PROCESSEURS ET ARCHITECTURES NUMÉRIQUES: PROBLÈMES

EXEMPLES DE PROBLÈMES AVEC CORRIGÉS

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

TELECOM PARISTECH

Table des matières

1
Introduction

Ce support comprend des exemples de problèmes accompagnés d'une correction et de commentaires. Attention, dans la plupart des situations, il n'y pas une solution unique au problème, ainsi il peut exister des solutions plus élégantes, plus lourdes ou plus légères à celles proposées mais toutes exactes... Enfin, pour améliorer la compréhension, les chapitres du cours concernés par la résolution du problème sont référencés.

2
Analyse et mise au goût du jour d'un vieux vieux schéma

2.1 Énoncé du problème

Votre chef a retrouvé un vieux schéma montré en figure 2.1. Il désire en réaliser une nouvelle version. Pour cela vous devez utiliser le langage SystemVerilog. Les bascules du schéma sont supposées être mises à **`0`** au moment de la réinitialisation.

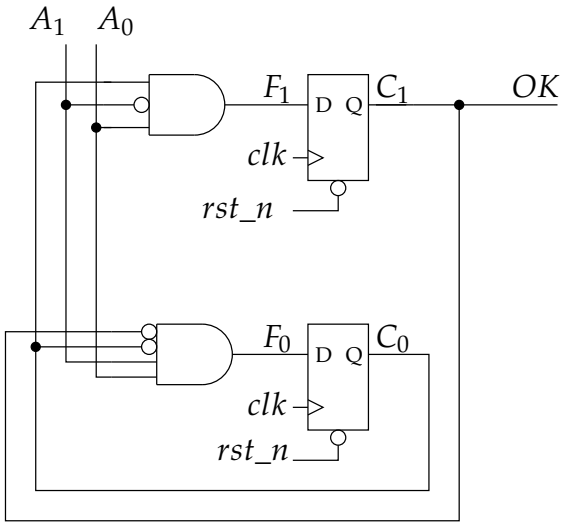


FIGURE 2.1: Vieux schéma

Question 1 : Écrivez (sans astuces...) le code SystemVerilog correspondant au schéma. Le code doit être complet de l'entête de module, à la fin de module.

Question 2 : Montrez que le couple de signaux C_1C_0 ne peut jamais atteindre la valeur **`11`**.

Vous montrez votre code à votre chef, il n'est pas content, il n'y comprend rien. Il vous propose alors la méthode suivante :

— Considérer les entrées A_1 et A_0 comme le codage binaire sur 2 bits d'un nombre nommé **A**

— Considérer que les signaux C_1 et C_0 comme le codage binaire d'un état **C** dont les valeurs symboliques seront nommées **S0** pour le code **`00`**, **S1** pour **`01`**, **S2** pour **`10`**.

Question 3 : Quand **C** est dans l'état **S0**, pour quelle(s) valeur(s) de **A** peut on changer d'état, et pour aller vers quel(s) état(s) futur(s) ?

Question 4 : Quand **C** est dans l'état **S1**, pour quelle(s) valeur(s) de **A** peut on changer d'état, et pour aller vers quel(s) état(s) futur(s) ?

Question 5 : Quand **C** est dans l'état **S2**, pour quelles(s) valeur(s) de **A** peut on changer d'état, et pour aller vers quel(s) état(s) futur(s) ?

Question 6 : En déduire un graphe d'états correspondant au circuit (sans oublier d'indiquer la valeur du signal **OK** dans chacun des états).

Question 7 : En déduire un code SystemVerilog équivalent au schéma, plus "compréhensible" pour un être humain et ne faisant pas usage d'équations booléennes.

2.2 *Proposition de correction*

Question 1 :

L'analyse du schéma est la suivante :

— Les entrées du schéma sont les signaux **A0**, **A1**, **clk** et **rst_n**. [1]

— Le schéma contient deux portes **ET**. [2]

— Certaines entrées des portes **ET** sont inversées. [3]

— Le schéma contient deux **bascules D** avec réinitialisation asynchrone active à l'état bas connectées au signal d'horloge **clk** et au signal de réinitialisation **rst_n** [4]

— Le schéma contient les signaux internes **F0**, **F1**, **C0**, **C1**

— Le signal de sortie **OK** est simplement connecté à **C1**

Cela conduit à une traduction directe dans le code 2.1.

1. Voir SystemVerilog A.2 (page 87)
2. Voir les portes élémentaires 1.2.2 (page 7)
3. Dans un schéma, le petit cercle sert à indiquer l'inversion d'un signal 1.2.1 (page 7)
4. Voir l'initialisation des bascules 2.3.4 (page 31)

Question 2 :

Raisonnons par l'absurde. Supposons qu'au cycle courant le couple **(C1,C0)** ait la valeur **(1,1)**. Cela implique qu'au cycle précédent le couple **(F1,F0)** était égal à **(1,1)**. Ce n'est pas possible car les deux portes **AND** admettent respectivement **A1** et **!A1** comme entrées.

Question 3 :

Dans l'état **S0** le signal **C0** vaut **0** ce qui force le signal **F1** à **0**. Les deux seuls état futurs ne peuvent dont être que **S0** ou **S1**. Le choix est déterminé par la valeur de **A**. En résumé :

— Si l'état courant est **S0** et **A** est différent de **2'b11** alors l'état est conservé.

— Si l'état courant est **S0** et **A** est égal à **2'b11** alors l'état futur est **S1**.

Question 4 :

Dans l'état **S1** le signal **C0** vaut **1** ce qui force le signal **F0** à **0**. Les deux seuls état futurs ne peuvent dont être que **S0** ou **S2**. Le choix est déterminé par la valeur de **A**. En résumé :

— Si l'état courant est **S1** et **A** est différent de **2'b01** alors l'état est conservé.

— Si l'état courant est **S1** et **A** est égal à **2'b01** alors l'état futur est **S2**.

Question 5 :

Dans l'état **S2** le signal **C0** vaut **0** ce qui force le signal **F1** à **0**. De même, le signal **C1** vaut **1** ce qui force le signal **F0** à **0**.

— Si l'état courant est **S2** le seul état futur possible est **S0**

Question 6 :

Le graphe demandé pourrait être réalisé de la façon suivante :

— On ne représente dans ce graphe que les changements d'état et les conditions associées.

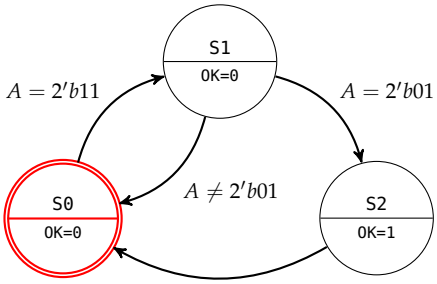


FIGURE 2.2: Proposition de graphe

— On entoure d'un double cercle l'état à l'initialisation (sous l'action du signal `rst_n`

Remarquez la transition sans condition entre l'état `S2` et l'état `S0` : L'automate étant synchrone, il reste dans l'état `S2` pendant exactement un cycle d'horloge.

Question 7 :

— Nous pouvons utiliser un vecteur de bits pour représenter `A`. [5]

— Nous pouvons utiliser un type énuméré pour représenter l'état `C`. [6]

5. Voir SystemVerilog A.9 (page 92)

6. Voir Unités de Contrôle 3.2.1 (page 43)

Cela conduit au code 2.2. Dans ce code, nous n'avons pas pris la peine de coder explicitement un état futur (qui correspond aux signaux `F0` et `F1`. Cela permet de réduire le codage de l'enchaînement d'états à un seul processus synchrone.

Commentaires :

Nous pouvons maintenant interpréter ce que fait ce bloc de traitement : Après initialisation, il est en attente de l'arrivée d'un code `2'b11` sur l'entrée A, suivie immédiatement (au cycle suivant) par un code `2'b01`. Si cette séquence est détectée, il génère une impulsion d'une durée de 1 cycle sur la sortie OK.

```
module vieux_module(                        // L'entête de module
    input logic clk,
    input logic rst_n,
    input logic  A0,
    input logic  A1,
    output logic OK
    ) ;

logic F0, F1, C0, C1 ;                     // Les signaux internes du module

always @(*) F1 <= A0 & !A1 & C0 ;          // La porte AND générant le signal F1

always @(*) F0 <= A0 & A1 & !C0 & !C1 ;    // La porte AND générant le signal F0

always @(posedge clk or negedge rst_n)     // La bascule D générant le signal F1
  if(!rst_n) C1 <= 1'b0 ;
  else C1 <= F1 ;

always @(posedge clk or negedge rst_n)      // La bascule D générant le signal F0
  if(!rst_n) C0 <= 1'b0 ;
  else C0 <= F0 ;

always @(*) OK <= C1 ;                      // La connection de OK au signal C1

endmodule
```

CODE 2.1: Réponse à la question 1

```
// L'entête de module avec les déclarations d'entrées/sorties
module nouveau_module(
    input logic clk,
    input logic rst_n,
    input logic [1:0] A,
    output logic OK
    ) ;
// Le registre d'états est décrit sous forme de type énuméré.
enum logic [1:0] {S0,S1,S2} C ;
//
always @(posedge clk or negedge rst_n)
  if(!rst_n)
    C <= S0 ;
  else
    case (C)
      S0: if (A==2'b11) C <= S1  else C <= S0 ;
      S1: if (A==2'b01) C <= S2  else C <= S0 ;
      S2: C <= S0 ;
      default :; // ce cas n'est jamais atteint.
  endcase

// Le calcul de OK en fonction de l'état des registres
always @(*)
    OK <= (C == S2)  ;
endmodule
```

CODE 2.2: Réponse à la question 7

3
Vitesse d'une balle de tennis

3.1 Enoncé du problème

Nous voulons concevoir un système qui permet de mesurer la vitesse d'une balle de tennis lors de son passage entre deux portiques. Les portiques contiennent un système optique composé d'un émetteur et d'un récepteur de lumière infrarouge (non visible). À son passage, la balle coupe le faisceau lumineux et une impulsion électrique est générée au niveau de chaque portique (un signal passe à `1` puis redescend à `0`).

Pour mesurer la vitesse de la balle, nous devons mesurer le temps entre les impulsions générées au niveau de chaque portique. La distance entre les portiques étant connue, il suffira par la suite de faire une simple division (non demandée ici).

Dimensionnement du système

Les deux portiques sont situés à 10 mètres l'un de l'autre. Les balles de tennis ont un diamètre de 10cm et se déplacent à une vitesse qui varie entre 3,6km/h et 360km/h. Le système de traitement réalisé en logique synchrone dispose d'une horloge *clk*.

Pour être sûr de détecter le passage de la balle, l'impulsion générée par le récepteur du portique doit durer au moins deux périodes de l'horloge *clk*.

Question 1 : À laquelle des fréquences suivantes doit–on faire fonctionner notre système ?

— 10 Hz

— 1 KHz

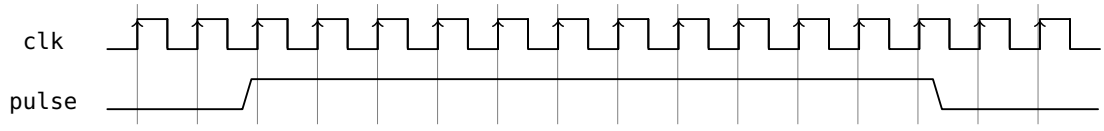
— 100 KHz

Le temps de vol de la balle entre les deux portiques sera mesuré comme un multiple de la période de l'horloge *clk*.

Question 2 : Sur combien de bits faut–il compter pour mesurer le temps de vol pour toutes les vitesses prévues ?

Mise en forme des impulsions

La durée des impulsions (`pulse`) au niveau des portiques dépend de la taille de la balle et de la vitesse à laquelle elle passe. Pour simplifier la suite nous voulons générer à chaque passage de balle une impulsion (`top`) dont la durée est d'une seule période de l'horloge.



Question 3 : Proposez le schéma ou le code SystemVerilog d'un système générant un top d'un cycle d'horloge à chaque passage de balle.

Calcul du temps de vol

Considérant que nous avons maintenant des tops d'un cycle d'horloge, nous voulons mesurer le temps de vol entre les deux portiques.

Question 4 : Proposez le schéma d'un système permettant de compter le nombre de cycles d'horloge entre les deux impulsions générées au niveau de chaque portique.

Question 5 : Donnez le code SystemVerilog correspondant.

3.2 *Proposition de correction*

Question 1 :

La durée de l'impulsion générée par le récepteur du portique correspond au temps pendant lequel la balle coupe le faisceau lumineux. Elle dépend donc de la vitesse de la balle V et de son diamètre D_{balle}. La durée t_{min} minimale correspond à la vitesse maximale de la balle V_{max}. On obtient :

$$t_{min} = \frac{D_{balle}}{V_{max}}$$

L'impulsion devant durer au minimum 2 périodes de l'horloge *clk* du système, nous en déduisons la contrainte sur la période d'horloge *Tclk* :

$$Tclk \leq \frac{D_{balle}}{2 \times V_{max}}$$

D'où une fréquence d'horloge minimale $Fclk_{min}$:

$$Fclk_{min} = \frac{2 \times V_{max}}{D_{balle}}$$

L'application numérique donne $Fclk_{min} = 2kHz$.

Compte tenu du choix proposé, la seule solution possible est $Fclk = 100kHz$.

Question 2 :

L'énoncé nous invite à utiliser un compteur synchrone (voir 2.5.2 page 36) pour mesurer le temps (c'est à dire un nomvre de périodes d'horloge). La taille (nombre de bits) du compteur doit être suffisante pour mesurer le temps de vol de la balle la plus lente. Si on appelle V_{min} la vitesse minimale de la balle et $D_{portiques}$ la distance entre les deux portiques. Alors le temps de vol maximum $Tvol_{max}$ est :

$$Tvol_{max} = \frac{D_{portique}}{V_{min}}$$

Le nombre de cycles d'horloge minimum $Nbcycles_{min}$ nécessaires à la mesure de ce temps de vol est :

$$Nbcycles_{min} = Fclk \times \frac{D_{portique}}{V_{min}}$$

L'application numérique donne $Nbcycles_{min} = 10^6$. Il faut donc réaliser un compteur pouvant compter au moins 1 million de cycles.

— Un compteur binaire de largeur N bits permet de compter de la valeur 0 à la valeur $2^N - 1$.

— Il nous faut donc $N \geq log_2(Nbcycles_{min})$ (log_2 étant le logarithme en base 2).

— L'application numérique donne $N \geq 19.93$, soit $N = 20$.

— Pour éviter de calculer explicitement un log_2 on aurait pu se souvenir de $2^{10} = 1024$ ce qui induit naturellement le résultat $N = 20$...

Question 3 :

Il s'agit d'un problème classique de détection du passage de `0` à `1` d'un signal qui n'est pas une horloge. La réponse se trouve dans le cours, annexe C.8, page 117.

Question 4 :

On peut résoudre le problème de la façon suivante :

1. Un détecteur de passage de `0` à `1` placé au niveau du premier portique génère un signal `start` durant exactement un cycle d'horloge.
2. Un détecteur de passage de `0` à `1` placé au niveau du deuxième portique génère un signal `stop` durant exactement un cycle d'horloge.
3. Un compteur 20 bits est mis à zéro et autorisé à compter lorsque `start` vaut `1`.
4. Le compteur 20 bits est arrêté lorsque `stop` vaut `1`.

Nous pouvons créer un signal intermédiaire `enable` permettant d'autoriser l'incrémentation du compteur. Le comportement du signal `enable` est le suivant :

1. Enable est initialisé à `0`.
2. Enable passe à `1` lorsque `start` vaut `1`.
3. Enable repasse à `0` lorsque `stop` vaut `1`.

Le schéma suivant permet d'obtenir ce comportement. La méthode d'initialisation de la bascule D n'est pas indiquée, mais le `0` en haut à droite signifie qu'elle devra être initialisée à `0` :

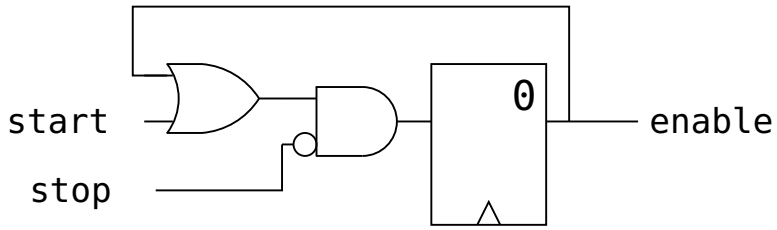


FIGURE 3.1: Génération d'un signal d'autorisation de comptage

Le schéma suivant propose une implémentation du compteur proprement dit. Il est réinitialisé de manière synchrone par le signal `start`. Le comptage s'effectue lorsque `enable` est actif. Les traits en gras représentent le signal de 20 bits correspondant au signal `compteur` et au calcul de sa valeur future. Le schéma mélange arithmétique booléenne et arithmétique : additionner `1` à `compteur` lorsque le signal `enable` vaut `1` revient à additionner `enable`...

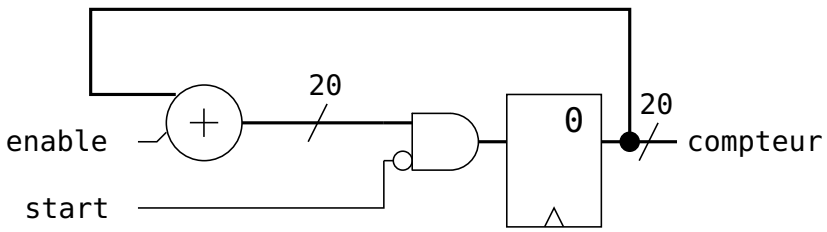


FIGURE 3.2: Le compteur proprement dit

Question 5 :

Le schéma étant déjà réalisé, le code SystemVerilog en est une simple traduction. Nous n'avons pas précisé dans les schémas précédents les méthodes d'initialisation des registre, nous choisissons arbitrairement une initialisation asynchrone sur un signal `reset_n` actif à l'état bas.

Notez, dans le code proposé, la déclaration du registre `compteur` de 20 bits.

```
module nouveau_module(
    input logic clk,
    input logic reset_n,
    input logic start,
    input logic stop,
    output logic [19:0] compteur ;
    ) ;

logic enable ;

// Code pour la génération du signal enable
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    enable <= 1'b0 ;
  else
    if(stop)
       enable <= 1'b0;
    else
      if(start)
        enable <= 1'b1 ;

// Code du compteur
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    compteur  <= 20'd0 ;
  else
     if(start)
       compteur <= 20'd0 ;
     else if (enable)
       compteur <= compteur + 1'b1 ;
endmodule
```

CODE 3.1: Réponse à la question 5

4
Où le lecteur découvre la Rétroconception

4.1 Enoncé du problème

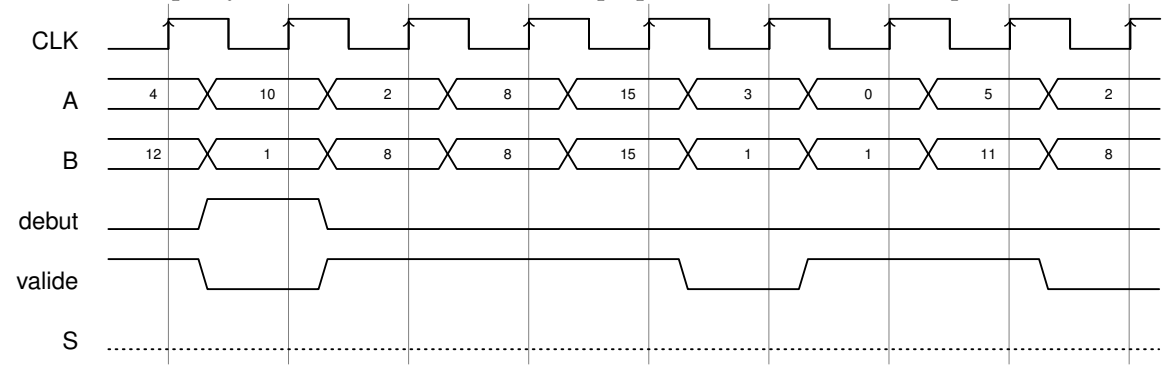
La rétroconception (ou reverse-engineering) n'est pas la conception à la mode antique, mais l'analyse de la conception d'autres ingénieurs ou scientifiques de manière à comprendre, copier et éventuellement améliorer un dispositif existant.

Vous disposez du code SystemVerilog de l'opérateur F1 suivant, qui est manifestement en logique synchrone...

```
module F1( input  logic clk,debut,valide,
           input  logic [3:0] A,B,
           output logic [7:0] S
         );
always @(posedge clk)
  if(debut) S <= 8'b0 ;
  else if(valide)
       S <= A * B + S ;
endmodule
```

Code 4.1: code du module F1

Question 1 : Complétez le chronogramme correspondant au test de la fonction F1. Vous indiquerez précisément, à chaque cycle, la valeur de la sortie *S*. Expliquez le traitement effectué par la fonction.



Question 2 : Vous disposez d'une variante F2 du code SystemVerilog de l'opérateur. L'opérateur F2 réalise–t–il le même calcul que l'opérateur F1 ? Justifiez votre réponse qu'elle soit positive ou négative.

Question 3 : Vous disposez d'une variante F3 du code SystemVerilog de l'opérateur. L'opérateur F3 réalise–t–il le même calcul que l'un des opérateurs F1 ou F2 ? Justifiez votre réponse.

Question 4 : Nous désirons choisir, parmi F1 F2 ou F3, l'opérateur pouvant fonctionner à la fréquence d'horloge la plus élevée (indépendamment la fonction réalisée). Quel opérateur choisissez vous ? Justifiez votre réponse.

```
module F2( input  logic clk, debut, valide,
           input  logic [3:0] A,B,
           output logic [7:0] S
         );
logic [7:0] P ;
always @(posedge clk)
if(debut) S <= 8'b0 ;
else if(valide) begin
     P <= A * B ;
     S <= P + S ;
end
endmodule
```

CODE 4.2: code du module F2

```
module F3( input  logic clk, debut, valide,
           input  logic [3:0] A,B,
           output logic [7:0] S
         );
logic [7:0] P,X;

always @( * ) P <= A * B ;

always @( * ) begin
  if(debut)       X <= 8'b0 ;
  else if(valide) X <= P + S ;
  else            X <= S ;
end

always @(posedge clk) S <= X ;

endmodule
```

CODE 4.3: code du module F3

4.2 *Proposition de correction*

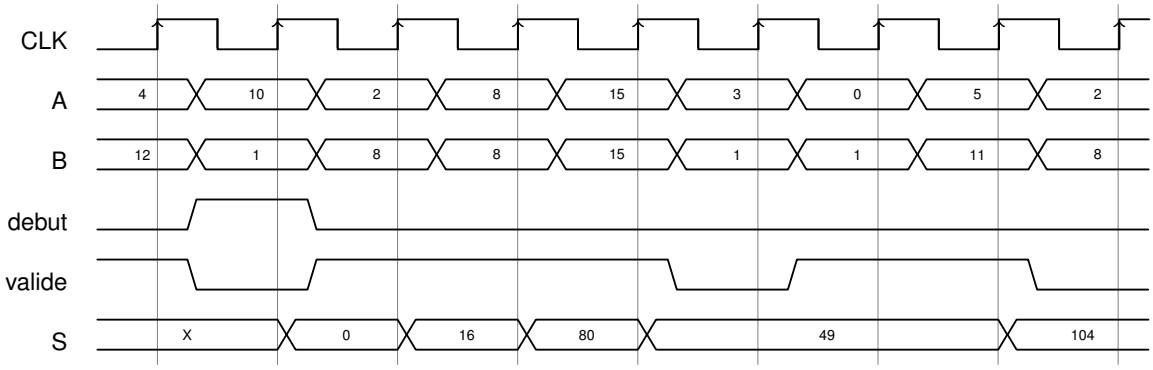
Question 1 :

Le signal **S** est affecté dans un processus (**always**) synchrone du front montant (**posedge**) de l'horloge **clk**. Il faut examiner l'état des entrées **A**, **B**, **debut** et **valide** au moment du front montant de **clk** pour déterminer la nouvelle valeur de **S** après le front montant. La nouvelle valeur de **S** est simplement la somme de la valeur courante de **S** avec le produit de **A** et de **B** si le signal **valide** est actif. Enfin, ne pas oublier que dans un processus synchrone, la conservation de la valeur est implicite : il n'est pas nécessaire d'indiquer ce qui se passe lorsque **debut** et **valide** sont nuls.

Attention :

— Le dispositif n'étant pas initialisé, au début du chronogramme nous ne connaissons pas la la valeur de **S**, c'est la raison pour laquelle la valeur indiquée est **X** jusqu'au moment ou le signal **debut** devient actif.

— Le signal **S** étant codé sur 8 bits, le calcul se fait modulo 255.

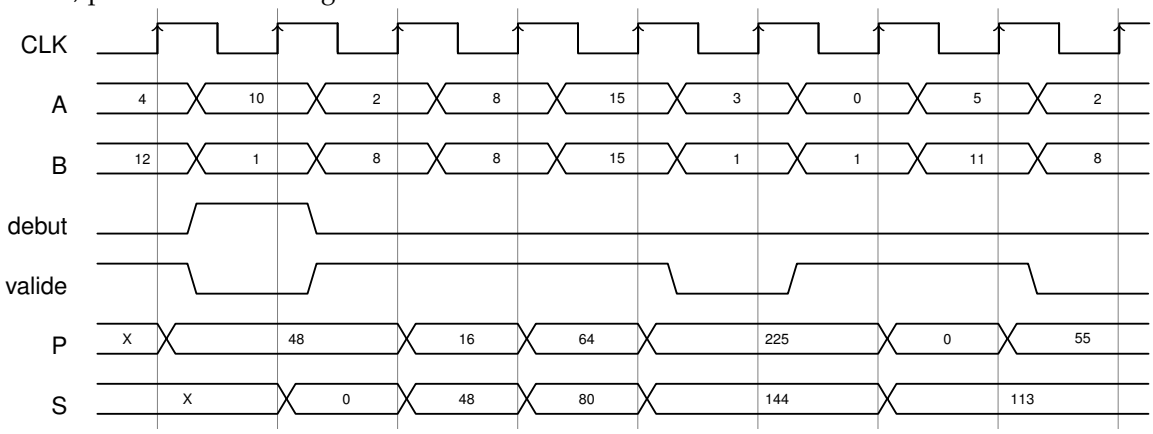


Question 2 :

Pour comprendre les éventuelles différences entre **F1** et **F2** il faut se souvenir que tout signal affecté par l'opérateur "**<=**" dans un processus synchrone est implanté matériellement sous forme de registres. Le signal **P** nouvellement introduit a donc un retard d'un cycle par rapport aux signaux d'entrée **A** et **B**. Le signal **S** aura, en conséquence un retard de 2 cycles par rapport aux entrées **A** et **B**.

Le code **F2** ne donne donc pas le même résultat que le code **F1**.

Pour mieux comprendre, nous pouvons reprendre le chronogramme en introduisant le signal **P**. Pour la création d'un tel chronogramme, la bonne méthode consiste à d'abord construire le signal **P** sur toute sa durée, puis construire le signal **S**.



Question 3 :

Le code **F3** est totalement équivalent au code **F1**.

Le calcul combinatoire "**A*B + S**" a simplement été extrait du processus synchrone et encodé sous la forme de 2 processus combinatoires explicites, le premier calculant le produit et le second l'accumulation l'accumulation.

Le signal **X** représente la valeur future (au prochain cycle d'horloge de **S**.

Remarquez que l'on est obligé dans ce style d'écriture de coder la totalité des situations possibles (dont le maintien de la valeur courante par le bloc "**else X <= S**") pour garantir que processus est bien combinatoire.

Question 4 :

Les implémentations **F1** et **F3** sont équivalentes.

L'implémentation **F2** décompose le calcul en deux étapes (2 cycles), le premier cycle étant consacré au calcul du produit et le deuxième cycle étant consacré à l'accumulation.

Ainsi les chemins combinatoires de l'implémentation **F2** seront temporellement plus courts que ceux des deux autres implémentations.

La fréquence d'horloge de l'implémentation **F2** pourra donc être plus élevée.

Ce type de structure, ainsi que l'amélioration de performance obtenue est décrite dans le chapitre 2.5.3 du polycopié (page 38).

5
Amélioration du nanoprocesseur

Pour résoudre ce problème, il faut maîtriser l'ensemble du cours sur la logique combinatoire, la logique séquentielle et évidemment le cours sur le nanoprocesseur (Voir poly chapitre 4, page 47), ainsi que le TP associé.

5.1 Énoncé du problème

Tel que proposé en cours et en TP, le nanoprocesseur manque de fonctionnalités essentielles. Il est, par exemple, difficile de structurer le code en fonctions et sous-programmes.

Nous désirons ajouter deux instructions au nanoprocesseur :

— **JSR** : pour "Jump to SubRoutine". Son argument est l'adresse en mémoire du début du sous-programme appelé.

— **RTS** : pour "ReTurn from Subroutine". Son argument est ignoré

— Ces instructions sont des instructions de saut au même titre que les instructions JMP, JNZ et JNC.

Nous rappelons que le microprocesseur est piloté par un automate en 3 cycles nommés **IF**, **AF** et **EX** pour "Instruction Fetch", "Address Fetch" et "Execute".

Vous trouverez, figure 5.1 , un schéma du nanoprocesseur. Enfin les codes du compteur de programme et du contrôleur de base sont rappelés en codes 5.1 et code 5.2.

Code 5.1: Code de base du PC. Seul le code utile est indiqué

```
...
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    PC <= 0 ;
  else
    if(Load_PC)
       PC <= Q ;
    else
       PC <= PC + Inc_PC ;
...
```

```
...
always @(*)
begin
   Inc_PC    <= (Etat == IF) || (Etat == AF) ;
   Load_PC   <= (Etat == AF) && ((I == JMP || (I==JNC && !C) || (I==JNZ && !Z)) ;
   Load_Add  <= (Etat == AF) ;
   Sel_Add   <= (Etat == EX) ;
   Load_I    <= (Etat == IF) ;
   Load_AZC  <= ...
   WRITE     <= (Etat == EX) && (I ==  STA) ;
end
...
```

CODE 5.2: Code de base du contrôleur. Seul le code utile est indiqué

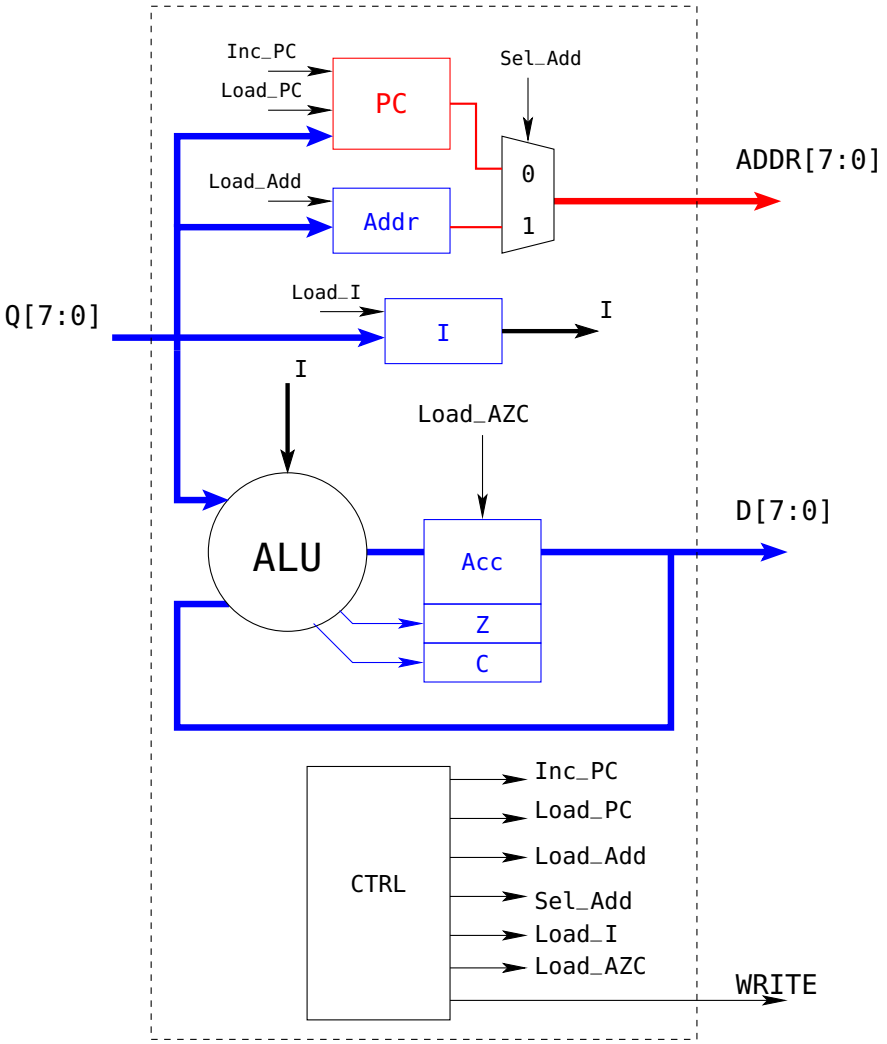


FIGURE 5.1: Schéma du nanoprocesseur

5.1.1 *Une approche simpliste*

Nous limitons l'appel de sous-programme à un seul niveau (seul le programme principal peut appeler un sous-programme).

L'exécution de **JSR** doit réaliser les actions suivantes :

— Sauvegarder dans le nanoprocesseur l'adresse de retour, c'est à dire l'adresse de l'instruction suivant l'instruction courante.

— Forcer le nanoprocesseur à se brancher à l'adresse du sous-programme comme dans le cas d'un **JMP**.

L'exécution de **RTS** doit réaliser les actions suivantes :

— Forcer le nanoprocesseur à se brancher à l'adresse d'instruction précédemment sauvegardée par l'instruction **JSR**.

Question 1 : Proposez une modification du schéma, et une modification des codes permettant d'implémenter les instructions **JSR** et **RTS**.

5.1.2 *Une approche plus générale (**BONUS**)*

Pour pouvoir généraliser les appels de sous-programmes (appels imbriqués), la méthode précédente nécessite de multiplier les registres dans le micro-processeur. Une méthode plus générique consiste à sauver l'adresse de retour de sous-programme dans la mémoire externe, et de ne conserver en interne que la position dans la mémoire de cette adresse de retour.

Pour cela nous ajoutons au nanoprocesseur un registre spécifique appelé "pointeur de pile" ou **SP** pour "stack pointer". Nous utiliserons les adresses "hautes" de la mémoire pour stocker les adresses de retour.

La gestion de **SP** en relation avec **JSR** et **RTS** est la suivante :

— A l'initialisation **SP** est forcé à la valeur maximale des adresses en mémoire : 255.

— A chaque exécution de **JSR** :

  — L'adresse de retour de sous programme est stockée en mémoire à l'adresse pointée par **SP**

  — **SP** est décrémenté de 1 (pour anticiper le stockage d'une éventuelle nouvelle adresse de retour...)

— A chaque exécution de RTS,

  — **SP** est incrémenté de 1 (pour revenir dans l'état avant l'appel du sous-programme)

  — Le nanoprocesseur récupère la donnée pointée par **SP** : l'adresse de retour.

Nous ne chercherons pas à gérer les cas limites (trop d'appels imbriqués pour la taille de la mémoire, **RTS** sans **JSR** préalable...).

Question 2 : Proposez une modification du schéma, et une modification du code permettant d'implémenter les instructions JSR et RTS.

5.2 *Proposition de correction*

Question 1 :

L'éxécution du saut proprement dit par l'instruction **JSR** est similaire aux autres sauts : il suffit donc de modifier le calcul du signal **Load_PC** dans le contrôleur. La nouvelle équation devient :

```
Load_PC <= (Etat == AF) && (I == JMP)||(i==JNC && !C)||(I==JNZ && !Z)||(I=JSR)) ;
```

Il faut sauvegarder l'adresse de retour (l'adresse de l'instruction suivant l'instruction courante). Pour cela nous ajoutons un nouveau signal de contrôle nommé **Push_PC** à la sortie du contrôleur. Ce signal sera utilisé en entrée du bloc en charge de calcul du PC et sera calculé pendant le cycle **AF** comme **Load_PC**.

```
Push_PC <= (Etat == AF) && (I=JSR) ;
```

Le code 5.3 est une proposition de modification du PC. Nous ajoutons un registre de sauvegarde **RPC** qui est chargé si **Push_PC** est actif. Ainsi **RPC** prend la valeur qu'aurait du prendre le **PC** si le programme avait continué en séquence.

```
...
output logic [7:0] RPC ;
...
always @(posedge clk or negedge reset_n)
  if(!reset_n) begin
    PC <= 0 ;
    RPC <= 0 ;
  end else begin
    if(Push_PC) RPC <= PC + Inc_PC ;
    if(Load_PC) PC  <= Q else PC <= PC + Inc_PC ;
  end
...
```

CODE 5.3: Introduction du registre RPC

L'instruction **RTS**, est encore dans un cas similaire à une instruction de type **JMP**, mais il faut cette fois utiliser la valeur courante du registre **RPC** pour effectuer le saut. Cela signifie une nouvelle modification du signal **Load_PC** :

```
Load_PC <= (Etat == AF) && ((I == JMP)||(i==JNC && !C)||(I==JNZ && !Z)||(I==JSR)||(I==RTS));
```

Enfin, il faut forcer le bloc PC à récupérer l'adresse de retour. Pour cela nous introduisons un signal **Pop_PC** dans le contrôleur :

```
Pop_PC <= (Etat == AF) && (I=RTS) ;
```

Le code 5.4 est une proposition de modification du PC. En fonction de la valeur du signal **Pop_PC** un choix est fait entre **Q** (cas des instructions de type **JMP** ou **JSR**) et **RPC** (cas de l'instruction **RTS**).

```
...
logic [7:0] RPC ;
always @(posedge clk or negedge reset_n)
  if(!reset_n) begin
    PC <= 0 ;
    RPC <= 0 ;
  end
  else begin
    if(Push_PC) RPC <= PC + Inc_PC ;
    if(Load_PC) begin
      if(Pop_PC) PC <= RPC ;
      else       PC <= Q ;
    end
    else PC <= PC + Inc_PC ;
  end
...
```

CODE 5.4: Code du PC avec JSR et RTS implémentés

Question 2 :

Pour établir le code, il est utile de bien associer les actions à réaliser à chaque état de l'automate du nanoprocesseur.

Cas de **JSR** :

— **IF** : Exécution classique. (récupération de l'instruction)

— **AF** : Exécution classique pour un saut : **Load_pc=1** en même temps nous sauvegardons la valeur incrémentée de **PC** dans **RPC**

— **EX** : Nous sauvons en mémoire (**write=1**) la valeur de **RPC** (capturée au cycle AF) à l'adresse **SP**, et nous décrémentons **SP** (**Push_PC=1**).

Cas de **RTS** :

— **IF** : exécution classique. (récupèration de l'instruction)

— **AF** : Nous incrémentons **SP** (**Pop_PC=1**)

— **EX** : Nous lisons (**write=0**) la valeur en mémoire à l'adresse **SP** et nous la plaçons dans le **PC** (**Load_pc=1**) pour exécuter le saut.

Les nouvelles équations pour les signaux de contrôle sont :

```
Load_PC <= ((Etat == AF) && ((I == JMP)||(i==JNC && !C)||(I==JNZ && !Z)||(I==JSR)) ||
           ((Etat == EX) && (I==RTS));
Push_PC <= (Etat == EX) && (I=JSR) ;
Pop_PC  <= (Etat == AF) && (I=RTS) ;
```

Nous supposons que nous disposons d'un nouveau bloc **SP** chargé de gérer le pointeur de pile. Ce bloc est un bloc synchrone qui utilise les signaux **Push_PC** et **Pop_PC**.

```
...
logic [7:0] SP ;
always @(posedge clk or negedge reset_n)
  if(!reset_n) begin
    SP <= 255 ;
  end
  else begin
    if(Push_PC) SP <= SP - 1;
    if(Pop_PC)  SP <= SP + 1;
  end
...
```

CODE 5.5: Code du bloc **SP**

Le code de gestion du PC est à nouveau amendé : Le registre RPC ne sert qu'a préparer la future sauvegarde du PC dans la mémoire.

```
...
logic [7:0] RPC ;
always @(posedge clk or negedge reset_n)
  if(!reset_n) begin
    PC <= 0 ;
    RPC <= 0 ;
  end
  else begin
    if(Load_PC) begin
       PC <= Q ;
       RPC <= PC + Inc_PC
    end
    else PC <= PC + Inc_PC ;
  end
...
```

CODE 5.6: Code du PC ajusté pour sauvegarder le PC dans RPC.

Enfin il faut mettre en place les multiplexeurs commandés par un nouveau signal de contrôle **Sel_SP** permettant de :

— gérer la sauvegarde de **RPC** en mémoire dans l'instruction **JSR**,

— relire l'adresse de retour en mémoire pour l'instruction **RTS**.

```
Sel_SP <= (Etat == EX) && (I==JSR || I==RTS) ;
```

Cela abouti au schéma de la figure 5.2.

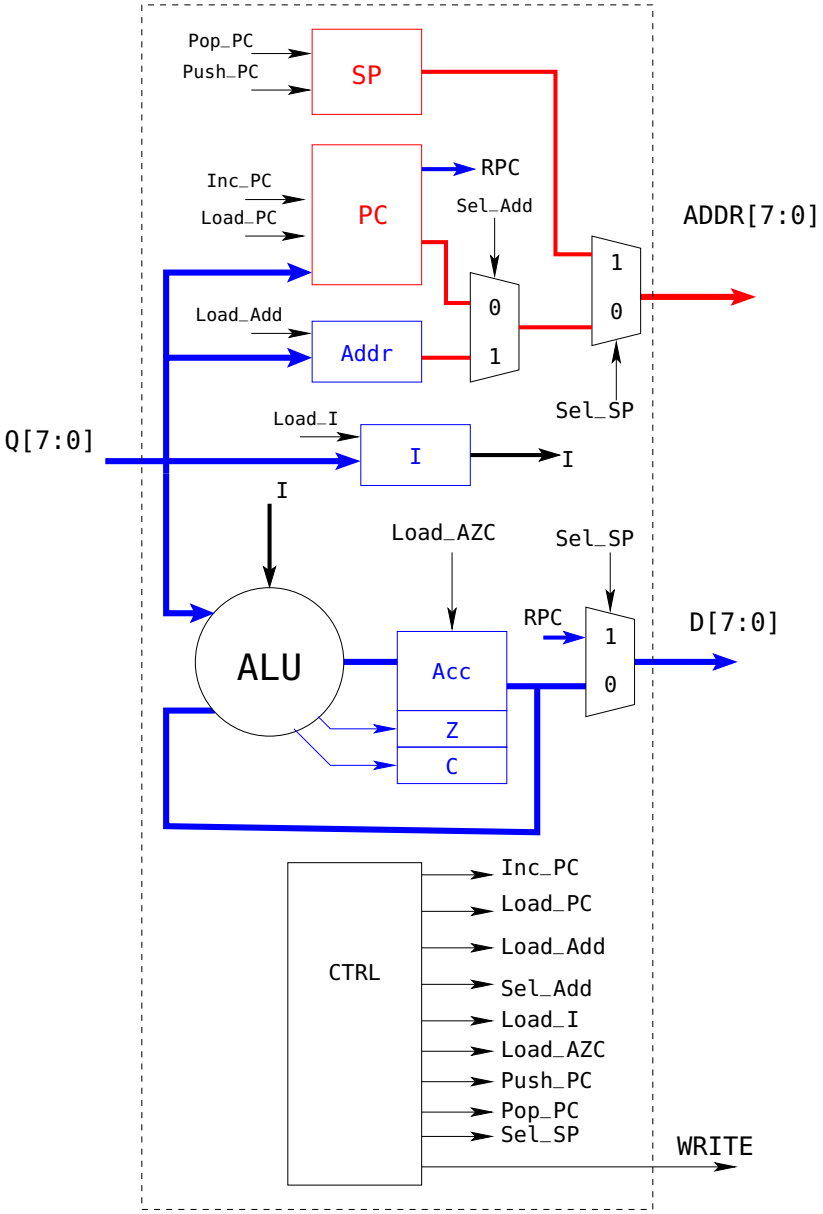


FIGURE 5.2: Schéma du nanoprocesseur modifié

6
Techniques basse consommation

La résolution de ce problème, nécessite la compréhension des chapitres sur la logique syncrhone, le pipeline, ainsi que le chapitre 5 consacré à la logique CMOS.

6.1 Enoncé du problème

L'objectif de cet exercice est d'explorer des techniques architecturales permettant de réduire la consommation des circuits intégrés. Nous disposons d'une fonction de calcul combinatoire **F** pour laquelle nous connaissons :

— Le temps de calcul T_{calc} en fonction de la tension d'alimentation V_{dd} (voir **figure 6.3**)

— L'énergie consommée par calcul E_{calc} en fonction de la tension d'alimentation V_{dd} (voir **figure 6.4**)

Puissance consommée par un opérateur de calcul est égale au produit de sa fréquence de fonctionnement F_{calc} par l'énergie consommée à chaque calcul :

— $P_{calc} = F_{calc} * E_{calc}$

*6.1.1 Analyse d'une structure synchrone utilisant la fonction **F***

La **figure 6.1** présente une utilisation de la fonction **F** dans un environnement synchrone. La structure reçoit une suite de valeurs A_0,A_1,... et doit générer une suite de valeurs **F(**A_0**)**, **F(**A_1**)**,...

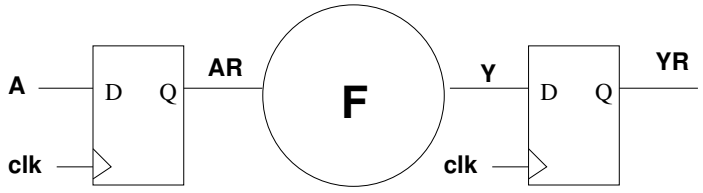


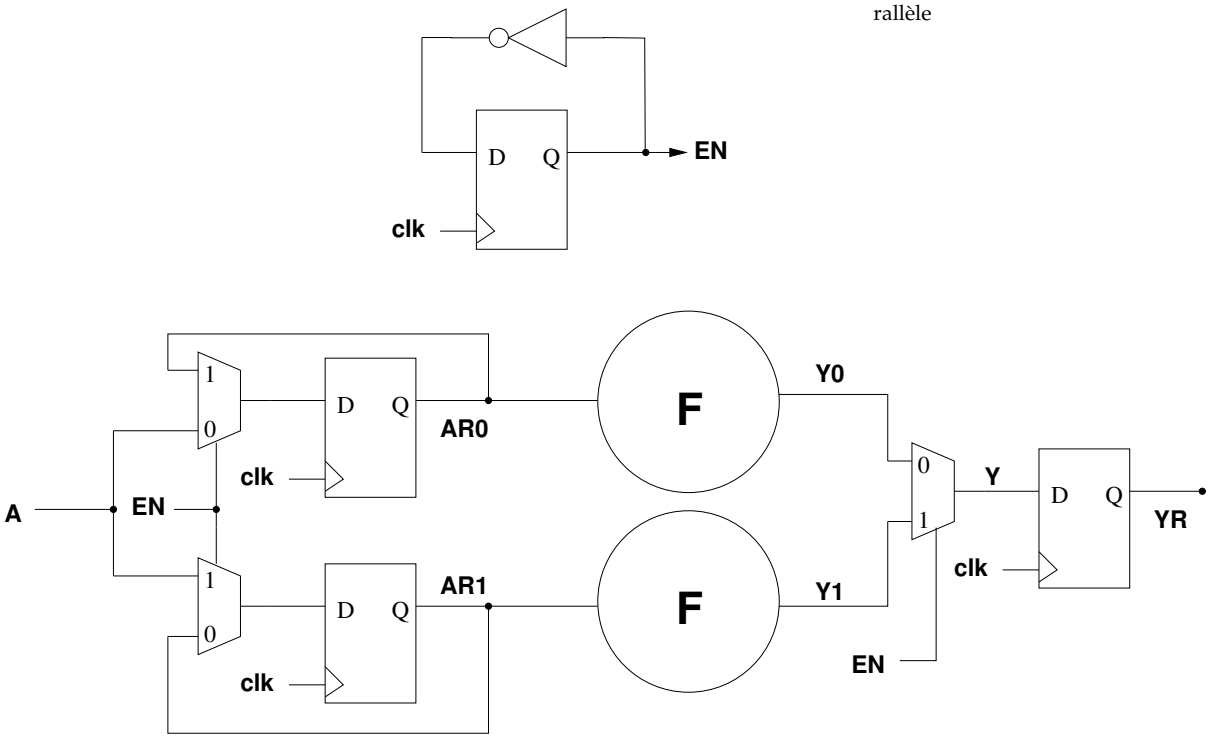
FIGURE 6.1: Architecture synchrone

Question 1 : Complétez le chronogramme de la **figure 6.7**.
Question 2 : En négligeant le temps de propagation des bascules, déterminez la tension d'alimentation minimale V_{ref} permettant de faire fonctionner cette structure avec une horloge **CLK** de fréquence $F_{ref} = 1GHz$.
Question 3 : En négligeant la consommation des bascules, déterminez la puissance consommée par cette structure pour la tension V_{ref} et la fréquence F_{ref}. N'oubliez pas de préciser les unités...

6.1.2 Analyse d'une structure synchrone parallélisée

La **figure 6.2** présente une version parallélisée de l'architecture précédente, dans laquelle nous dédoublons la fonction **F**.

FIGURE 6.2: Architecture synchrone parallèle



Question 4 : Complétez le chronogramme de la **figure 6.6**. N'oubliez pas la génération du signal **EN**.

Question 5 : Expliquez le fonctionnement de cette structure.

Question 6 : Pendant combien de périodes de l'horloge **CLK** chacune des données A_i est elle maintenue en entrée de l'une ou l'autre des fonctions **F** ? De combien de temps dispose-t-on pour le calcul de chaque **F(**A_i**)** sachant que l'horloge a une fréquence $F_{ref} = 1GHz$? (on négligera le temps de propagation dans les multiplexeurs).

Question 7 : Compte tenu du nouveau temps de calcul disponible, montrez qu'il est possible de diminuer la tension d'alimentation du montage tout en fixant la fréquence F_{ref} à $1GHz$. Déterminez la tension d'alimentation minimale V_{min} permettant de conserver cet fréquence de $1GHz$.

Question 8 : En déduire l'énergie E_{min} consommée par la fonction **F** à la tension V_{min}. Puis la puissance totale consommée par la structure parallèle. N'oubliez pas la encore de préciser les unités.

Question 9 : Nous avons utilisé une technique de parallélisme pour diminuer la consommation d'une structure de calcul synchrone. Pensez vous qu'une technique de pipeline (pipeline de la fonction **F** en deux sous-fonctions **F1** et **F2** de temps de calcul $T_{calc}/2$) permettrait d'obtenir un résultat similaire ? Expliquez.

FIGURE 6.3: Temps de calcul de la fonction **F**

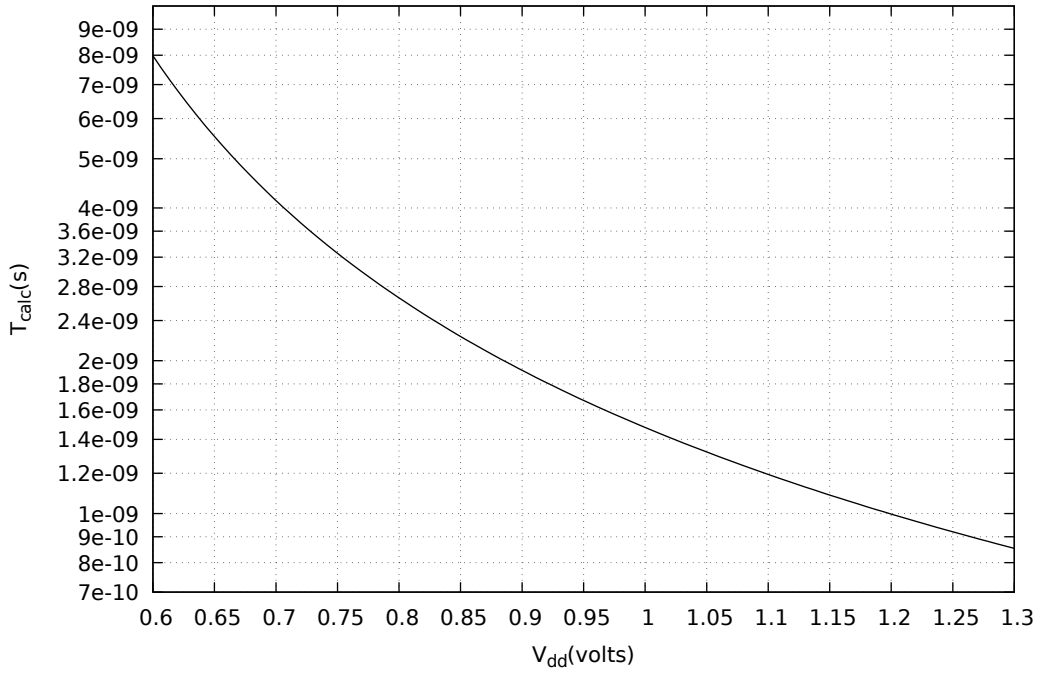


FIGURE 6.4: Energie consommée par calcul de la fonction **F**

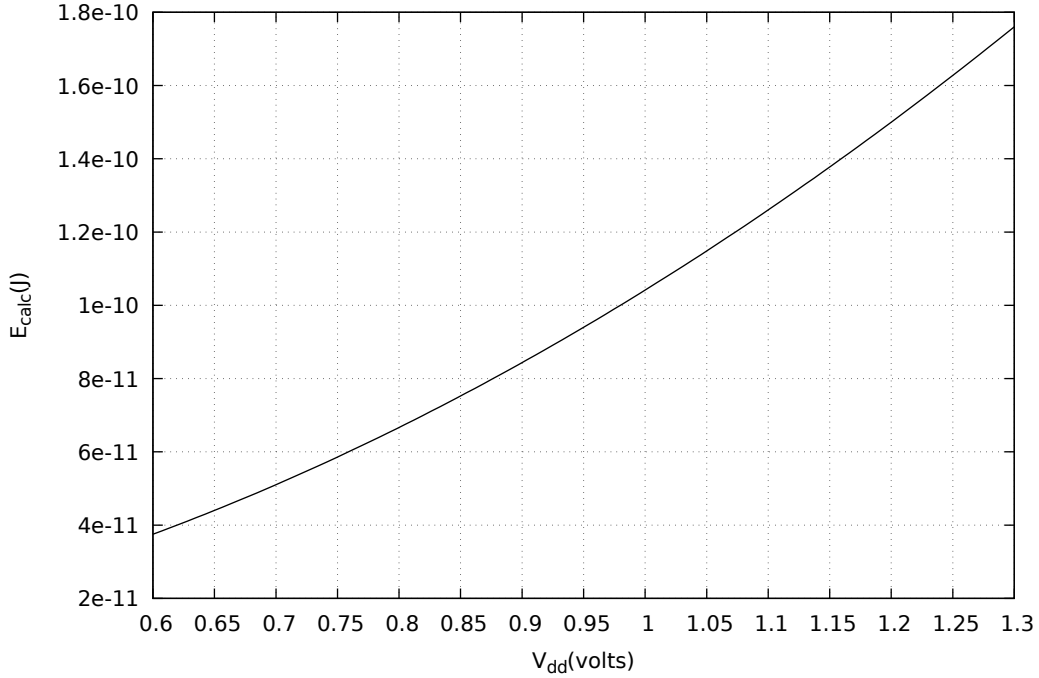


FIGURE 6.5: Chronogramme de fonctionnement de la structure synchrone (Question 1)

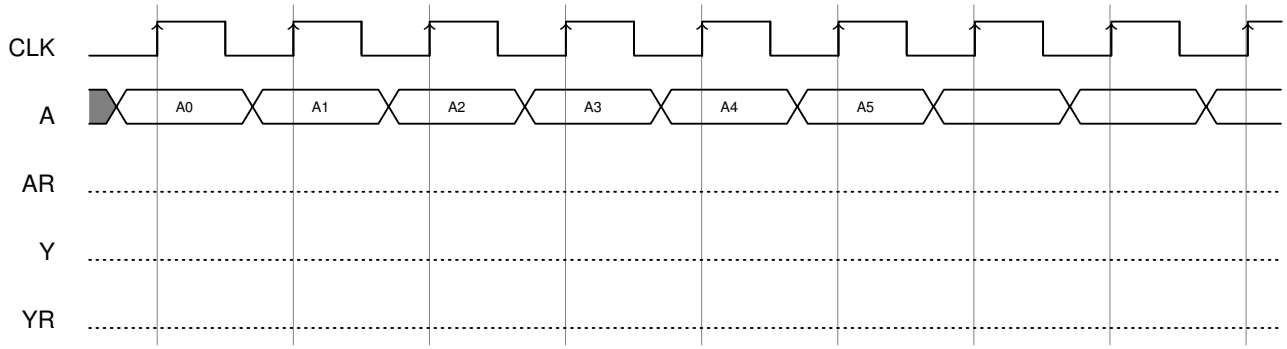
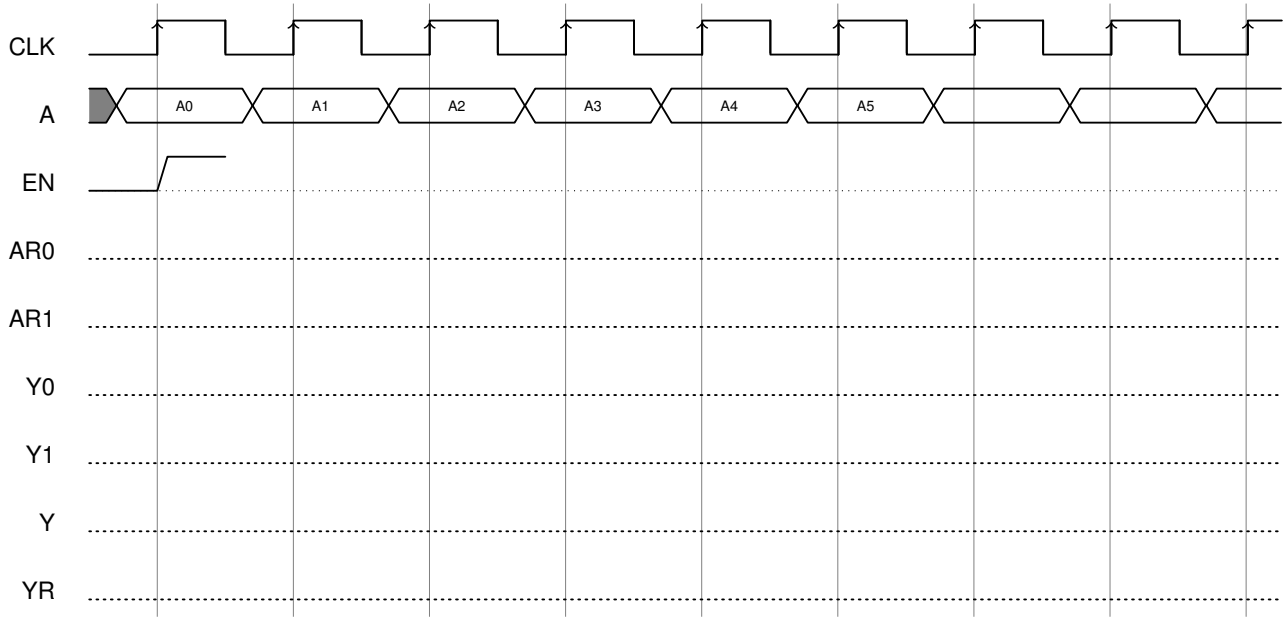


FIGURE 6.6: Chronogramme de fonctionnement de la structure parallèle synchrone (Question 4)

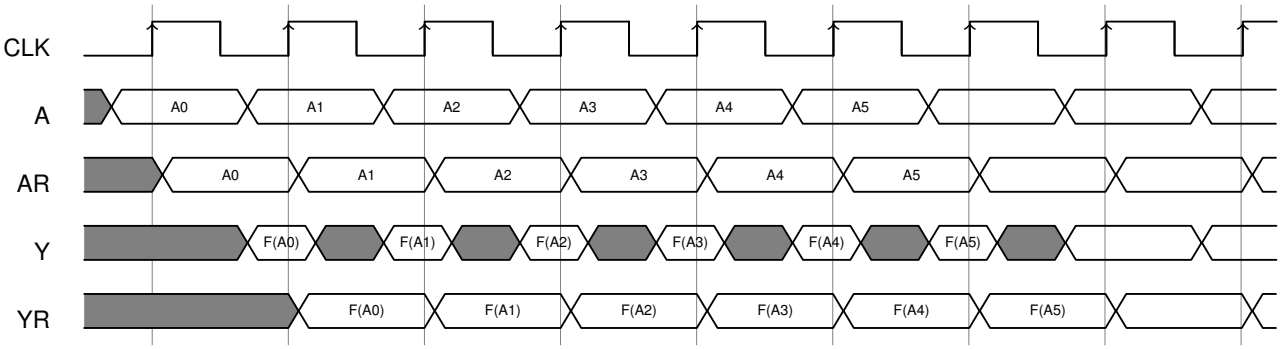


6.2 *Proposition de correction*

Question 1 :

Le signal **AR** est la sortie d'une bascule D, il prend donc la valeur de **A** après chaque front montant de l'horloge. Le signal **Y** est la sortie d'un calcul combinatoire fonction de **AR**, il changera à chaque changement de **AR** avec un retard du au temps de calcul de **F**. Pendant ce temps de calcul, sa valeur n'est pas stable. Enfin le signal **YR** est la sortie d'une bascule D, il prend donc la valeur de **Y** après chaque front montant de l'horloge. Evidemment cela ne foncionne que si le temps de calcul de **F** est inférieur à la période de l'horloge.

FIGURE 6.7: Chronogramme de fonctionnement de la structure synchrone (Question 1)



Question 2 :

Pour que la structure fonctionne à au moins $1GHz$, il faut que le temps de calcul de la fonction **F** soit inférieure à $1ns$. En examinant la figure 6.3, nous en déduisons une tension d'alimentation minimale de $V_{ref} = 1.2V$.

Question 3 :

En examinant la figure 6.4, nous pouvons déterminer l'énergie utilisée par la fonction **F** à chaque calcul pour la tension V_{ref} : $E_{ref} = 1.5 \times 10^{-10} J$.

Il suffit de multiplier par la fréquence de fonctionnement pour obtenir la puissance consommée : $P_{ref} = F_{ref} * E_{ref} = 0.15W$

Question 4 :

Avant de construire le chronogramme, on peut observer que le schéma repose sur l'utilisation du signal **EN**. La structure de génération du signal **EN** est assez facile à comprendre : La valeur de **EN** au prochain cycle d'horloge est l'opposé de sa valeur au cycle courant.

On peut enfin observer que les signaux **AR0** et **AR1** sont générés par des **bascules D avec Enable** telles que décrites dans l'annexe C.1.1 page 111. La seule différence d'une bascule à l'autre est que leurs signaux de validation sont opposés.

Pour créer convenablement le chronogramme, le plus simple est de traiter d'abord le signal **EN**, puis le signal **AR0** puis le signal **Y0** qui s'en déduit directement. On traitera ensuite **AR1**, et **Y1**. On traitera enfin **Y** puis **YR**. Cela aboutit au chronogramme de la figure 6.8.

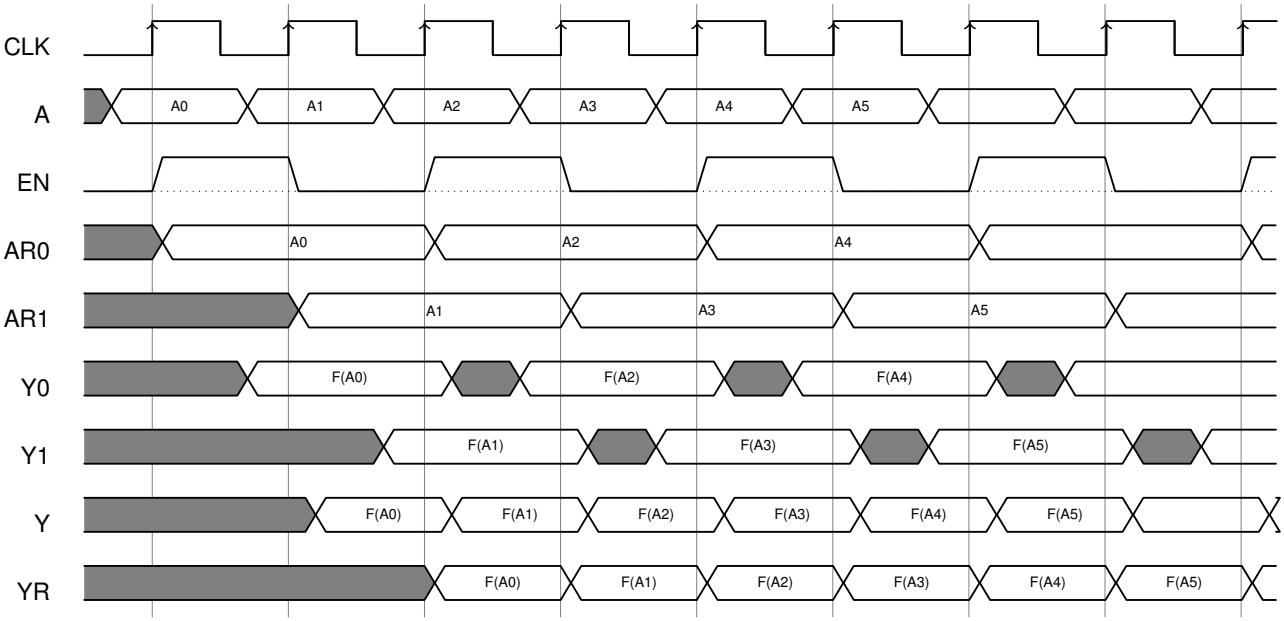
Question 5 :

Le calcul des échantillons pairs est réalisé par la branche supérieure de la structure (signaux **AR0** et **Y0**). Le calcul des échantillons impairs est réalisé par la branche inférieure de la structure (signaux **AR1** et **Y1**). Le multiplexeur de sortie est chargé de choisir le bon résultat pour le stockage dans le registre **YR**.

Question 6 :

Les données sont maintenues en entrée de chaque fonction **F** pendant 2 cycles d'horloge. On dispose

FIGURE 6.8: Chronogramme de fonctionnement de la structure parallèle synchrone (Question 4)



donc de deux cycles d'horloge pour réaliser le calcul de la fonction **F**, soit $2ns$.

Question 7 :

En reprenant l'abaque du temps de calcul de **F**, on voit que l'on peut diminuer la tension d'alimentation tout en conservant un temps de calcul inférieur ou égal à $2ns$. La valeur minimale de la tension est donc $V_{min} = 0.9V$.

Question 8 :

En reprenant l'abaque de consommation de **F**, on obtient une énergie $E_{min} = 8.24 \times 10^{-11}J$ pour la fonction **F**. La puissance consommée par la structure de calcul sera :

$P_{min} = (2 * E_{min}) * (F_{ref}/2) = 0.084W$

En effet, la structure contient 2 fonctions **F**, mais chacune de ces fonctions travaille à fréquence moitié.

Question 9 :

En mettant en place le pipeline, on obtient deux fonctions **F1** et **F2** dont le temps de calcul est de $0.5ns$ à la tension d'alimentation $1.2V$. On peut donc diminuer la tension d'alimentation jusqu'au point au le temps de calcul de ces fonctions devient égal à $1ns$. La conséquence sera la encore une diminution de l'énergie consommée, puis de la puissance consommée.

7
Réalisation d'un Digicode

7.1 Enoncé du problème

On désire réaliser un digicode. L'objectif de l'exercice est de concevoir un système permettant de détecter l'entrée du bon code (**246A**) sur un clavier et qui, dans ce cas, passse une sortie **`S`** à 1 pendant un cycle d'horloge permettant ainsi d'ouvrir la serrure.

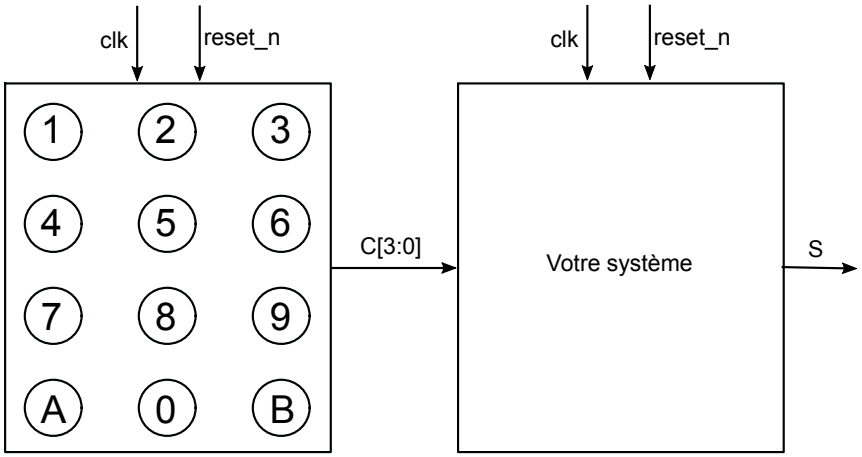
Le système dispose d'une horloge **`clk`** à 10MHz et d'un signal d'initialisation **`reset_n`**, actif à l'état bas. Ces deux signaux seront implicites sur les schémas, donc non représentés. On se contentera d'indiquer la valeur d'initialisation des éventuelles bascules D. Enfin l'appui d'une touche est sensée durer plusieur cycles d'horloge.

Vous disposez d'un clavier tel que décrit en figure 7.1. Ce clavier dispose en sortie d'un bus **synchrone** sur 4 bits, **`C[3:0]`**, indiquant la touche appuyée :

— Si on appuie sur touche, **`C`** prend comme valeur le numéro de la touche : 0 pour la touche 0, 1 pour la touche 1, . . . , 9 pour la touche 9, 10 (0xA) pour la touche A, 11 (0xB) pour la touche B.

— Si on n'appuie sur aucune touche, ou si on appuie sur deux touches en même temps, **`C`** prend comme valeur 15 (0xF en hexadécimal).

FIGURE 7.1: Digicode



Question 1 Détection de l'appui sur une touche

Faites le schéma d'un dispositif qui produit un signal **`enable`** passant à 1 pendant un seul cycle d'horloge lorsqu'une touche est pressée, quelle que soit la durée de l'appui sur la touche.

Question 2 Détection du code

Faites le schéma d'un dispositif qui produit un signal **OK** qui passe à 1 uniquement lorsque l'utilisateur a entré le bon code (246A). Ce signal peut rester à 1 aussi longtemps qu'on le souhaite. Si l'utilisateur a entré une séquence de touches ne correspondant pas au bon code, ce signal doit passer à 0.

Question 3 Ouverture de la porte

Faites le schéma d'un dispositif prenant en entrée le signal **OK** produit à la question 2 et produisant, lorsque cette entrée passe à 1, un signal de sortie **S** qui ne vaudra 1 que pendant un seul cycle d'horloge.

7.2 *Proposition de correction*

Question 1 Détection de l'appui sur une touche

Il s'agit de détecter, d'un cycle d'horloge à l'autre, le changement de la valeur de **C** de la valeur $0xF$ à une valeur différente de $0xF$. On peut réaliser cela en deux étapes :

1. Créer un nouveau signal combinatoire **TA** pour "Touche Appuyée" à partir du signal **C**.
2. Mettre en place un détecteur de changement d'état (Il s'agit d'un problème classique de détection du passage de 0 à 1 d'un signal qui n'est pas une horloge. La réponse se trouve dans le cours, annexe C.8, page 117).

Nous aboutissons au schéma de la figure 7.2.

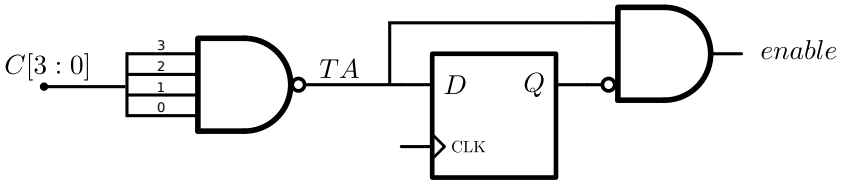


FIGURE 7.2: Détection de l'appui d'une touche sur le clavier

Question 2 Détection du code

La détection de la séquence fait appel à de la logique synchrone. Plusieurs solutions sont possibles faisant appel, ou non, à l'utilisation d'automates. Dans tous les cas notre structure passe par des états successifs correspondants à l'appui d'une touche du digicode. La solution la plus simple est (mais pas forcément la plus compacte) de placer les codes successifs dans un registre à décalage et de comparer chacune des sorties du registre avec le code désiré.

Nous proposons le schéma de la figure 7.3.

Nous utilisons des **bascules avec enable** (voir C.1.1, page 111), utilisant le signal **enable** créé précédemment. Ainsi elles ne sont mises à jour qu'à l'appui d'une touche du digicode. Le code courant, ainsi que les 3 sorties du registre à décalage sont comparés aux différents codes attendus pour former les signaux B_A, B_6, B_4 et B_2. Si ces quatre signaux sont à 1 alors l'utilisateur a bien tapé les codes successifs.

Question 3 Ouverture de la porte

Il s'agit une fois de plus d'un détecteur de passage de 0 à 1, tel que nous l'avons déjà utilisé dans la question 1.

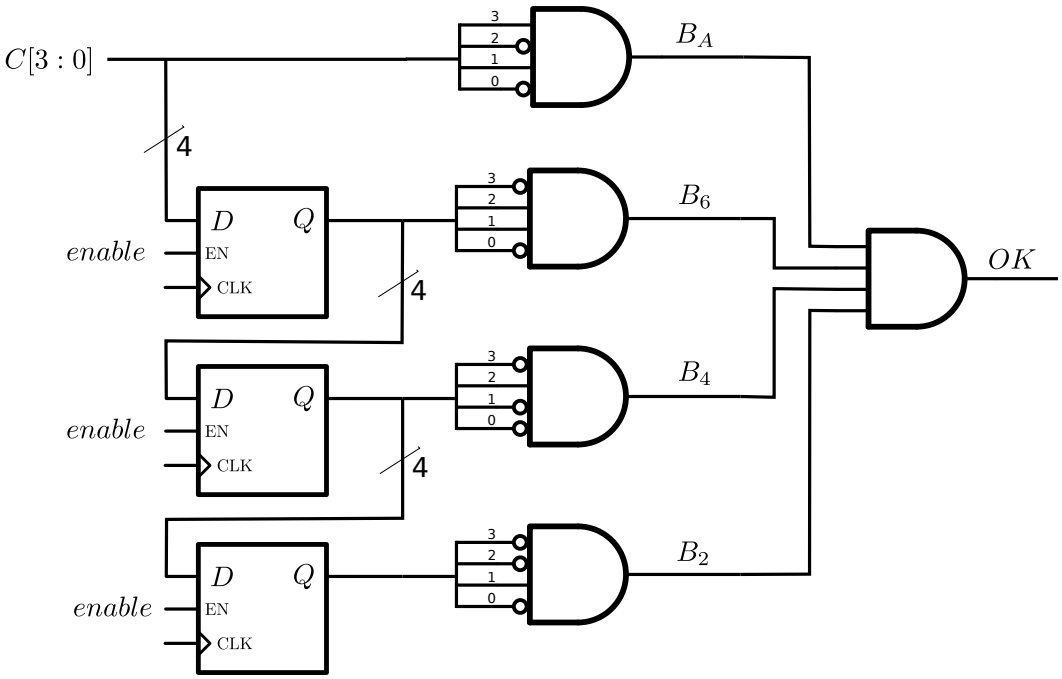


FIGURE 7.3: Détection de la séquence

8
Détection des touches d'un digicode

8.1 Enoncé du problème

L'objectif de l'exercice est de concevoir un système permettant de détecter quelle touche du clavier d'un digicode est appuyée.

Le système dispose d'une horloge **`clk`** à 10MHz et d'un signal d'initialisation **`reset_n`**, actif à l'état bas.

Vous disposez d'un clavier, dit "à balayage" comme décrit en figure 8.1. Le clavier dispose de deux sorties, **`data`** et **`scanning`** chacune sur un bit. Le clavier balaye en permanence les lignes et colonnes du clavier suivant la séquence suivante :

— Il balaye les 3 colonnes une par une, de la gauche vers la droite, et passe **`data`** à 1 quand il détecte une touche appuyée sur la colonne courante.

— Puis balaye les 4 lignes une par une en commençant par le haut, et passe **`data`** à 1 quand il détecte une touche appuyée sur la ligne courante.

— Le signal **`scanning`** passe à 1 pendant toute la durée du balayage.

— Entre deux balayages, **`scanning`** repasse à 0 pendant exactement un cycle d'horloge.

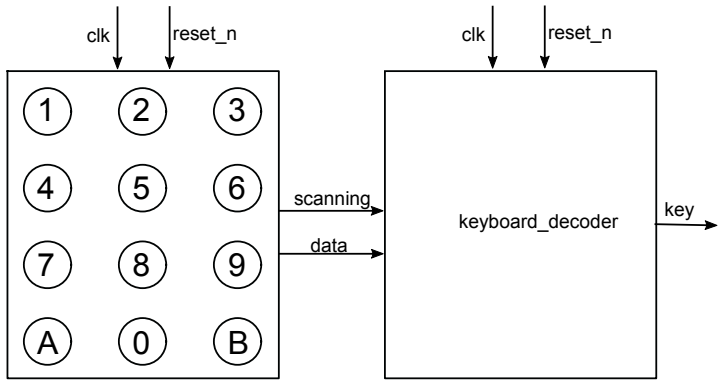


FIGURE 8.1: Digicode

Les figures suivantes montrent quelques cas de figure d'utilisation du clavier. Dans ces chronogrammes, la ligne "Étape" indique ce que le clavier est en train de balayer.

— Si on n'appuie sur aucune touche, on obtiendra le chronogramme de la figure 8.2.

— Si on appuie sur la touche 2 (colonne 1, ligne 0), on obtiendra le chronogramme de la figure 8.3.

— Si on appuie sur la touche B (colonne 2, ligne 3), on obtiendra le chronogramme de la figure 8.4.

Question 1 Interprétation des chronogrammes

Examinez le chronogramme de la figure 8.5. Qu'a fait l'utilisateur du clavier ?

FIGURE 8.2: Aucune touche

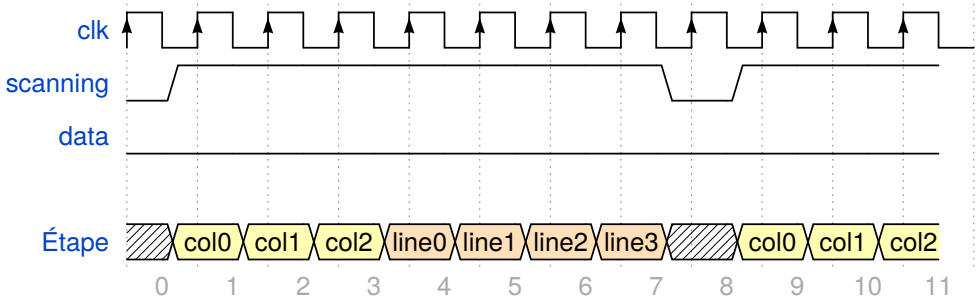


FIGURE 8.3: Touche 2

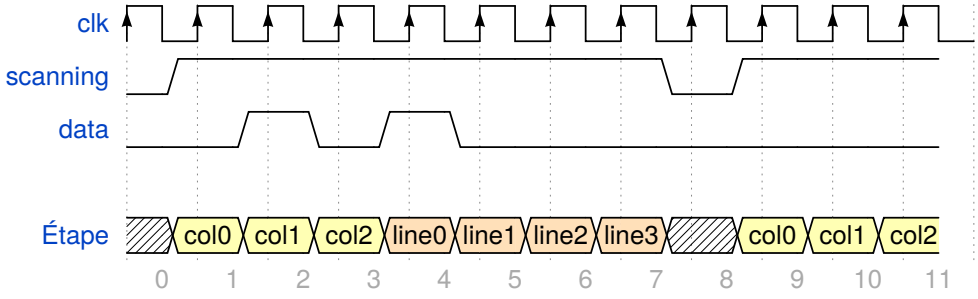


FIGURE 8.4: Touche B

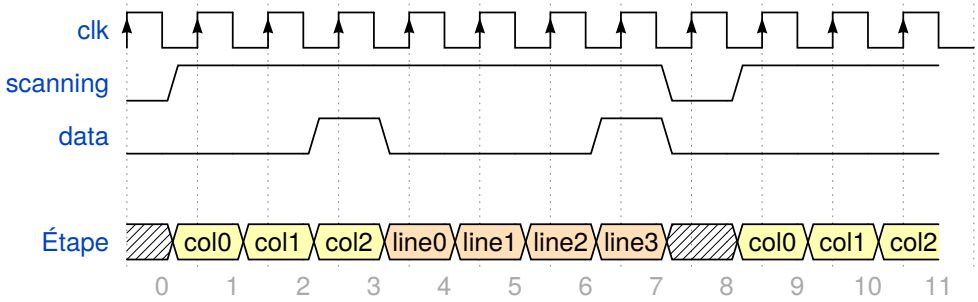
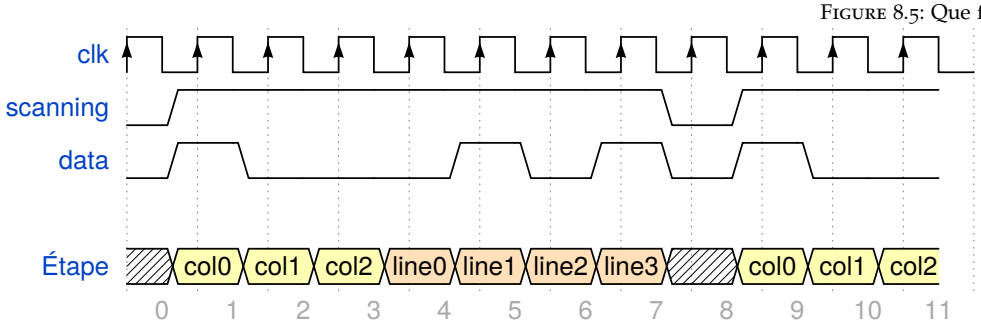


FIGURE 8.5: Que fait l'utilisateur ?



L'objectif de la suite est d'écrire le code SystemVerilog d'un système faisant en sorte que ce clavier ait le comportement suivant :

— Quand une touche est pressée, on sort son numéro sur le signal **`key`**.

— Quand aucun touche n'est pressée, ou quand plusieurs touches sont pressées à la fois, on sort **`0xF`**.

Dans la suite vous allez construire le code bloc par bloc. **Chaque bloc est simple, sans piège, et fait moins de 8 lignes**.
On partira du squelette de code 8.1.

```
module keyboard_decoder(input logic clk,
                        input logic reset_n,
                        input logic scanning,
                        input logic data,
                        output logic[3:0] key
                        );

   // Votre code ici !

endmodule
```

Code 8.1: Squelette du module

Question 2 Génération des étapes
On veut un "compteur d'étape" **`cpt`** qui vaut **`0`** par défaut, est incrémenté à la fin de chaque cycle où **`scanning`** vaut **`1`**, et remis à zéro à la fin d'un cycle où **`scanning`** vaut **`0`**.

— Jusqu'à combien doit pouvoir compter ce compteur ?

— Écrivez le code SystemVerilog d'un bloc séquentiel qui génère **`cpt`**.

Question 3 Stockage de la colonne
On veut stocker dans un signal **`col`** le numéro de la colonne de la touche pressée. Si aucune touche n'est appuyée, ou si plusieurs touches sont appuyées, on y stockera n'importe quoi.

— Sur combien de bits doit être codé **`col`** ?

— En remarquant que pour générer **`col`** il suffit de regarder **`data`** pendant que **`cpt`** est compris entre **`0`** et **`2`**, donnez le code SystemVerilog d'un bloc séquentiel qui génère **`col`**.

Question 4 Stockage de la ligne
On veut stocker dans un signal **`line`** le numéro de la ligne touche pressée. Si aucune touche n'est appuyée, ou si plusieurs touches sont appuyées, on y stockera n'importe quoi.

— Sur combien de bits doit être codé **`line`** ?

— En remarquant que pour générer **`line`** il suffit de regarder **`data`** pendant que **`cpt`** est compris entre **`3`** et **`6`**, donnez le code SystemVerilog d'un bloc séquentiel qui génère **`line`**.

Question 5 Détection d'un appui valide

Pendant que **`scanning`** est haut, combien de fois **`data`** est-il à **`1`** pendant le balayage si :

— l'utilisateur n'appuie sur aucune touche,

— l'utilisateur appuie sur une seule touche,

— l'utilisateur appuie sur plusieurs touches ?

Donnez le code SystemVerilog d'un bloc séquentiel qui génère un **`n_data`** indiquant le nombre de fois où **`data`** est passé à **`1`** pendant que **`scanning`** était haut.

Question 6 Génération de la sortie key

À partir des signaux générés prédécemment, on propose le code 8.2 pour générer la sortie **`key`**.

```
// Lorsque qu'on a finit le scan, on sort le numéro de la touche appuyée.
// Pour cela, il faut que n\_data ==... Sinon, ça veut dire qu'on n'a
// appuyé sur aucune touche ou qu'on a commencé à appuyer pendant que
// clavier scannait ou qu'on a appuyé sur deux touches en même temps.
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    // Au reset, on sort le code "aucune touche" : 0xF.
    key <= 0xF;
  // Lorsque le compteur indique qu'on a finit le scan,
  else if (cpt==8)
    // si on a bien eu l'appui d'une seule touche, on génère son code sur key.
    if (n_data == ...)
         // Pour les touches de 1 à A compris, c'est simple.
         key <= col + (3*line);
         // Pour 0 et B, c'est un cas particulier.
         if ((line == 3) & (col == 1))
             key <= 0;
         if ((line == 3) & (col == 2))
             key <= 0xB;
    else
      // sinon (n_data est invalide), on sort le code 0xF.
      key <= 4'hF;
```

CODE 8.2: Squelette du module

Corrigez ce code.

8.2 *Proposition de correction*

Question 1 Interprétation des chronogrammes

L'utilisateur appuie simultanément sur les touches **4** et **A**. De plus, il appuie suffisamment longtemps pour que le balayage suivant détecte une touche appuyée.

Question 2 Génération des étapes

Le compteur doit pouvoir compter jusqu'à **7**. Le code 8.3 correspond à ce compteur. Notez, la limitation du compteur à 3 bits, ce qui le ramène automatiquement à **0** a la fin de la séquence de **scanning** à **1**.

CODE 8.3: Code du compteur

```
logic [2:0] cpt ;
always @(posedge clk)
  if(!scanning)
    cpt <= '0 ;
  else
    if(scanning)
       cpt <= cpt + 1'b1 ;
```

Question 3 Stockage de la colonne

Le signal **col** est compris entre **0** et **2**. Il suffit de deux bits pour le coder. Comme indiqué dans le code 8.4, il suffit d'examiner les deux bits de poids faible de **cpt** pour obtenir **col**. On ne met à jour le signal **col** que si **data** vaut **1**.

CODE 8.4: Code pour la génération de **col**

```
logic [1:0] col ;
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    col <= '0 ;
  else if(data && (cpt < 3))
       col <= cpt[1:0] ;
```

Question 4 Stockage de la ligne

Le signal **line** est compris entre **0** et **3**. Il suffit de deux bits pour le coder. Comme indiqué dans le code 8.5, il faut de soustraire 3 au compteur **cpt** pour obtenir la valeur de **line**. On ne met à jour le signal **line** que si **data** vaut 1.

CODE 8.5: Code pour la génération de **line**

```
logic [1:0] line ;
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    line <= '0 ;
  else
    if(data && (cpt >= 3))
       line <= cpt - 2'd3 ;
```

Question 5 Détection d'un appui valide

1. Si l'utilisateur n'appuie sur aucune touche, data n'est jamais à **1**.
2. Si l'utilisateur appuie sur une seule touche, data passe à **1** deux fois
3. Si l'utilisateur appuie sur plus d'une touche, data passe à **1** au moins **3** fois.

Il suffit d'un compteur qui est remis à zero lorsque **scanning** vaut **0** et qui s'incrémente à chaque passage de **data** à **1**. Dans le pire des cas, si toutes les touches sont appuyées, **data** est à **1** durant **7** cycles consécutifs. Le compteur **n_data** peut donc être codé sur 3 bits.

```
logic [2:0] n_data ;
always @(posedge clk)
  if(!scanning)
    n_data <= '0 ;
  else if(data)
       n_data <= n_data+1'b1  ;
```

Code 8.6: Code pour la génération du **n_data**

Question 6 Génération de la sortie key

Nous suggérons les modifications suivantes :

— remplacer `if (n_data == ...)` par `if (n_data == 2'd2)` [1],

— remplacer `else if (cpt==8)` par `else if (cpt==7)` [2]

— remplacer la formule dur calcul de la clef par `key <= col + (3*line) + 1'b1`.

1. valeur attendue quand une seule touche est appuyée
2. Le compteur va de **0** à **7**. Le 8e cycle est donc le numéro **7**

Nous obtenons le code suivant :

```
always @(posedge clk or negedge reset_n)
  if(!reset_n)
    // Au reset, on sort le code "aucune touche" : 0xF.
    key <= 0xF;
  // Lorsque le compteur indique qu'on a finit le scan,
  else if (cpt==7)
    // si on a bien eu l'appui d'une seule touche, on génère son code sur key.
    if (n_data == 2)
         // Pour les touches de 1 à A compris, c'est simple.
         key <= col + (3*line)+1;
         // Pour 0 et B, c'est un cas particulier.
         if ((line == 3) & (col == 1))
             key <= 0;
         if ((line == 3) & (col == 2))
             key <= 0xB;
    else
      // sinon (n_data est invalide), on sort le code 0xF.
      key <= 4'hF;
```

Code 8.7: Module corrigé

9
Compteur de Gray

9.1 Enoncé du problème

Question 1 : Codez en SystemVerilog un compteur binaire sur 3 bits avec remise à zéro (reset) asynchrone.

On désire changer la séquence binaire $0 \Rightarrow 1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 5 \Rightarrow 6 \Rightarrow 7$ en séquence de Gray $0 \Rightarrow 1 \Rightarrow 3 \Rightarrow 2 \Rightarrow 6 \Rightarrow 7 \Rightarrow 5 \Rightarrow 4$.

Question 2 : Ajoutez (toujours en SystemVerilog) un bloc de logique combinatoire aux sorties du compteur binaire de façon à ce que les nouvelles sorties suivent cette séquence.

On veut maintenant générer directement la séquence de Gray . On nommera C_2^c, C_1^c, C_0^c les valeurs courantes des 3 bits du compteur de gray, et on nommera C_2^f, C_1^f, C_0^f les valeurs futures

Question 3 : Calculez les équations booléennes de C_2^f, C_1^f et C_0^f en fonction de C_1^c, C_1^c et C_0^c. Conseil : créez une table de vérité.

Question 4 : En déduire un code SystemVerilog du compteur de gray. Ce code ne doit comporter ni construction **if/then/else** ni construction **case**.

9.2 Proposition de correction

Question 1 : C'est un compteur synchrone classique, qui utilise exactement 3 bits. En dehors de toute autre indication on suppose qu'il compte modulo 8.

```
logic [2:0] binCpt ;
always @(posedge clk or posedge reset)
  if(reset)
    binCpt <= '0 ;
  else
    binCpt <= binCpt + 1  ;
```

Code 9.1: Le compteur binaire

Question 2 :

On nous demande de générer les codes de gray de manière combinatoire, il suffit d'une construction de type `case` dans un process `always@(*)` tel qu'indiqué dans le code 9.2.

```
logic [2:0] grayCpt ;
always @(*)
   case(binCpt)
      3'd0 : grayCpt <= 3'd0 ;
      3'd1 : grayCpt <= 3'd1 ;
      3'd2 : grayCpt <= 3'd3 ;
      3'd3 : grayCpt <= 3'd2 ;
      3'd4 : grayCpt <= 3'd6 ;
      3'd5 : grayCpt <= 3'd7 ;
      3'd6 : grayCpt <= 3'd5 ;
      3'd7 : grayCpt <= 3'd4 ;
   endcase
```

CODE 9.2: Le compteur de gray

Question 3 :

Dans la table 9.1, nous trions les entrées dans l'ordre des codes, et non pas dans l'ordre des transitions successives.

C_2^c	C_1^c	C_0^c	C_2^f	C_1^f	C_0^f	Transition
0	0	0	0	0	1	0 vers 1
0	0	1	0	1	1	1 vers 3
0	1	0	1	1	0	2 vers 6
0	1	1	0	1	0	3 vers 2
1	0	0	0	0	0	4 vers 0
1	0	1	1	0	0	5 vers 4
1	1	0	1	1	1	6 vers 7
1	1	1	1	0	1	7 vers 5

TABLE 9.1: Evolution des états du compteur de gray

Les situations où le bit C_0^f vaut 1 sont résumées dans la table suivante :

C_2^c	C_1^c	C_0^c	C_2^f	C_1^f	C_0^f	Transition
0	0	0	0	0	1	0 vers 1
0	0	1	0	1	1	1 vers 3
1	1	0	1	1	1	6 vers 7
1	1	1	1	0	1	7 vers 5

Cela aboutit à l'équation booléenne :

$$C_0^f = \overline{C_2^c} \cdot \overline{C_1^c} \cdot \overline{C_0^c} + \overline{C_2^c} \cdot \overline{C_1^c} \cdot C_0^c + C_2^c \cdot C_1^c \cdot \overline{C_0^c} + C_2^c \cdot C_1^c \cdot C_0^c$$

Nous remarquons que C_0^f vaut 1 si et seulement si C_1^c et C_2^c sont identiques. L'expression peut être simplifiée en :

$$C_0^f = \overline{C_2^c \oplus C_1^c}$$

Notez qu'il n'était pas demandé de simplifier les équations, les deux réponses étaient donc valables.

Les situations où le bit C_1^f vaut 1 sont résumées dans la table suivante :

C_2^c	C_1^c	C_0^c	C_2^f	C_1^f	C_0^f	Transition
0	0	1	0	1	1	1 vers 3
0	1	0	1	1	0	2 vers 6
0	1	1	0	1	0	3 vers 2
1	1	0	1	1	1	6 vers 7

Cela aboutit à l'équation booléenne :

$$C_1^f = \overline{C_2^c} \cdot \overline{C_1^c} \cdot C_0^c + \overline{C_2^c} \cdot C_1^c \cdot \overline{C_0^c} + \overline{C_2^c} \cdot C_1^c \cdot C_0^c + C_2^c \cdot C_1^c \cdot \overline{C_0^c}$$

Qui peut être simplifiée en :

$$C_1^f = \overline{C_2^c}.C_1^c + C_1^c.\overline{C_0^c} + \overline{C_2^c}.C_0^c$$

.

Enfin Les situations où le bit C_2^f vaut 1 sont résumées dans la table suivante :

C_2^c	C_1^c	C_0^c	C_2^f	C_1^f	C_0^f	Transition
0	1	0	1	1	0	2 vers 6
1	0	1	1	0	0	5 vers 4
1	1	0	1	1	1	6 vers 7
1	1	1	1	0	1	7 vers 5

Cela aboutit à l'équation booléenne :

$$C_2^f = \overline{C_2^c} \cdot C_1^c \cdot \overline{C_0^c} + C_2^c \cdot \overline{C_1^c} \cdot C_0^c + C_2^c \cdot C_1^c \cdot \overline{C_0^c} + C_2^c \cdot C_1^c \cdot C_0^c$$

Qui peut être simplifiée en :

$$C_2^f = C_2^c.C_1^c + C_2^c.C_0^c + C_1^c.\overline{C_0^c}$$

.

Question 4 :

Les codes sont directement générés par un processus synchrone à partir des équations précédentes :

```
logic [2:0] grayCpt ;
always @(posedge clk or posedge reset)
  if(reset)
      grayCpt <= '0 ;
  else begin
      grayCpt[0] <= ~( grayCpt[2] ^  grayCpt[1])  ;
      grayCpt[1] <=  (~grayCpt[2] &  grayCpt[1]) |
                     ( grayCpt[1] & ~grayCpt[0]) |
                     (~grayCpt[2] &  grayCpt[0])  ;
      grayCpt[2] <=  (~grayCpt[2] &  grayCpt[1]) |
                     ( grayCpt[1] &  grayCpt[0]) |
                     ( grayCpt[1] & ~grayCpt[0])  ;
  end
```

CODE 9.3: Le compteur de gray directement généré

10
Arithmétique avec représentations redondantes

10.1 Enoncé du problème

L'étude de la représentation des nombres dans les machines a fait l'objet de nombreux travaux de recherche. Nous allons étudier une notation dite de type Avizienis en base 2. Dans cette représentation, nous appellerons les chiffres des **Rbits**. Un nombre **A** de **N** Rbits se code sous la forme suivante :

$$A = \sum_{i=0}^{N-1} a_i 2^i \; avec \; a_i \in \{-1, 0, 1\}$$

Enfin, pour faciliter l'écriture de constantes on écrira les valeurs $\{-1, 0, 1\}$ sous la forme $\{\bar{1}, 0, 1\}$.

Question 1 Quelle est la valeur décimale du nombre $1\bar{1}00\bar{1}01$?

Question 2 Expliquez pourquoi cette représentation est qualifiée de redondante, proposez quelques exemples.

Question 3 Quelles sont les valeurs minimales et maximales pouvant être atteintes par un mot de N Rbits.

Dans la pratique, réaliser du matériel utilisant cette représentation est complexe (un transistor est soit passant, soit bloqué...). On va donc devoir coder un Rbit sur 2 bits. Pour cela on utilise la convention suivante :

$$a_i = a_i^+ - a_i^- \; avec \; a_i^+ \in \{0, 1\} \; et \; a_i^- \in \{0, 1\}$$

La figure 10.1 représente une structure d'addition pour deux nombres A et B de 4 Rbits. Elle est composée d'un assemblage de cellules de type **PMP** et **MMP** qui sont dérivées de l'additionneur 1 bit de base. Les sorties **R** et **S** étant des bits de retenue et de somme. Nous ne chercherons pas à démontrer son fonctionnement.

Question 4 En supposant que le temps de calcul d'une cellule PMP ou MMP soit de 1 dans une unité arbitraire, déterminez le temps de calcul de l'additionneur.

Question 5 Quel serait le temps de calcul d'un additionneur de mots de N Rbits ? Comparez ce résultat à celui de l'additionneur a propagation de retenue N bits. Qu'en pensez-vous ?

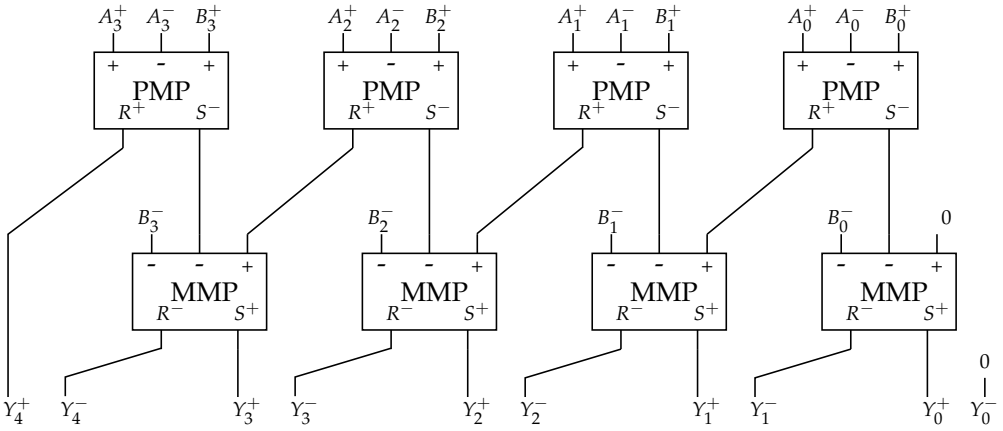
Supposons maintenant que l'on construise un Nanoprocesseur utilisant la notation redondante pour l'arithmétique. Seule L'ALU du Nanoprocesseur est modifiée. Le nombre de bits des données et le reste de l'architecture reste inchangé.

Question 6 Quel est le nombre de Rbit des données traitées ?

L'ALU du nanoprocesseur doit fournir un drapeau **Z** indiquant si le résultat du calcul est nul.

Question 7 Faites le schéma de la logique nécessaire à la génération du drapeau **Z** à partir d'un calcul en codage redondant.

FIGURE 10.1: Additionneur en codage redondant



Le nanoprocesseur peut exécuter une instruction **ADD** ou une instruction **SUB**.

Question 8 Quel matériel faut-il ajouter à la figure 10.1 pour que l'additionneur puisse arbitrairement calculer une addition ou une soustraction.

Pour comparer deux nombres, on utilise classiquement l'instruction **SUB** du Nanoprocesseur et on examine la retenue sortante en utilisant le drapeau **C**.

Question 9 Est-ce toujours possible avec notre nouvelle notation ? Faut-il rajouter du matériel à l'ALU pour obtenir cette information ? Si votre réponse est positive, détaillez ce matériel et dans ce cas que pensez-vous de l'intéret de la représentation redondante (dans le cas du Nanoprocesseur) ?

10.2 Proposition de correction

Question 1 :

En tenant compte du poids de chaque *Rbit* le nombre vaut : $64 - 32 - 4 + 1 = 29$.

Question 2 :

Un même nombre peut avoir plusieurs représentations. Par exemple le nombre 15 peut être codé sous la forme $1\overline{1}000\overline{1}$ ou 001111 ou encore sous la forme $01000\overline{1}$.

Question 3 :

La valeur minimale est atteinte lorsque tous les *Rbits* valent $\overline{1}$, ce qui correspond au nombre $-(2^n - 1)$. La valeur maximale est atteinte lorsque tous les *Rbits* valent 1, ce qui correspond au nombre $2^n - 1$.

Question 4 :

Le nombre maximal de cellules traversées entre une entrée quelconque du dispositif, et une sortie quelconque est de 2. Le temps de calcul de l'additionneur est de 2.

Question 5 :

On voit qu'il n'y a pas de propagation de retenue latérale dans la structure, le temps de calcul est indépendant du nombre de *Rbits* des données. Le temps de calcul d'une addition à propagation de retenue est en O(N). On remarque donc que cette structure d'addition semble plus intéressante que la classique propagation de retenue.

Question 6 :

Nous devons utiliser 2 bits pour représenter un *Rbit*, le chemin de données du Nanoprocesseur est de 8bits. En conséquence nous ne pouvons manipuler que des mots de 4 *Rbits*.

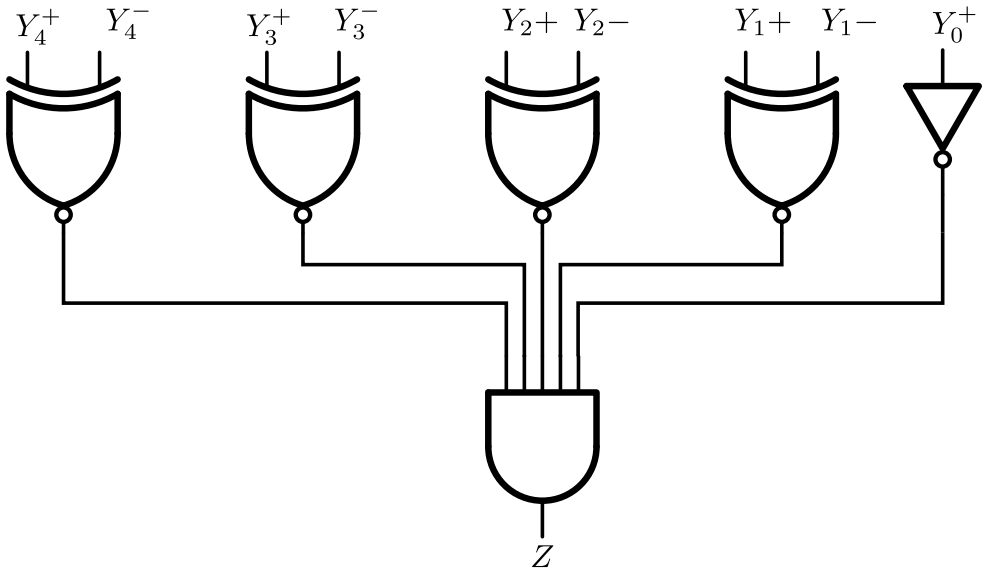
Question 7 :

Pour détecter si le résultat est nul, il faut détecter que chaque *Rbit* du résultat est nul. Compte tenu de la redondance, 2 cas sont possibles pour chaque Y_i :

1. $Y_i^+ = 0$ et $Y_i^- = 0$
2. $Y_i^+ = 1$ et $Y_i^- = 1$

Cela correspond à la fonction logique *XNOR*, sauf pour le bit Y_0 pour lequel le calcul se réduit à un simple inverseur. Nous en déduisons le schéma de la figure 10.2

FIGURE 10.2: Calcul du bit Z à partir du résultat en notation redondante



Question 8 :

Nous voulons calculer $A - B$. En notation redondante, pour obtenir $-B$ il suffit d'échanger les bits B_i^+ et les bits B_i^-. Pour chaque *Rbit* de B nous utiliserons donc 2 multiplexeurs commandés par le signal demandant la soustraction.

Question 9 :

En notation redondante, il n'est pas possible de détecter si le résultat d'un calcul est positif ou négatif sans examiner tous les *Rbits* un à un. Une solution est de calculer en arithmétique *standard* la différence entre les deux mots binaires Z^+ et Z^-. On obtient ainsi un nombre dont on peut déterminer le signe. Mais cela revient à ajouter un soustracteur derrière l'additionneur *standard* ce qui fait perdre l'intéret du temps de calcul constant...

11
Fibonacci ?

La résolution de ce problème, nécessite la compréhension des chapitres sur la logique synchrone, ainsi que des techniques de codage de base SystemVerilog associées.

11.1 Enoncé du problème

Un concepteur prétend que le code SystemVerilog suivant génère la suite de Fibonnacci en calculant sur 8 bits les nombres U_n. On rappelle que la suite de Fibonacci est donnée par : $U_n = U_{n-1} + U_{n-2}$, avec $U_1 = 1$ et $U_0 = 0$.

Code 11.1: Code Fibonacci proposé

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

   // Un-1 (= Un au cycle precedent)
   logic [7:0]        U_1;

   // Un-2 (= Un-1 au cycle precedent)
   logic [7:0]        U_2;

   always @(posedge clk or negedge reset_n)
     if(!reset_n) begin
          U   <= 1;
          U_1 <= 1;
          U_2 <= 0;
       end
     else begin
          // Un = Un-1 + Un-2
          U    <= U_1 + U_2;
          // Un-1 = Un retarde d'un cycle
          U_1  <= U;
          // Un-2 = Un-1 retarde d'un cycle
          U_2  <= U_1;
       end
endmodule
```

1. Réalisez un tableau des valeurs de **`U`**, **`U_1`** et **`U_2`** pendant les 10 premiers cycles d'horloge (après relâchement du **`reset_n`**).
2. Tracez le schéma du circuit modélisé par ce code (l'horloge et le signal d'initialisation ne seront pas représentés).
3. Quelle est l'équation de la suite U_n générée ?
4. Proposez une version corrigée du code SystemVerilog de façon à générer la suite de Fibonacci.

11.2 Proposition de correction

Question 1 :

Pour interpréter correctement un code SystemVerilog, il faut examiner chaque signal faisant l'objet d'une affectation. En général la forme d'écriture est de type : **`signal <= équation`**.

— Si **`signal`** est affecté dans un processus combinatoire **always@(*)**, alors il prend la valeur **`équation`** via un calcul combinatoire.

— Si **`signal`** est affecté dans un processus synchrone **always@(posedge...)**, alors il correspond à un **registre** et prend, **après** le front d'horloge, la valeur qu'avait **`équation`** **avant** le front de l'horloge.

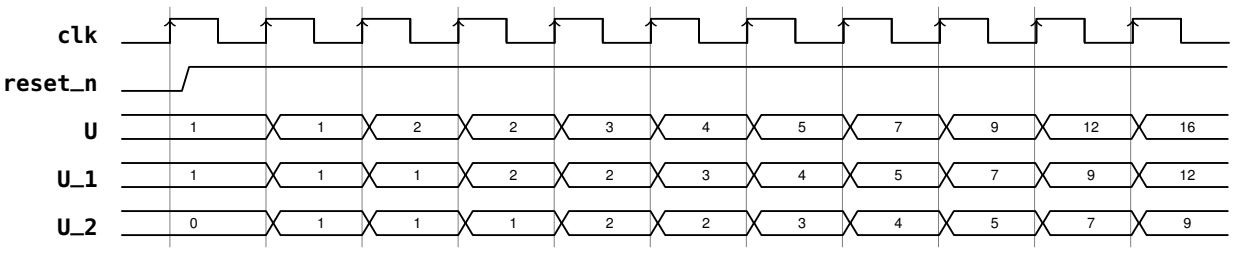
En conséquence, du point de vue d'un chronogramme, le signal généré aura les caractéristiques suivantes :

— Si le signal est **combinatoire** on indiquera ses évolutions temporelles comme une simple conséquence des signaux qui ont servi à la calculer, en ajoutant éventuellement un petit délai pour montrer clairement cette conséquence.

— Si le signal est **synchrone** on indiquera ses évolutions temporelles comme une simple conséquence du front de l'horloge, en ajoutant éventuellement un petit délai pour montrer clairement cette conséquence.

Dans notre cas, les signaux **`U_1`**, **`U_2`** et **`U`** sont tous calculés dans un processus synchrone, d'où le chronogramme :

FIGURE 11.1: Chronogramme de fonctionnement (Question 1)



Question 2 :

Comme indiqué dans la question, nous ne représentons ni l'horloge, ni le signal de remise à zéro, dans le schéma. Cependant, nous indiquons clairement que les signaux **`U`**, **`U_1`** et **`U_2`** sont générés par des registres synchrones (le triangle en bas à gauche correspond à connection d'horloge). De plus nous indiquons en bas à droite de chaque registre la valeur forcée à l'initialisation. Nous indiquons de plus le nombre de bits des signaux (trait barrant le signal accompagné d'un nombre).

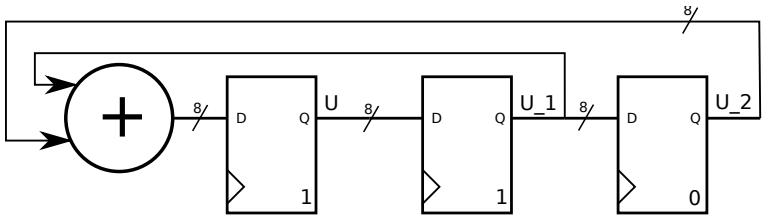


FIGURE 11.2: Schéma équivalent (Question 2)

Question 3 :

Le signal **U** est retardé d'un cycle excédentaire.

La véritable suite générée est : $U_n = U_{n-2} + U_{n-3}$, avec $U_0 = 1$, $U_1 = 1$ et $U_2 = 0$.

Question 4 :

Pour créer le schéma, il suffit de 2 registres au lieu de 3. Si nous ne conservons que les registres **U_1** et **U_2** et nous transformons le signal **U** en signal combinatoire. Nous aurons le comportement suivant à chaque front d'horloge :

— Juste avant le front d'horloge :

  — **U_1** et **U_2** contiennent respectivement les valeurs U_{n-1} et U_{n-2}

  — **U** est établit combinatoirement à la valeur $U_n = U_{n-1} + U_{n-2}$

— Juste après le front d'horloge :

  — **U_1** et **U_2** contiennent respectivement les valeurs U_n et U_{n-1}

  — Le signal **U** commence le calcul combinatoire de $U_{n+1} = U_n + U_{n-1}$

Le nouveau code peut être le suivant :

CODE 11.2: Code Fibonacci première solution (Question 4)

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

   logic [7:0]        U_1;
   logic [7:0]        U_2;

   always @(*)
          U    <= U_1 + U_2;

   always @(posedge clk or negedge reset_n)
     if(!reset_n) begin
          U_1 <= 1;
          U_2 <= 0;
       end
     else begin
          U_1  <= U;
          U_2  <= U_1;
       end
endmodule
```

Une écriture plus compacte consiste à intégrer le calcul combinatoire dans le processus synchrone. Il suffit de renommer les signaux, et de supprimer le signal **U_2**. (Les deux solutions sont exactement équivalentes du point de vue "matériel").

CODE 11.3: Code Fibonacci deuxième solution (Question 4)

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

   logic [7:0]        U_1;

   always @(posedge clk or negedge reset_n)
     if(!reset_n) begin
          U   <= 1;
          U_1 <= 0;
       end
     else begin
          U    <= U + U_1;
          U_1  <= U;
       end
endmodule
```

12
Etude d'une fonction de rendez-vous

La résolution de ce problème nécessite la compréhension des notions de temps de propagation dans la logique combinatoire (premier chapitre) ainsi que la maîtrise de techniques de constructions de portes logiques (chapitre technologie).

12.1 Enoncé du problème

Nous désirons réaliser une fonction logique **RV** à deux entrées **A** et **B** et une sortie **Y**.

12.1.1 Analyse d'un version de RV basée sur l'assemblage de portes logiques de base

Le schéma de la **figure 12.1** présente une réalisation **RV** à partir de portes logiques **ET** et **OU** dont les caractéristiques sont les suivantes :

— Les portes **ET** et **OU** ont **toutes** un temps de propagation de $1ns$.

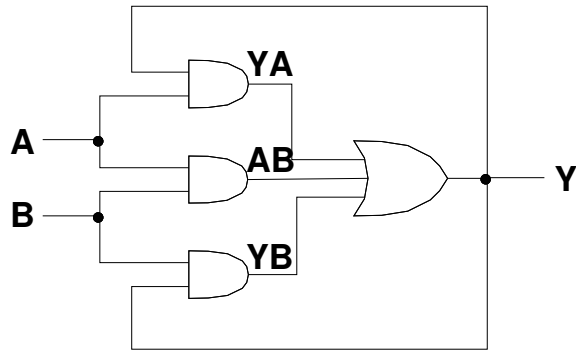


FIGURE 12.1: Schéma de la fonction **RV**

Question 1 : Complétez, de manière précise (avec les temps de propagation), le chronogramme de la **figure 12.3**.
Question 2 : Démontrez, en utilisant le chronogramme, que la fonction **RV** est une fonction séquentielle.
Question 3 : Pourquoi appelle-t-on cette fonction, fonction de "rendez-vous" ?

12.1.2 Analyse d'une version de RV basée sur une fonction CMOS spécifique

La **figure 12.2** présente une alternative pour la réalisation de la fonction **RV**. Pour analyser le schéma on supposera

— Qu'un transistor bloqué a une résistance équivalente infinie

— Qu'un transistor passant a une résistance équivalente nulle

— Que l'entrée d'un inverseur est électriquement équivalente à une capacité connectée entre cette entrée et la masse.

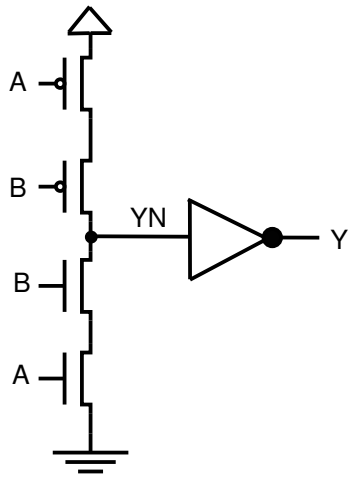


FIGURE 12.2: Schéma alternatif pour la fonction **RV**

Question 4 : Complétez le chronogramme de la **figure 12.4** en indiquant l'état des réseaux de transistors **N** et **P** (**B**loqué ou **P**assant) dans chaque situation. Les temps de propagations seront considérés comme **nuls**.

Question 5 : Que se passe-t-il lorsque le réseau **P** et le réseau **N** sont tous les deux bloqués ?

On utilise maintenant un modèle plus réaliste des transistors :

— Même dans l'état bloqué, la résistance équivalente des transistors n'est pas infinie.

Question 6 : Quelles sont les conséquences sur les conditions limites d'utilisation de cette porte ?

FIGURE 12.3: Chronogramme de la Question 1 (**les temps indiqués sont en** *ns*)

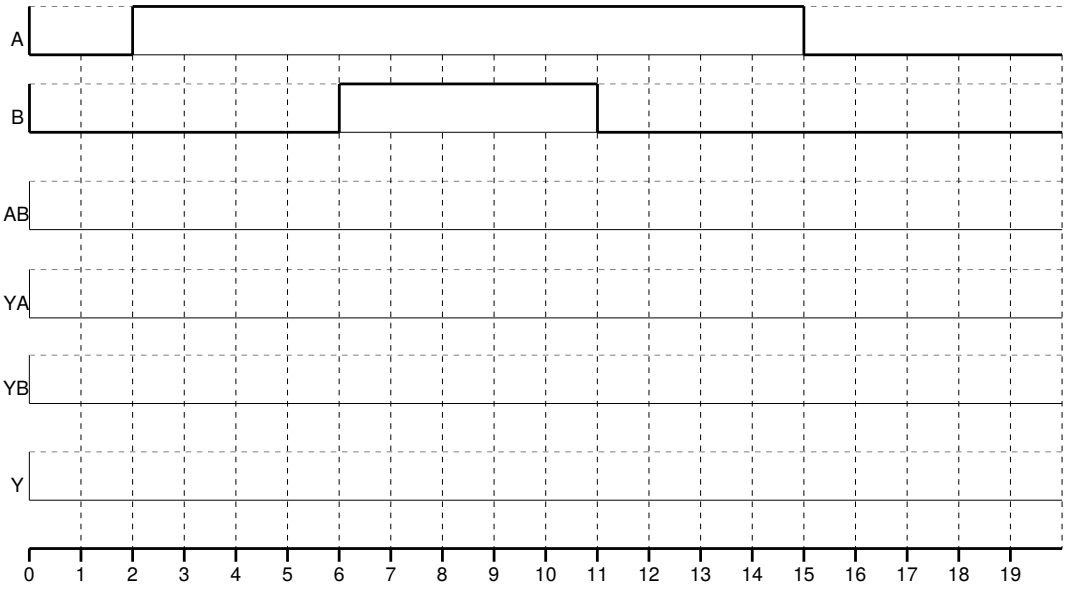
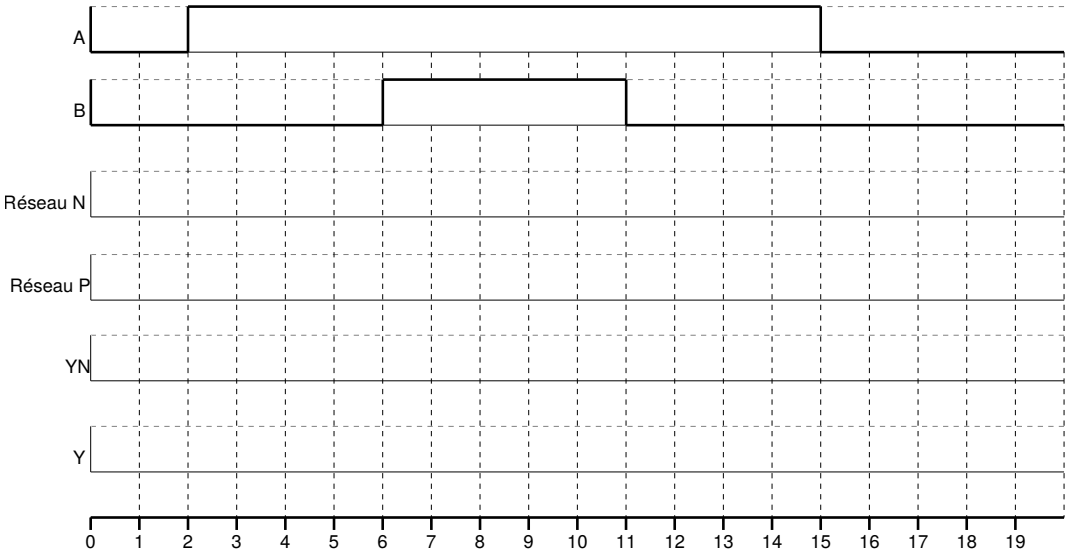


FIGURE 12.4: Chronogramme de la Question 4 (**les temps indiqués sont en** *ns*)



12.2 *proposition de correction*

Question 1 :

On demande dans l'énoncé un chronogramme précis, c'est-à-dire qu'il faut reporter les temps avec précision. Ainsi, par exemple, si un changement du signal A provoque un changement du signal YA il faut indiquer ce changement $1ns$ après le changement de A. On peut commencer par tracer le signal AB qui ne dépend que de A et de B.

L'enchainement des situations est le suivant :

— Tant que le signal AB n'est pas égal à 1 le signal Y ne peut pas quitter la valeur 0. Il en est de même pour les signaux YA et YB. On peut donc tracer les signaux Y, YA et YB comme constants à zéro jusqu'à l'instant $7ns$.

— Le signal Y passe à 1 à l'instant $8ns$ a cause du changement de AB. En conséquence les signaux YA et YB passent à 1 à l'instant $9ns$.

— Le signal YB repasse à 0 $1ns$ après la redesente de B. Mais le signal Y est maintenu car YA ne bouge pas.

— Le signal YA repasse à 0 $1ns$ après la redescente de A.

— Le signal Y repasse à 0 $1ns$ après la redescente de YA

Question 2 :

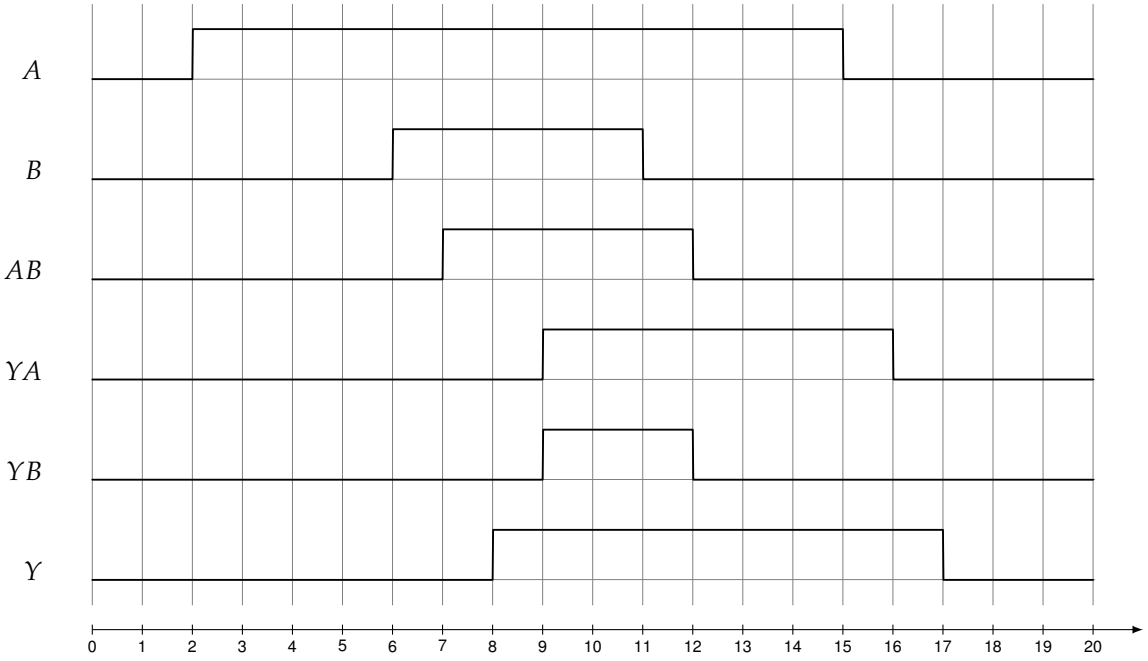
Les entrées du schéma sont les signaux A et B, la sortie est le signal Y.

Entre les instants $2ns$ et $6ns$ les deux entrées A et B sont stables et égales à 1 et 0 respectivement. La sortie Y est stable et maintenue à 0.

Entre les instants $11ns$ et $15ns$ les deux entrées A et B sont stables et de nouveau égales à 1 et 0 respectivement. La sortie Y est stable et maintenue à 1.

Nous avons donc un dispositif qui génère deux valeurs de sorties différentes pour des valeurs d'entrées identiques. Cela signifie que la sortie dépend de l'historique du changement des entrées, c'est donc un dispositif séquentiel.

FIGURE 12.5: Chronogramme de la réponse à la Question 1



Question 3 :

— Lorsque les deux entrées A et B valent 0 la sortie Y passe à 0.

— Lorsque les deux entrées A et B valent 1 la sortie Y passe à 1.

— Lorsque les deux entrées sont de valeurs complémentaires (01 ou 10) , la sortie conserve son état.

Ainsi la fonction détecte le rendez-vous de deux valeurs identiques de A et B, via le signal Y, et conserve cette valeur jusqu'au prochain rendez-vous.

Question 4 et Question 5 :
Rappelons que dans le modèle logique que nous utilisons :

— Les transistor PMOS sont passants si leur grille est pilotée par un 0 logique

— Les transistors NMOS sont passants si leur grille est pilotée par un 1 logique.

Les transistors se comportent comme des interrupteurs connectés (dans ce montage) en série

— Le réseau NMOS sera passant si les deux transitors du réseau NMOS sont passants.

— Le réseau PMOS sera passant si les deux transitors du réseau PMOS sont passants.

Après examen des situations nous obtenons 3 cas possibles :

— Le réseau NMOS est passant, le réseau PMOS est bloqué : le signal YB est forcé à 0

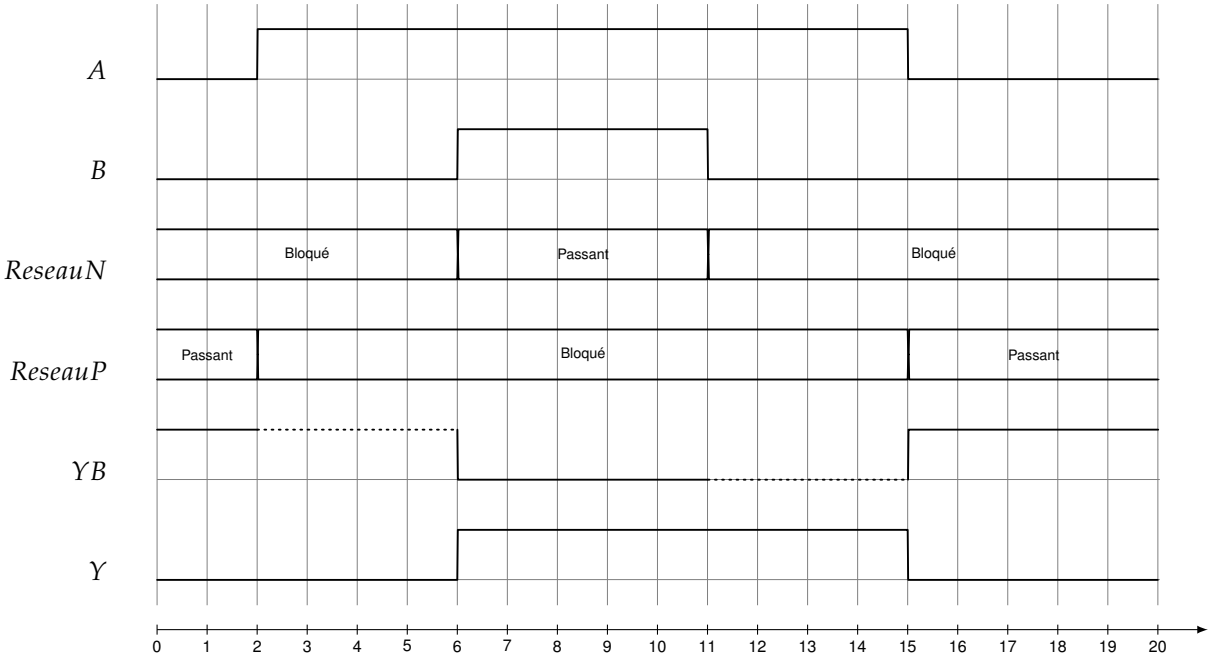
— Le réseau PMOS est passant, le réseau NMOS est bloqué : le signal YB est forcé à 1

— Les réseaux NMOS et PMOS sont tous les deux bloqués : le noeud YB n'est plus forcé à une quelconque valeur. L'énoncé indique que l'entrée de l'inverseur est équivalente à une capacité : cette capacité conserve son état (chargé ou déchargé) et maintient donc la valeur du signal YB

.

La fonction de mémorisation nécessaire à la fonction de rendez-vous est obtenue par un stockage capacitif de l'état du signal YB.

FIGURE 12.6: Chronogramme de la réponse à la Question 4



Question 6 :

Des courants de fuites peuvent traverser les transistors PMOS et NMOS. Ansi la charge stockée en YA va évoluer dans le temps et provoquer un changement d'état de l'inverseur de sortie. L'information de "rendez-vous" ne peut être conservée indéfiniment, à la différence du premier montage.

13
Comptage de moutons

La résolution de ce problème nécessite la maîtrise de la logique combinatoire, de la logique séquentielle et des techniques de codage SystemVerilog associées.

13.1 Enoncé du problème

Notes :

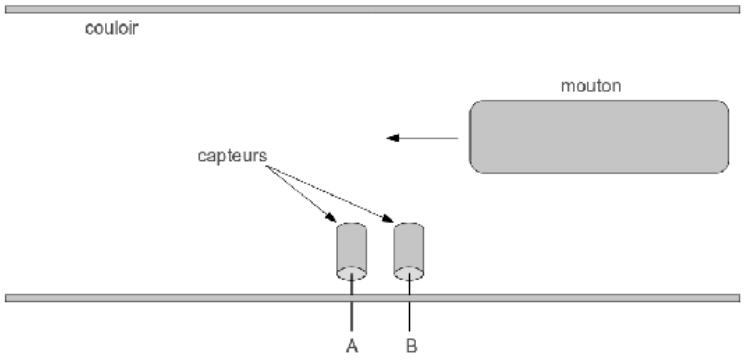
— Toutes les bascules de vos circuits seront synchrones sur une horloge **clk** et **uniquement sur celle-ci**. Elles seront, de plus, initialisées par un signal **reset_n** asynchrone actif à l'état bas (**0**).

— Tous les signaux utilisés devront être explicitement déclarés.

— La taille (nombre de bits) de tous les signaux utilisés devra être explicitement déclarée.

— Il sera inutile de déclarer les entêtes (**module xxx(input...**), et fin (**endmodule**) des modules.

On cherche à compter les moutons d'un troupeau (comportant moins de 2^{32} moutons). Pour cela, les moutons passent l'un après l'autre dans un couloir où sont placés deux détecteurs. Chaque détecteur renvoie un signal sur 1 bit, valant **1** lorsqu'un mouton passe en face de lui et **0** sinon.



Dispositif de comptage

Les deux détecteurs, A et B, sont éloignés d'environ 10cm : lors du passage d'un mouton, il y a donc un moment durant lequel les deux détecteurs voient en même temps le mouton : $A = B = 1$.

On supposera que les sorties des détecteurs sont synchrones sur l'horloge **clk** de votre circuit, et que l'horloge va suffisamment vite pour qu'il s'écoule plusieurs cycles entre deux événements successifs sur le bus de sortie des capteurs.

13.1.1 Question 1

On suppose pour l'instant que les moutons ne font qu'avancer de la droite vers la gauche.

En vous aidant éventuellement d'un chronogramme (réalisé par vos soins) des sorties des capteurs lors du passage d'un mouton, donnez le code SystemVerilog d'un système comptant le nombre de moutons passés dans le couloir.

13.1.2 Question 2

Les moutons étant des animaux capricieux, une fois arrivés tout au bout du couloir ils peuvent faire demi-tour et revenir dans l'enclos de départ. Il faut dans ce cas les dé-compter !

Donnez une **nouvelle** version de votre code de façon à pouvoir maintenir le compte correct des moutons arrivés à destination.

13.1.3 Question 3 (BONUS)

L'imagination des moutons étant sans bornes, ils peuvent avoir envie de faire demi-tour au moment précis où ils passent devant les détecteurs.

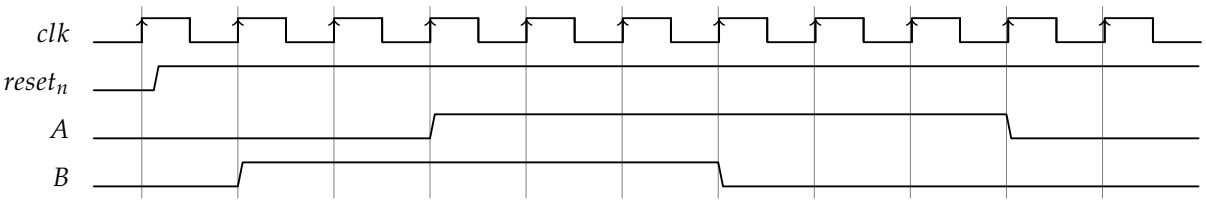
Si nécessaire, donnez une nouvelle version de votre code, de façon à prendre en compte le fait que les moutons peuvent faire demi-tour n'importe quand.

13.2 proposition de correction

13.2.1 Question 1

Nous construisons un chronogramme représentant l'évolution de l'horloge *clk* et des signaux A et B pendant le passage d'un mouton. Les signaux A et B restent stables pendant plus d'un cyle (comme indiqué dans l'énoncé). Attention, le nombre de cycles pendant lesquels A et B sont stables est dépendant de la vitesse de déplacement du mouton, cependant ce nombre de cycles n'intervient pas dans la résolution du problème. Enfin, le mouton provenant de la droite, le capteur B est actionné avant le capteur A.

FIGURE 13.1: Chronogramme d'un mouton (Question 1)



Pour résoudre ce problème, on remarque que le passage de `1` à `0` du signal A suffit à signaler la sortie d'un mouton. Donc nous devons simplement implémenter un détecteur de passage de `1` à `0`. Le signal généré servira à valider l'incrémentation d'un compteur synchrone. Le signal *reset_n* servira à initialiser le compteur. Enfin, le troupeau étant limité à moins de 2^{32} moutons, nous pouvons implémenter le compteur sur 32 bits.

D'où le code suivant :

Code 13.1: Code proposé pour la question 1

```
// On suppose A et B préalablement déclarés
...
// Le détecteur de changement d'état de A
logic last_A ;
always@(posedge clk) last_A <= A ;

logic un_mouton_de_plus  ;
always@(*) un_mouton_de_plus <= last_A && !A ;

// Le compteur lui même
logic [31:0] cmpt_moutons ;
always @(posedge clk or negedge reset_n)
  if(!reset_n)
     cmpt_moutons <= '0 ;
  else
    if (un_mouton_de_plus)
      cmpt_moutons <= cmpt_moutons+1'b1 ;
// ...
```

13.2.2 *Question 2*

Comme le mouton peut revenir en arrière il faut pourvoir détecter le sens de déplacement. Si on considère le passage de `1` à `0` du capteur A :

— B vaut `0` si le mouton se déplace de droite à gauche et

— B vaut `1` si le mouton se déplace de gauche à droite.

Le code devient :

Code 13.2: Evolution du code pour la question 2

```
...
// Le détecteur de changement d'état de A
logic last_A ;
always@(posedge clk) last_A <= A ;

logic un_mouton_de_plus  ;
always@(*) un_mouton_de_plus <= !B && (last_A && !A) ;

logic un_mouton_de_moins  ;
always@(*) un_mouton_de_moins <= B && (last_A && !A) ;

// Le compteur lui même
logic [31:0] cmpt_moutons ;
always @(posedge clk or negedge reset_n)
  if(!reset_n)
     cmpt_moutons <= '0 ;
  else  begin
    if (un_mouton_de_plus) cmpt_moutons <= cmpt_moutons+1'b1 ;
    if (un_mouton_de_moins) cmpt_moutons <= cmpt_moutons-1'b1 ;
  end
// ...
```

13.2.3 *Question 3*

Le code précédent ne fonctionne pas pour un mouton capricieux : il ne faut pas décompter un mouton qui n'a pas été compté, ni recompter un mouton déjà compté. Pour résoudre ce problème, il ne faut valider le décrément que si on a préalablement réalisé un incrément. Il faut donc mémoriser l'information de l'incrément, puis la réinitialiser à chaque fois qu'un nouveau mouton rentre dans le couloir

Le code pourrait être :

CODE 13.3: Evolution du code pour la question 3

```
// ...
// Détection de l'arrivée d'un mouton
logic un_nouveau_mouton ;
always @(*)
    un_nouveau_mouton <= B && (!last_A && A) ;

// Mémorisation et réinitilisation de l'information indiquant qu'un
// mouton a été compté.
always @(posedge clk or negedge reset_n)
  if(!reset_n)
     deja_compte <= '0 ;
  else begin
     if(un_mouton_de_plus) deja_compte <= 1'b1 ;
     if(un_nouveau_mouton) deja_compte <= 1'b0 ;
  end

// On ne valide l'incrément du compteur pour ce mouton que si
// on ne l'a pas déjà compté
always@(*)
    un_mouton_de_plus <= !B && (last_A && !A) && !deja_compte  ;

// On ne valide le décrément compteur pour ce mouton que si
// on l'a déjà compté
always@(*)
    un_mouton_de_moins <= B && (last_A && !A) && deja_compte  ;

// ...
```