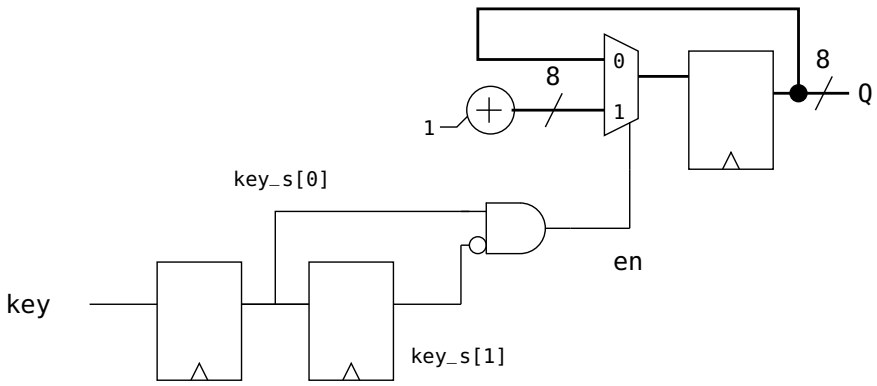


FIGURE C.9: Schéma d'un compteur contrôlé par une entrée extérieure

La sortie du détecteur de front est reliée à l'entrée d'activation (**en**) du compteur. Ainsi, à chaque fois que l'entrée **key** passe de zéro à un, le signal **en** passe à un durant exactement un cycle et le compteur s'incrémente une fois. Indépendamment de la durée de l'état haut du signal **key**.

```
module front( input  clk,
              input  key,
              output logic [7:0] Q
              );

   logic [1:0]key_s;
   logic en;

   always@(posedge clk)
   begin
     key_s[0] <= key;
     key_s[1] <= key_s[0];
   end

   always@(*)
     en <= key_s[0] & !key_s[1];

   always@(posedge clk)
     if(en)
        Q <= Q + 1;

endmodule
```

CODE C.9: Description SystemVerilog d'un compteur contrôlé par une entrée extérieure

Dans le code SystemVerilog, le signal interne **en** est généré par le détecteur de front et sert à contrôler le compteur.

Ce mécanisme nous permet de contrôler un système électronique synchrone sur une horloge rapide (imaginez un processeur fonctionnant à plusieurs dizaines/centaines de MHz) à partir d'évènements lents (un appui sur un bouton, le passage devant un capteur) tout en gardant un système synchrone.

C.4 *PWM : Pulse Width Modulation*

Le principe de fonctionnement d'un générateur de modulation largeur d'impulsion (PWM : Pulse Width Modulation*) est illustré par la figure C.10. Le rapport cyclique du signal* `pwm` *en sortie de ce module est contrôlé par la valeur du signal de consigne donné en entrée.*

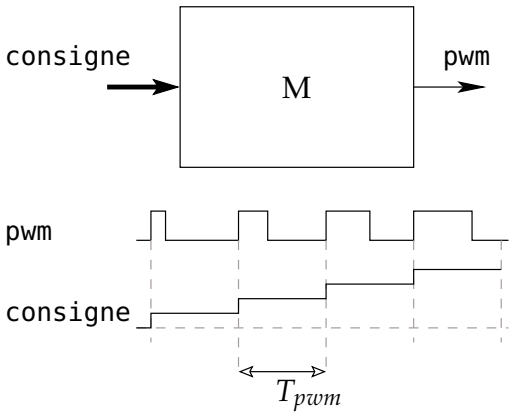


FIGURE C.10: Princime d'un générateur de PWM

Pour obtenir ce résultat, nous utilisons un compteur périodique dont la valeur se répète cycliquement (un compteur modulo). Pour générer la sortie, nous comparons la valeur de la consigne à la sortie du compteur. Si la sortie du compteur est inférieure à la consigne, alors la sortie est mise à `1` sinon elle est mise à `0`. La figure C.11 montre l'architecture d'un tel système.

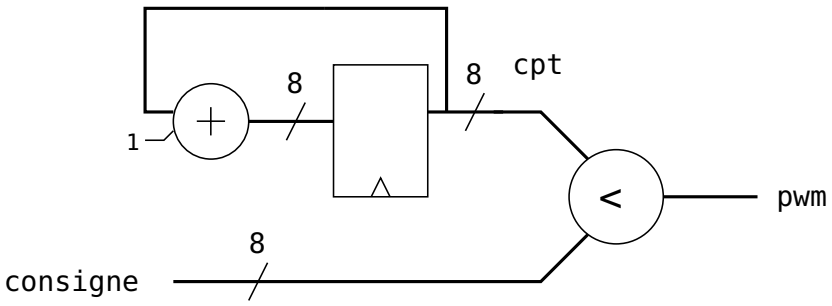


FIGURE C.11: Schéma d'un générateur de PWM

Le nombre de bits utiliser pour coder la consigne et la sortie du compteur nous donnera le nombre de pas avec lequel on peut régler la sortie de la PWM. Dans cet exemple, nous avons 256 niveaux et nous avons utilisé 8 bits pour coder la consigne et pour le registre du compteur. Aussi, l'horloge utilisée pour le compteur doit avoir une période T 256 fois plus petite que la période T_{pwm} désirée pour la PWM.

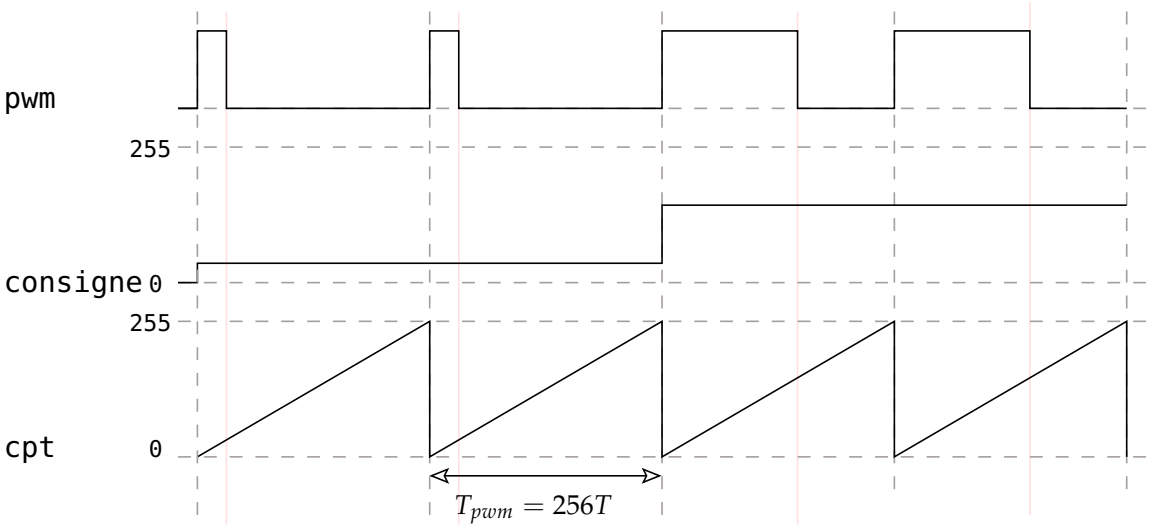


FIGURE C.12: Chronogramme de la PWM

Le chronogramme de la figure C.12 montre l'évolution du compteur ainsi que le signal généré en sortie pour deux valeurs de consigne. Souvenez-vous que la sortie du compteur est discrète et que si nous changions d'échelle de représentation nous observerions des marches.

CODE C.10: Description SystemVerilog d'un générateur de PWM

```
module pwm ( input  clk,
             input [7:0] consigne,
             output logic pwm );

    logic [7:0] cpt;
    // Un compteur modulo 256
    always@(posedge clk)
    begin
      cpt <= cpt + 1;
    end

    // la sortie vaut 1 si la valeur du compteur est
    // inférieure à la consigne
    always@(*)
      pwm <= (cpt < consigne);

endmodule
```

Le code SystemVerilog correspondant est donné dans la suite (code C.10).

Comme vous le constatez, la période de la PWM est liée à la période d'horloge du système.

Question : Comment feriez-vous pour ralentir la période de la PWM d'un facteur 1024 sans changer la fréquence de l'horloge de la logique synchrone ? Inspirez vous pour cela du détecteur de fronts qui contrôle un compteur dans les exemples précédents.