**Table des matières**

[présentation hardware de structure CPU 5](#__RefHeading___Toc1700_3587549001)

[TP 7](#__RefHeading___Toc1702_3587549001)

[VHDL 7](#__RefHeading___Toc1704_3587549001)

[MOSFET 10](#__RefHeading___Toc1706_3587549001)

[LUT (Look Up Table : Table de vérité) : 11](#__RefHeading___Toc1708_3587549001)

[Utilisation de git 12](#__RefHeading___Toc1710_3587549001)

[Logique numérique synchrone 12](#__RefHeading___Toc1712_3587549001)

[La bascule 12](#__RefHeading___Toc1714_3587549001)

[registre 13](#__RefHeading___Toc1716_3587549001)

[Les compteurs 14](#__RefHeading___Toc1718_3587549001)

[Compteurs de 4 bit 14](#__RefHeading___Toc1720_3587549001)

[Machine à état (FSM : Finale State Machine) 15](#__RefHeading___Toc1358_4277434356)

[Types machines à état 15](#__RefHeading___Toc1360_4277434356)

[gestion de timing 16](#__RefHeading___Toc1362_4277434356)

[cas particulier 17](#__RefHeading___Toc1409_3391481760)

[clock shew : 18](#__RefHeading___Toc1411_3391481760)

[Entrées/sorties dans FPGA 18](#__RefHeading___Toc1413_3391481760)

[Single ended 18](#__RefHeading___Toc1415_3391481760)

[interface de transmission, différentiel 19](#__RefHeading___Toc1417_3391481760)

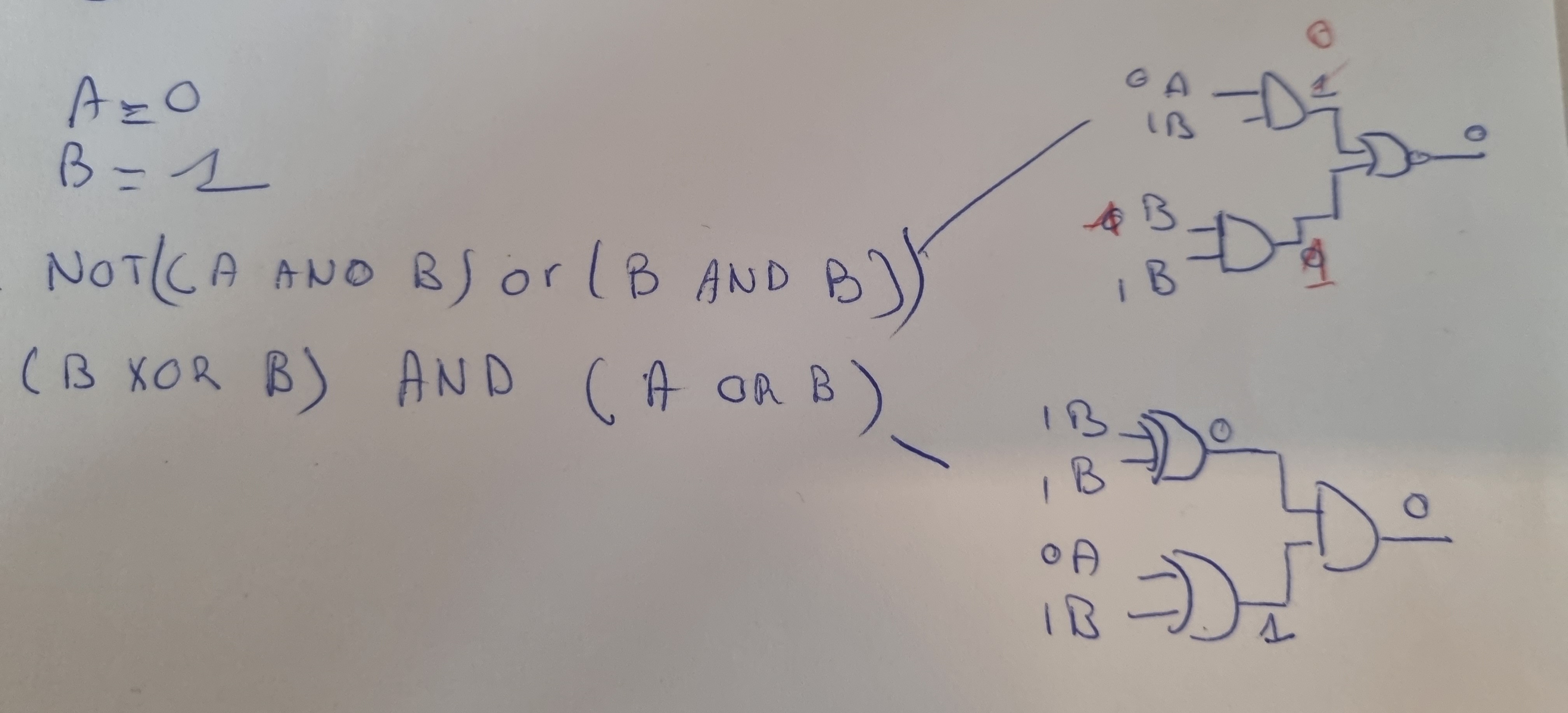
Chronogramme : chiffre informatique des valeurs binaires, mais aussi présenté dans le temps

table de vérité

schéma RTL : Register Transfer Level

portes logiques :

* AND
* OR
* XOR
* NAND
* NOR
* XNOR



TD : réaliser l’addition « A + B »

réaliser la table de vérité :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | S[1] | S[0] | S10 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 2 |

**TD diapo 20**

S0 : XOR

S1 : AND

réaliser un montage logique prenant en compte le carried bit.

S0 sera noté S

S1 noté Cout pour « carried bit input »

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | Cin | S[1]  Cout | S[0]  S | S10 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 2 |
| 1 | 0 | 1 | 1 | 0 | 2 |
| 1 | 1 | 1 | 1 | 1 | 3 |

Résolution : repérer les cas où la solution est vrai.

1ere étape : pour S = 1, donc repère la première équation pour S, et en ressortir l’équation booléenne : S = nA.A.n

Table de vérité pour S

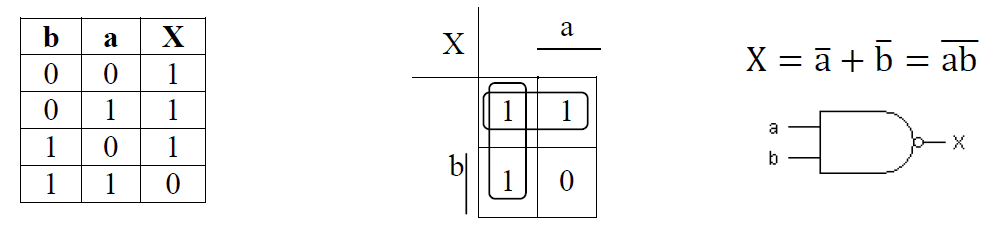
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | NOT A | NOT B | Not A . B | NOT B.A |  |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| A | B | NOT A | NOT B | Not( A . B) | B.A |  |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Réaliser une addition sur 4 bits, combinaison plusieurs full adders.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Y | CIN | NOT(Y°) | NOT(Cin) | NOT(CIN) AND (Y) | NOT(y) AND (Cin) | S |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |

**Table de Karnaugh**

****

|  |  |  |
| --- | --- | --- |
| X | a | not(a) |
| b | 0 | 1 |
| not(b) | 1 | 1 |

Une fois le tableau de Karnaugh réalisé, vient la simplification. Il faut regarder les groupement sur les même lignes, ou même colonne. Regarder si une ligne est tout le temps vraie : ici dans le cas de not(a) et de not (b). Donc ici l’équation est NOT(A) OR NOT(B)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | NOT(A) | NOT(B) | NOT(B) OR NOT(A) |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

On obtient la fonction NOT(B) OR NOT(A) = B NAND A

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | s1,s2 |  |  |  |
| s3,s4 | 00 | 10 | 11 | 10 |
| 00 | 1 | 1 | 1 | 1 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

Exercice : faire K-map pour S du circuit

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S | /B./Cin | /B.Cin | B.Cin | B./Cin |
| NOT(A) | 0 | 1 | 0 | 1 |
| A | 1 | 0 | 1 | 0 |

S = A.B.C XOR A.B.C XOR A.B.C XOR A.B.C

si on simplifie alors : S = Cin XOR A XOR B

developpement pour arriver à la simplification :

S = A.(B.Cin XOR B.Cin) XOR A(B XOR Cin)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| B | Cin | NOT(B) | NOT(Cin) | B.Cin | NOT(B).NOT(C) | B.Cin XOR NOT(B).NOT(C) |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 |

S = A.(B XNOR C) XOR A(B XOR C)

Y = B XOR C

S = A.(Y) XOR A.nY

S = A XOR Y = A XOR B XOR C

pour C :

C = B.C +A (nB.C + B.C + B.nC)

TD : proposer un diviseur

Réponse diapo 56.

Permet de réaliser une multiplication réalisée le plus rapidement possible

# présentation hardware de structure CPU

ça permettra de découvrir pourquoi le FPGA est plus intéressant qu’un micro

la base d’un calculateur : PU (Process Unit) et de la mémoire.

Généralement ils sont placés le plus proche possible, pour que les échanges soient les moins corrompus possible, qu’il y ait le moins de latence. C’est ainsi que nous verrons des mémoires gravées directement dans la puce.

Le meilleur est quand la mémoire est intégrée au chipset (et c’est moins cher).

Dans les téléphones FPGA vont se trouver d’avantage sur les téléphones. Ainsi quand il y a 5 caméras pour une photo par exemple, le FPGA sera utilisé car il y aura le « luxe » de choisir autant d’entrée que de sortie que nous voulons, et de pouvoir recevoir les images quasi en même temps.

gLe FPGA permet de prototyper des ASICS, avant de les concevoir, pour être sûr du travail à réaliser.

BRAM : embeded RAM, volatile. Son avantage, pas très très grande (genre 8 – 13 kB), mais intéressante pour stocker de la mémoire très temporaire.

GPU : aura une myriade de CPU en rang d’ognon.

Fait d’avantage de parallélisme

Un programme sera stocké dans la mémoire non volatile, pour pouvoir l’exécuter.

On va s’intéresser à ce qu’il y a dans la PU (Process Unit). Dedans, nous trouvons le « Control Unit ». Un message sera lu par le PU, décodé par le Control Unit. Interprété il sera envoyé vers l’unité nécessaire. Exemple : Arithmetic unit. L’information sera remontée alors au Control Unit, et stocké dans la mémoire, dans l’Output Data.

Ce circuit est Fetch (ce qu’il faut faire), decode, exécute, store.

Le « store » (stockage) coûte beaucoup d’énergie, du fait du passage entre la mémoire et le PU.

Dans le cas où il y aurait 2 calculs en parallèle, le full adder est mutualisé, ça ajoute un retard du temps d’au moins une opération. Donc risque de ne pas savoir quand l’information va arriver.

Pour un FPGA, il n’y a pas cet aspect d’étape, mais le calcul pourra être fait à la volée. Ça réduit la consommation d’énergie, et relance rapide car ne dépend pas de la mémoire. Certains cas, même si le développement peut prendre du temps, on demandera un FPGA pour une réaction le plus rapide possible.

Le chemin le plus long dans le système est le chemin critique. C’est lui qui nous empêchera de monter en fréquence.

**26/04/2023**

# TP

Un langage contient de nombreuses subtilités. Ici, peu d’exemple simple, mais plutôt des exemples complets, comprenant plusieurs élément simple.

Langage de description matérielle : bas niveau, pour décrire le comportement, l’entrée et la sortie de l’architecture numérique

3 principaux langages : VHDL, Verilog et SystemVerylog

ex : and en VHDL : c ≤ = A and B

les principaux langages :

* VHDL : plus couramment utilisé pour les systèmes complexes. Permet de simuler et donc d’observer plus facilement le signal
* Verilog : plus bas niveau que VHDL ; pas le même niveau de design
* SystemVerilog : extension de Verilog.

Pour passer du code à l’architecture physique, plusieurs étapes.

1. run Synthesis

synthétisation, implémentation, bitstream : 3 étapes obligatoires pour configurer la carte.

Synthétisation : synthèse qui programmera la carte

implémentation :

bitstream : pour le binaire qui programmera la carte

une fois toutes ces étapes : faire open target, puis auto connect

et enfin

# VHDL

On y déclare une entité. Puis à la suite, description de l’intérieur de la « boîte », de son comportement.

Une entrée pourra seulement être lue, pas modifier. La sortie ne pourra être qu’écrite.

‘‘<=’’ : affectation

logique séquentielle : la sortie dépend du résultat précédent et actuel du signal, et nécessite une horloge. Utilisé pour Registres, compteurs, machine d’état, mémoire, etc.

Pour un compteur par exemple : il faut un repère, savoir d’où on part pour pouvoir faire un +1.

logique combinatoire : dépends uniquement de l’état actuel du signaux, sortie immédiate. (pas d’horloge)

Dans le VHDL, il y a des instructions : concurentes et séquentielles.

Concurrente : pas de priorité des actions, tout s’exécute en même temps. Si 2 instructions essaient d’affecter le même signal, il va y avoir un conflit.

Instructions séquentielles : des instructions exécutées les unes après les autres. Si 2 instructions affectent le même signal, il prendra la dernière.

Un process (en VHDL): un ensemble d’instruction. On entre dans le process à partir des signaux qui seront les « simulis » ; s’il y a un changement sur ces signaux, on entrera dans le process. Le process peut s’exécuter en parallèle de tout le reste.

Les registres : élément mémorisant : il mémorisera quand il y aura un front montant de l’horloge : il met à jour l’élément mémorisé lors du signal d’horloge (montant dans l’exemple), ce qui change le signal envoyé en sortie. Le signal envoyé en sortie est toujours le signal mémorisé dans le registre.

Resetn = mise à zéro du registre si resetn = 0.

les machines à états : permet de représenté différents états du système : dans tel état, mon signal

vaut tel valeur.

Sur le schéma, clk veut dire quand clk = 1. led = 1 : valeur du signal dans un état.

Les process : un process synchrone, un process combinatoire.

Décalage : Il existe des symbole RTL pour ça. 2 types de décalage : par rotation et par insertion

Par insertion : on décale les bit, et on concatène avec la valeur définie de la variable MSB.

Par rotation : on décale les bits, et on concatène(on ajoute un bit aux autres bit) avec la valeur du MSB pour le décalage vers la gauche, par le LSB pour le décalage vers la droite.

**TP full adder.**

1)

S = Cin XOR (A XOR B)

Cout = (Cin AND (A XOR B)) OR (A AND B)

2)

entrées du full adder : A,B, Cin

Sorties du full adder : S, Cout

3) voir full\_adder.vhd

synoptique : permet de visualiser la fonction, de façon qui peut être très précise, ou d’autres plus détaillées.

C’est intéressant quand surtout nous sommes dans un système complexe ; permet de catégoriser les opérations, et de mieux s’y retrouver.

**Flot de développement**:

Commencer par schéma RTL / synoptique. C’est une bonne base, qu’il faut bien travailler et prendre du temps, car toute la suite se base sur elle. Il faut être le plus clair, le plus précis.

Ensuite retranscrire le schéma RTL en Vhdl. Si on fait des correction dans le VHDL, mettre à jour le RTL correspondant ; ça permettra par la suite d’y voir clair.

Les contraintes du système : le timing (qui correspond aux horloges) et des contraintes sur la carte (tel composant doit être proche de tel autre).

Grâce à la simulation et le *test bench*, ça permet de vérifier le comportement du système.

Une fois certain de notre système, on fait la synthèse qu’on étudie, pour voir si ça correspond à notre schéma RTL. Si non, ça veut certainement dire que nous avons mal écrit quelque chose dans les étapes précédentes.

Placement des sondes pour observer les signaux à certains moment, pour vérifier sur le système le comportement : l’ILA. On permet ainsi de vérifier les signaux, soit avec VIVADO, soit avec un oscilloscope.

Il va falloir ensuite placer les choses sur la carte, l’étudier pour voir s’il est correct.

Dernière étape : nous pouvons vérifier le fonctionnement sur une carte.

*Simulation :*

Pour toutes les situations, tester le comportement de l’architecture. C’est à dire piloter toutes les entrées, et observer le comportement en sortie s’il est correcte. Ces tests peuvent être réalisé automatiquement.

Process : sans liste de sensibilité, appelé à t = 0 et exécuté en boucle.

Les signaux de la simulation sont obtenus par le testBench.

**27/04/2023**

**Outils fpga**

**HDL CODER**: permet de générer un code, en couplant avec matlab.

Il est possible d’indiquer les contraintes de temps à Vivado, pour qu’il optimise le placement pour régler la latence en fonction de la criticité de la fonction.

Le chemin Critique est le chemin le plus long, celui qu’on voudra réduire d’avantage la latence.

Schéma data flow : établir un schéma de flux de donnée :

Arithmetic Unit (Full adder)

PU

A

Output 2

B

C

D

Arithmetic Unit (Full adder)

Output 1

l’étape d’implémentation pouvant prendre énormément de temps, l’étape « synthèse » permet de simuler et de vérifier que nous n’avons pas écrit n’importe quoi.

Boite de routage : tout les cables qui en sorte vont vers d’autres blocs, vers d’autres interface. 1er job : ça permet de rediriger. Autre job, un signal ne peut pas être traité par un seul LUT par exemple, ainsi transférer d’un LUT vers un autre

buffer : permet de passer de la tension intérieur à la tension extérieur du composant.

Slice :

Slice : appelé parfois aussi « logical element » (ou autre), est un regroupement de composants logiques bas niveau. Chaque FPGA aura sa propre architecture de slice, lui conférant des performances différentes.

Dans une Slice, on peut trouver différents éléments :

LUT : Look Up Table, permet de reproduire le comportement d’une porte logique. Latence un peu plus élevé qu’une porte logique.

MUX : permet l’aiguillage des signaux dans la Slice, ou d’une Slice à l’autre.

FF (Flip Flop) : mémorisation de signaux, 1 FF mémorise 1 bit

Carry 4 : permet d’optimiser dans la série Zync 7010 les fonctions de comptages récurrentes : des performances en latence. (plus d’information sur <https://www.fpga4fun.com/Counters.html>)

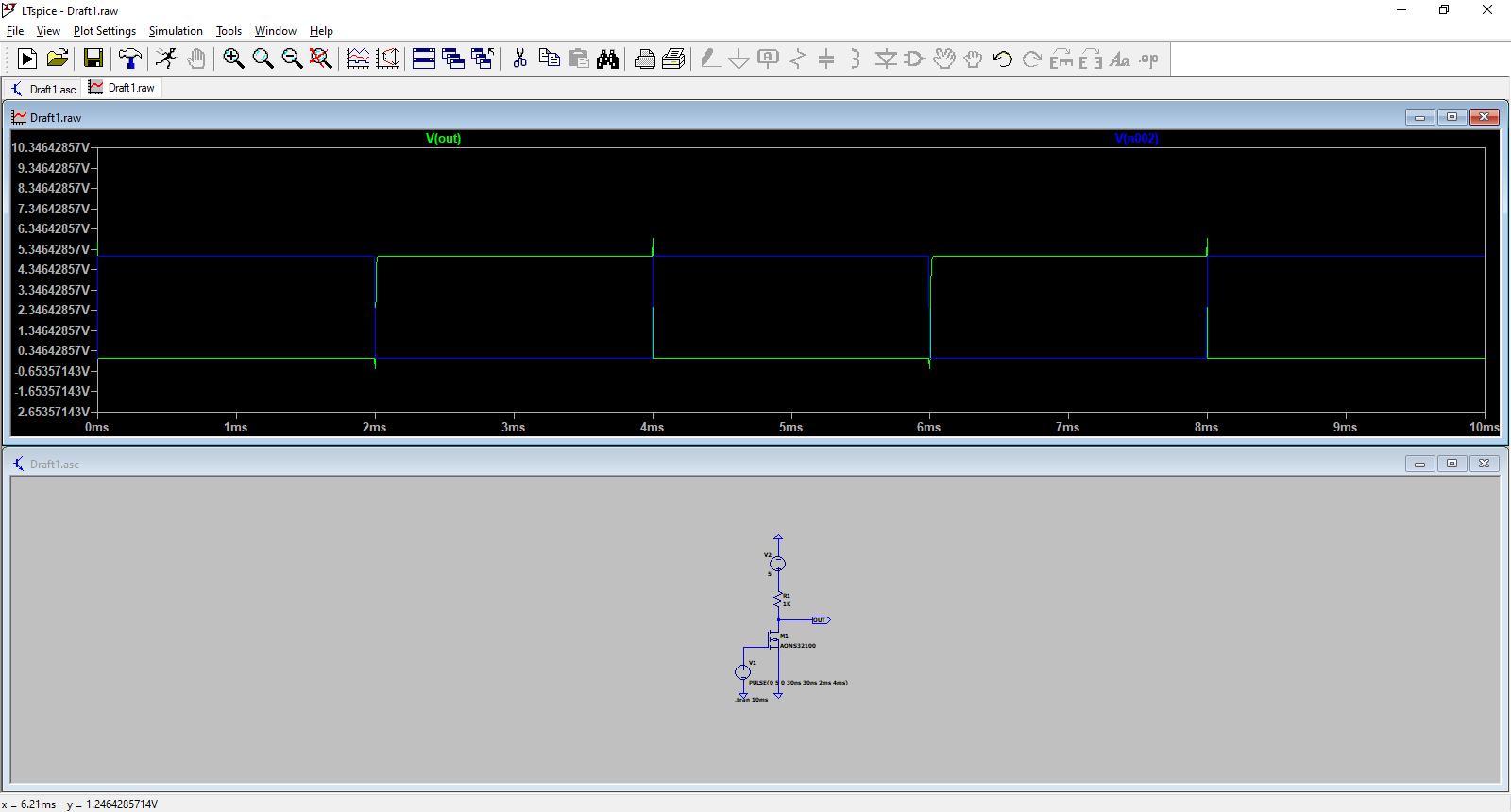
D’autres fonctions peuvent être ajouter ou regroupé selon les FPGA

# MOSFET

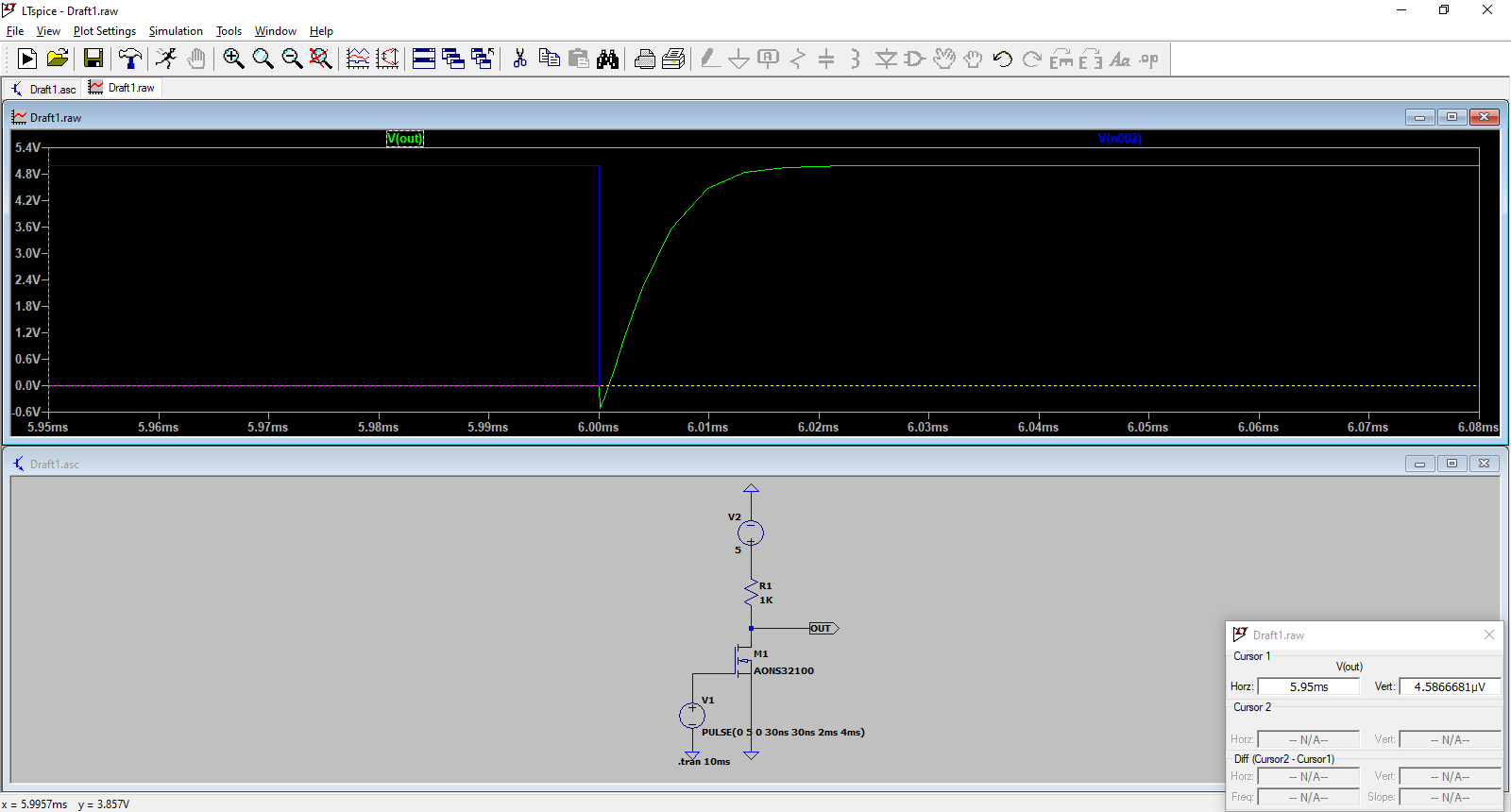
[**https://www.emse.fr/~dutertre/documents/2\_cours\_MOS\_2021.pdf**](https://www.emse.fr/~dutertre/documents/2_cours_MOS_2021.pdf) : exemple en détail des MOSFET

PMOS : fermé quand Vgs < -Vth (ex : Vth = -0,5V)

NMOS : fermé quand Vgs > Vth

**exercice simulation MOSFET**

Nous réalisons ici un circuit inverseur. Si nous zoomons, nous pouvons observer le temps de latence :



exercices

# LUT (Look Up Table : Table de vérité) :

c’est ce qui permet la reconfiguration des composants.

C’est une mémoire qui possède un bus d’adresse, sur lequel on connecte nos entrées. On prends tous les cas voulus, qu’on enregistre dans la mémoire. C’est programmer de la RAM.

Comment faire la translation entre des portes logiques et un adressage ? C’est l’outil de synthèse du logiciel de programmation du FPGA qui fera cette transition.

Matériellement, c’est des multiplexeur qui permettront de faire le multiplexage.

La mémoire sera principalement en RAM, mais parfois peut être en Flash, pour gagner du temps de traitement, le plus rapide possible.

Le bitstream devra être sauvegardé quelque part. Au démarrage le Bitstream récupéré sur une mémoire non volatile, pour pouvoir mettre à jour à la carte, qu’elle soit prêt à être exécutée.

RAM : intéressant pour la densité, la quantité d’éléments logiques dans une surface donnée. Donc nous pouvons avoir plus de données. Plus rapide d’accès

ROM : base flash, cadencé moins vite. Mais pas nécessaire d’avoir un système de mémorisation à l’extérieur du composant ; c’est embarquant. Moins sujet aux attaques matérielles.

Encore plus loin, les FPGA à base de fusibles : à la programmation, couper ou non des fusibles, ce qui donnera la programmation des LUT. Intéressant pour la résistance aux radiations, et très difficile de pirater.

# Utilisation de git

commandes pour git bash :

« git clone *adresse du fichier clone »*

*git status :* permet de voir ce qui n’a pas été envoyé ou pas

git add –all : ajouter les fichier à commiter.

git commit -m « texte description »

git config user.email « email@email.com »

git config user.name « name » : configuration du git pour l’utilisateur

git push

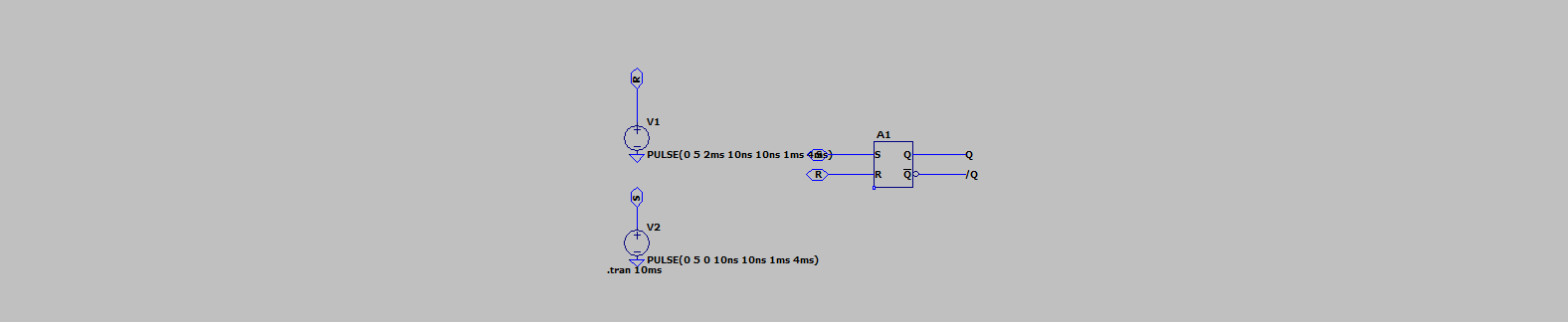
# Logique numérique synchrone

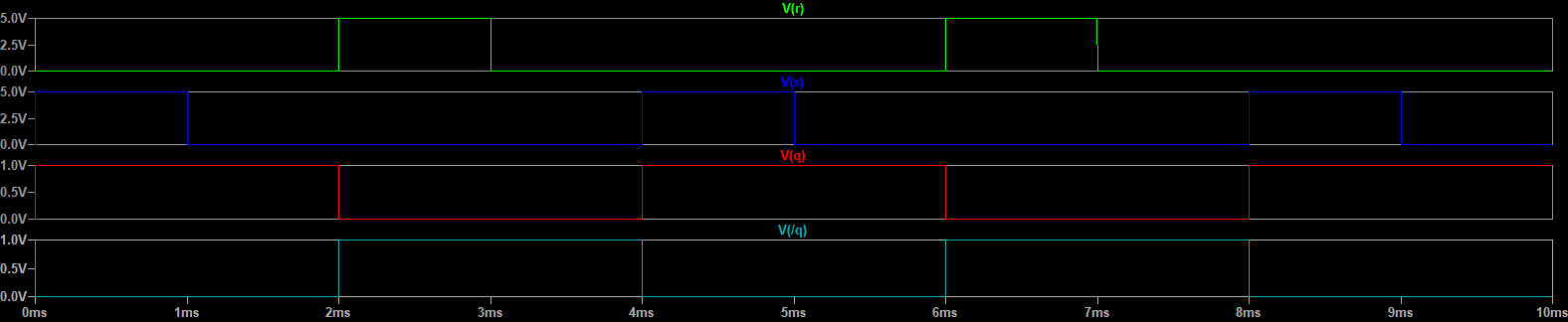
## La bascule

Un élément de mémorisation, avec 2 état stable qui peut être utilisé pour stocker des informations. Lorsque nous sommes dans un état « latch » (S = R = 0), la sortie Q conserve sa valeur d’état précédent.

Exemple : S = 0 et R = 1, puis S = 0 et R = 0, puis S = 1 et R = 0, puis S = R = 0

simulation sous LTPSPIC

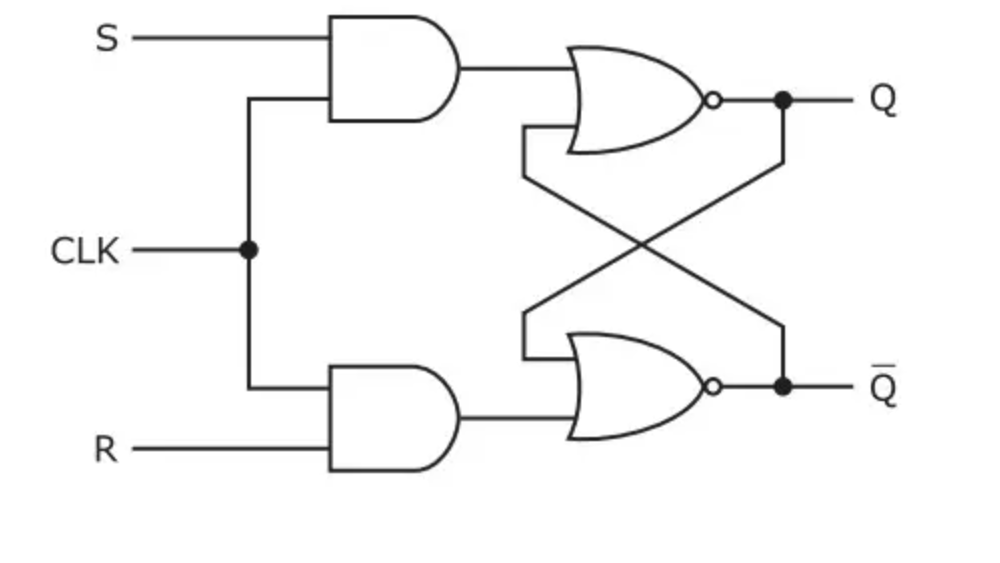


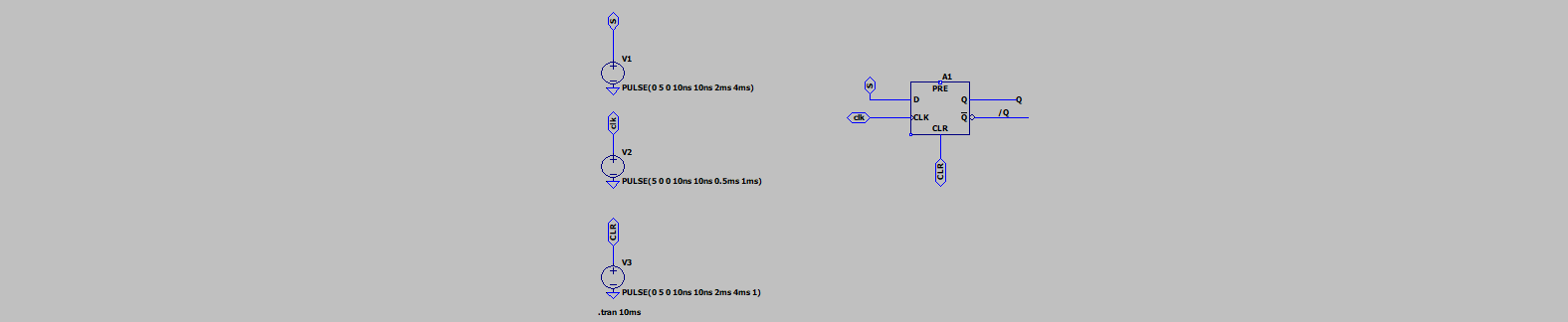


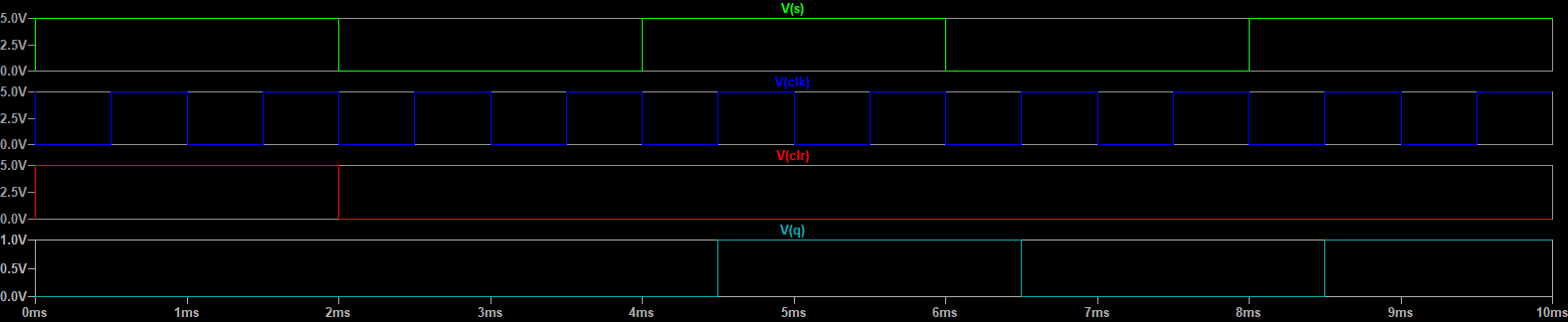
Nous observons bien alors la mémorisation de l’état précédent quand les broches d’entrées passent à 0.

## registre

Une fonction recherché est la synchronisation. Ce que nous obtenons alors avec le registres







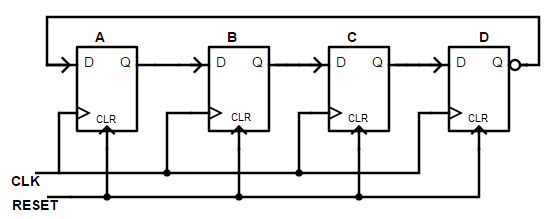
Dans le comportement d’un registre, le Reset est asynchrone ; c’est à dire qu’il permet de faire le reset même quand la clock ne vaut pas 1.

le RESET NOT est privilégié car moins enclin au problème électromagnétiques.

## Les compteurs

utilisés de façon très récurrentes dans les désign numériques. Ils permettent de réaliser par exemple les WatchDogs (interrompre un processus quand il ne répond pas au bout d’un certain temps).

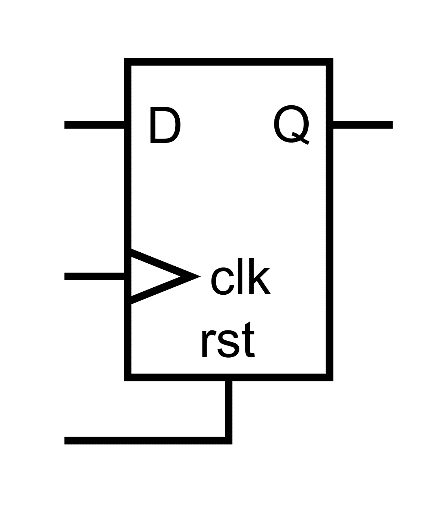
### Compteurs de 4 bit



réaliser le chronogramme.

Nous avons réalisé ici un compteur de Johnson.

Représentation RTL :



4

plusieurs type d’applications du compteur. Nous avons vu le Watchdog, il existe aussi les machines à état.

# Machine à état (FSM : Finale State Machine)

Ça permet de résoudre des équations un peu plus complexes : un ensemble d’état possible, et des évènement ou des conditions qui déclenchent la transition d’un état à l’autre.

Dessin 1: diagramme d'état

« Feu vert »

« Feu rouge »

« Feu rouge »

un double cercle représente d’où on part (à un reset machine)

⇒ on se concentre vraiment juste sur les états de la machine et ce qui va faire la transition.

Synoptique : vue globale.

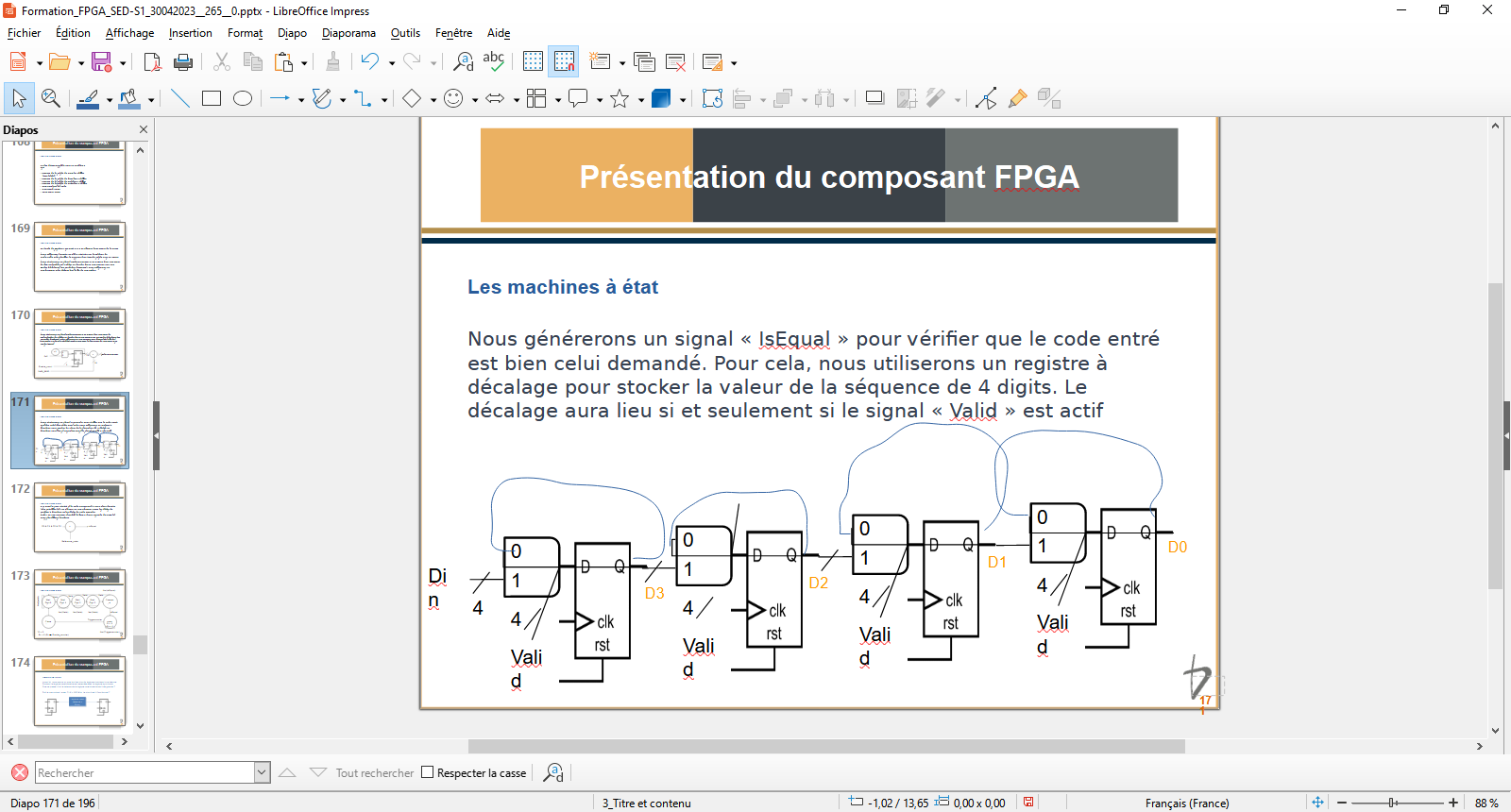
#### Types machines à état

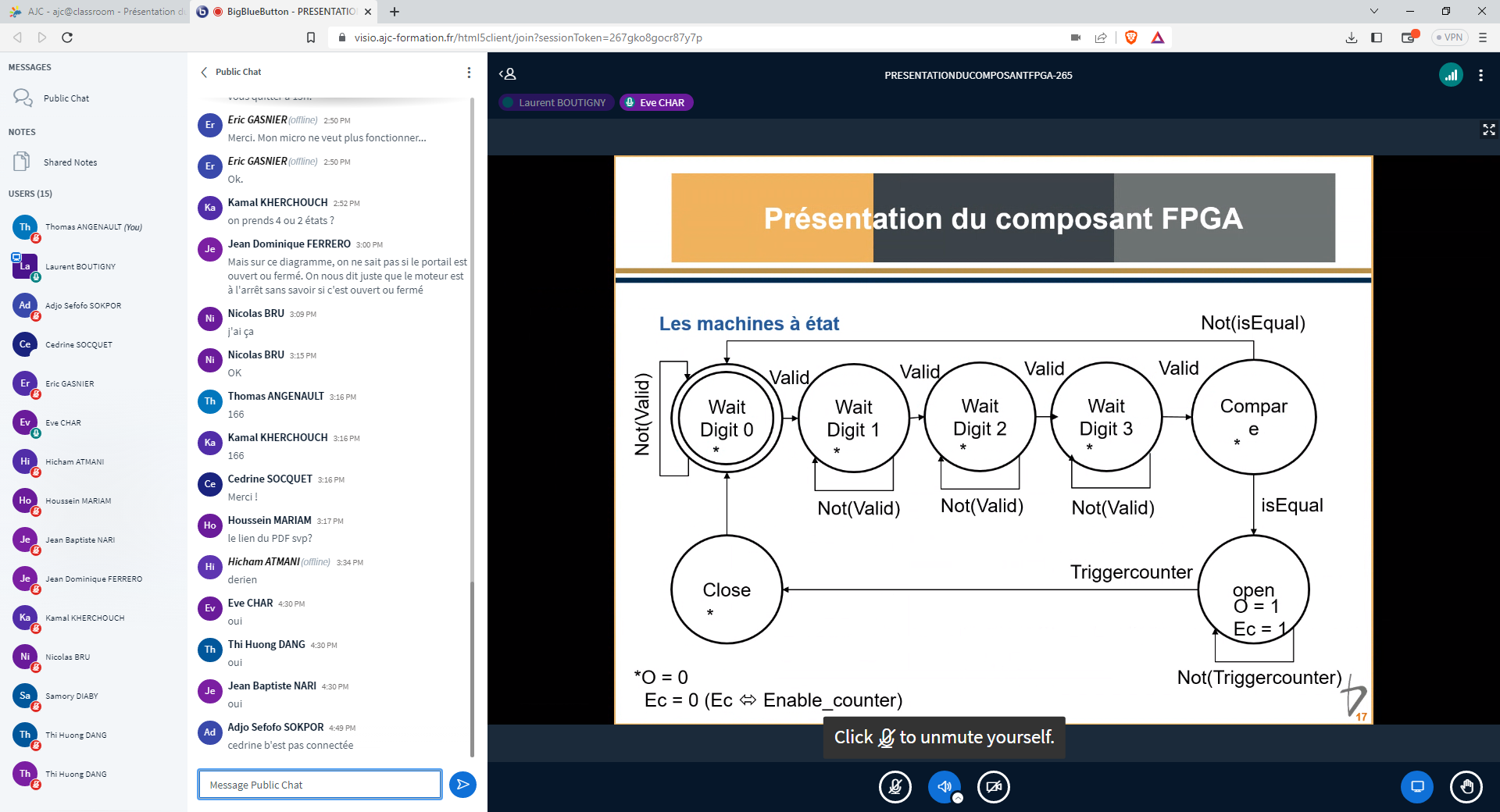
Machines de Moore : l’état suivant dépends uniquement de l’état courant

Machines de Mealy : on ajoute à l’état courant des signaux d’entrées pour le changement d’état.

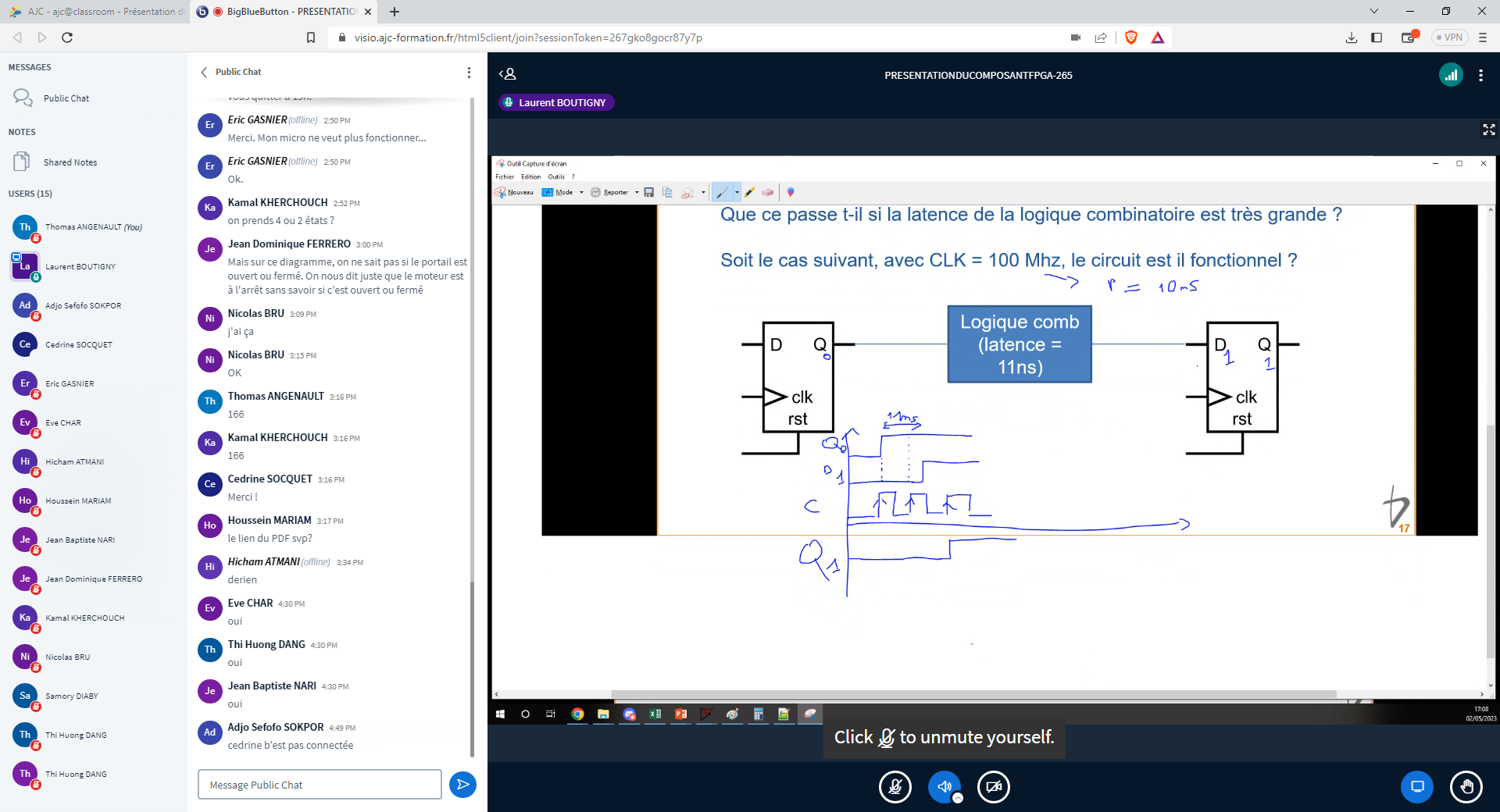
Possible d’avoir plusieurs FSM pour un même problème ; différentes solutions possibles.

Exercice Digicode :

Figure 1: registre à décalage pour récupéré le code entré

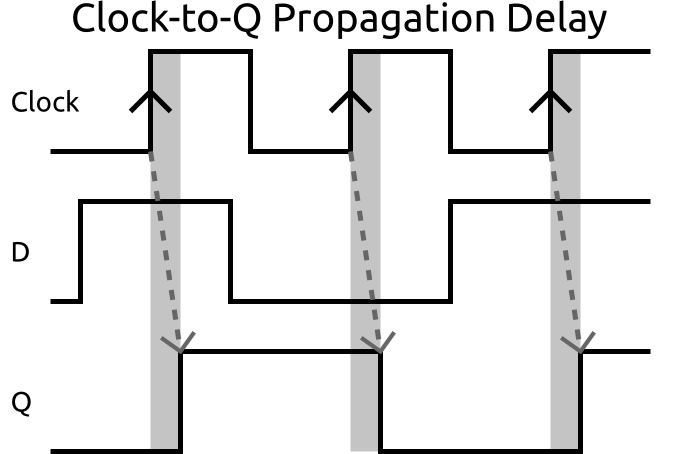


# gestion de timing



Dans l’exemple ci-dessus, le temps de latence décale Q1 d’un cycle d’horloge, ce qui fait que ça peut créer par la suite des problèmes de calcul. Par exemple, si nous associons Q1 à un Q2 qui arrive au front montant d’horloge précédent, il y aura une période d’horloge où le signal sera faux.

Il y a une latence entre l’horloge et la sortie synchrone :



Si le signal en entrée est dans un état transitoire, le signal de sortie sera alors dans un état intermédiaire, dit **métastable**, qui risque de transmettre des erreurs par la suite.

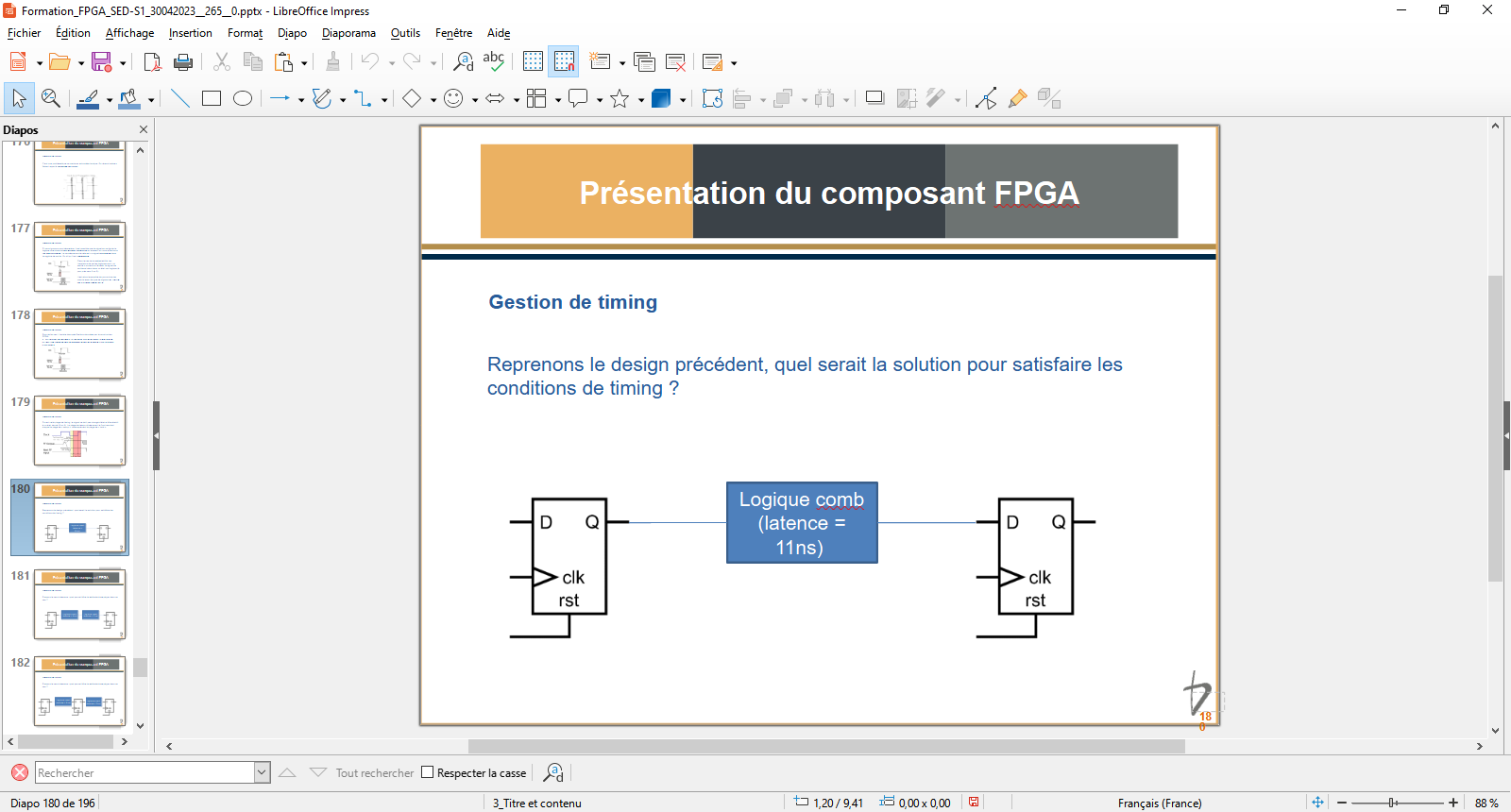
Pour éviter ce problème, une spécification a été donnée par le constructeur de FPGA : « *Tout signal, en entrée d’un registre, doit être dans un état stable durant une plage de temps s’étalant avant et après le front montant de l’horloge* ».

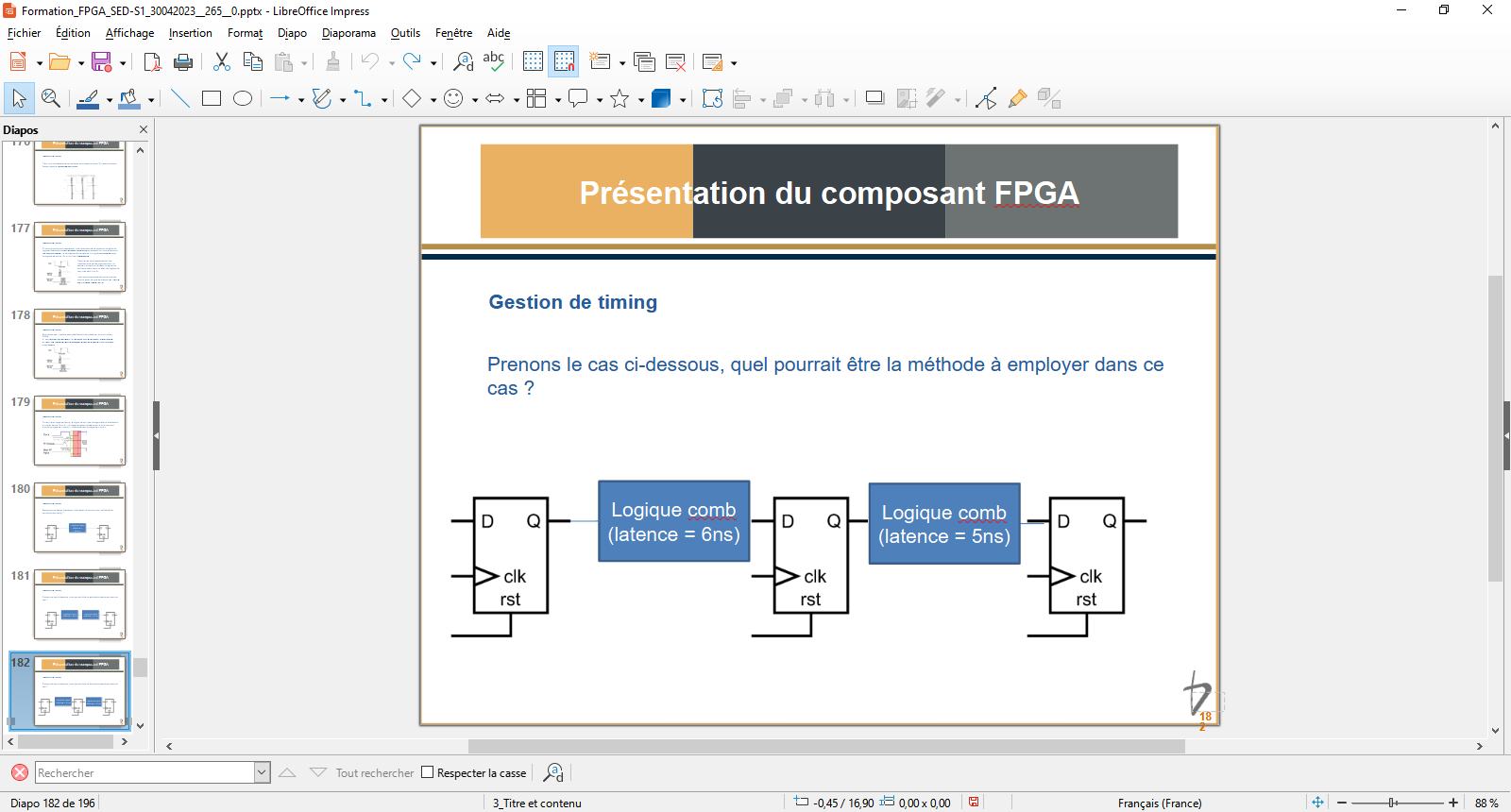
plage setup : temps où le signal doit être stable avant le front montant

plage hold : temps où le signal doit être stable après le front d’horloge

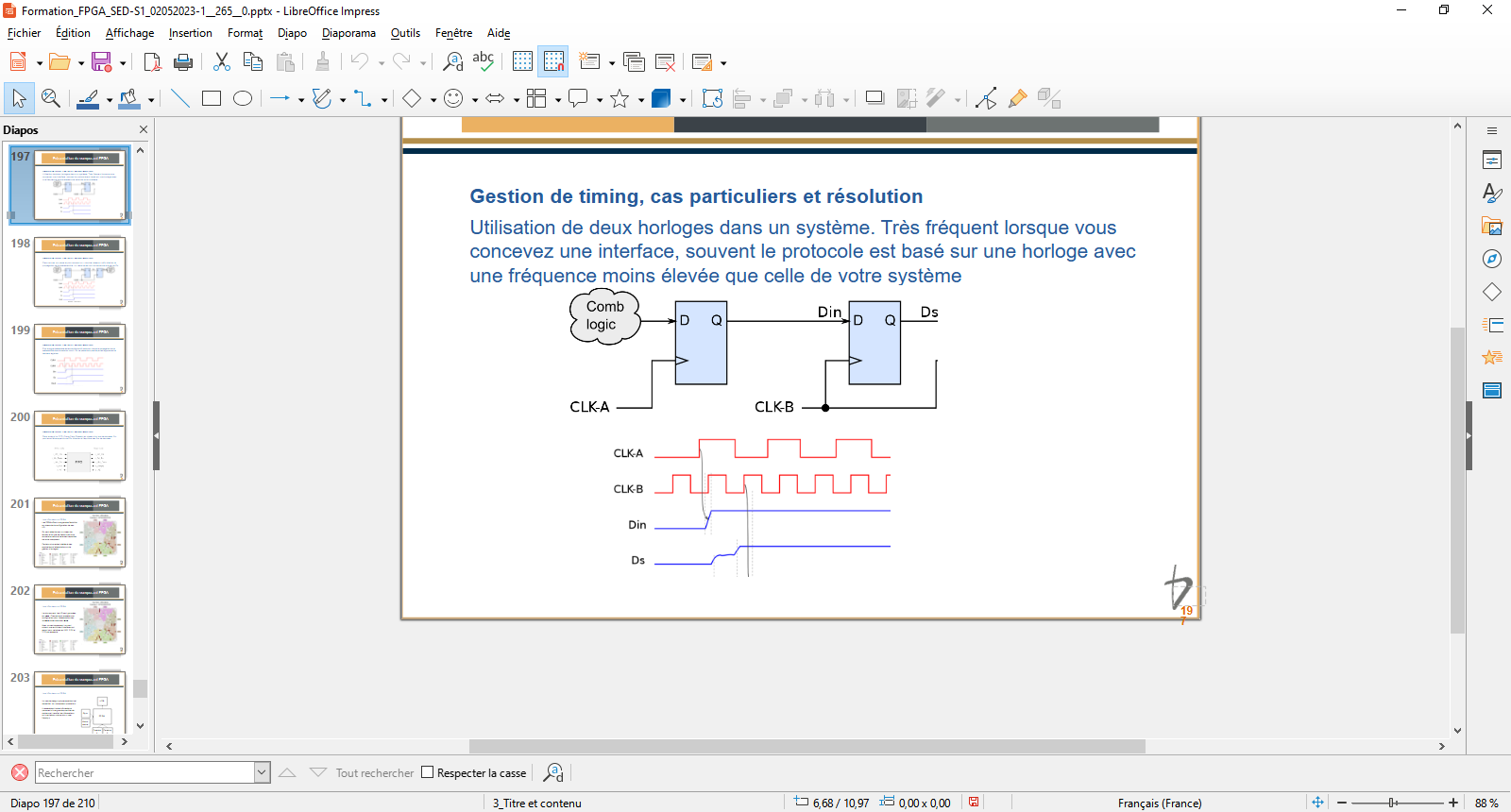
Slack : temps par rapport au début de la plage setup où le signal commence sa stabilité ; si positif, bonne stabilité, si négatif : pas bonne. Existe aussi avec la plage Hold.

exemple : comment modifier le premier schéma pour satisfaire les conditions de timing.

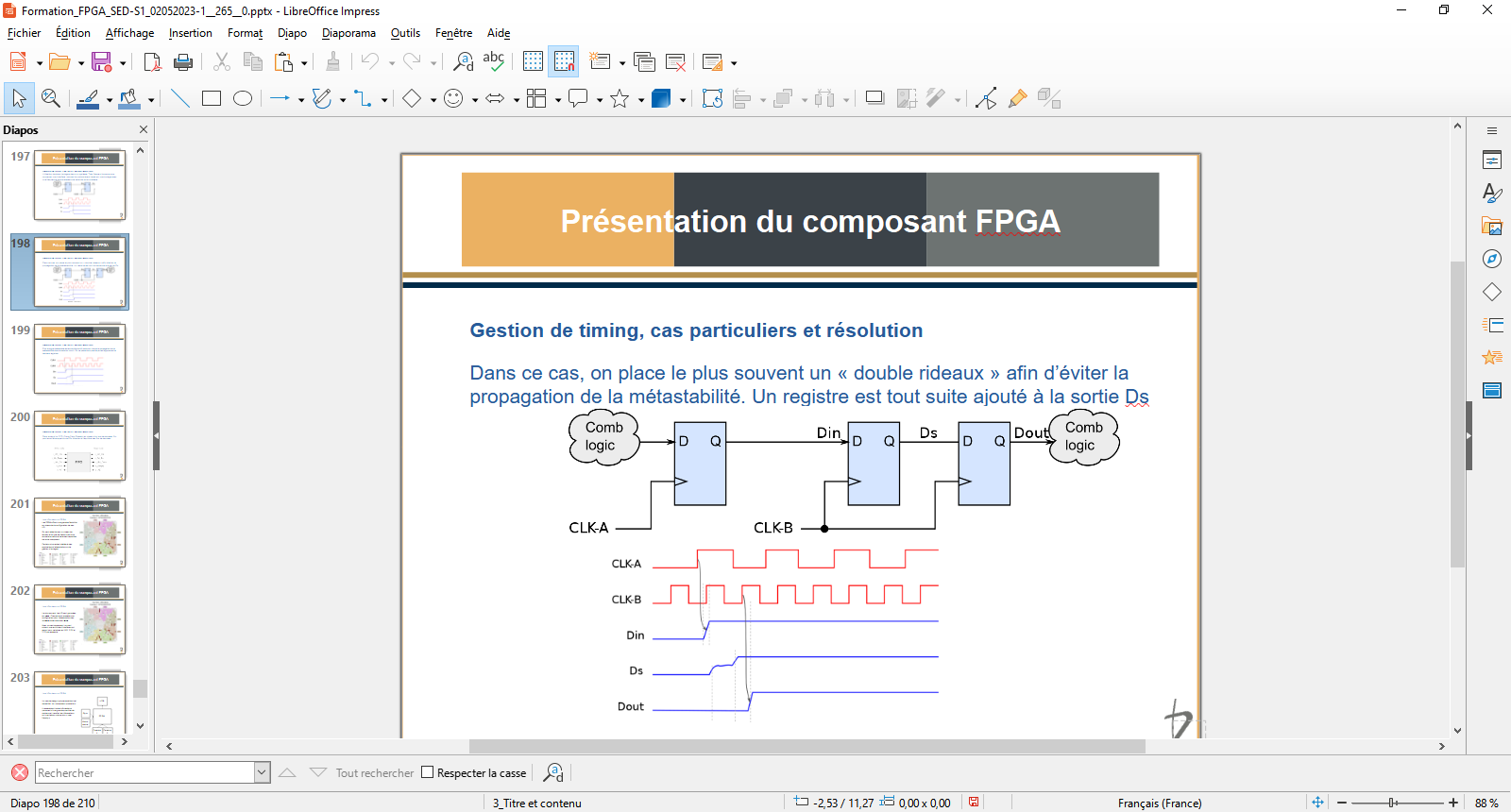




### cas particulier



Dans ce cas, nous avons le signal Din qui a une première instabilité, qui crée une métastabilité en sortie (Ds). Pour corrigé cela, remplacé souvent par un « double rideaux », qui permettra de récupérer le signal sur sa stabilité :



Il est possible aussi d’utiliser une FIFO pour un même résultat.

## clock shew :

Lorsque le signal n’arrive pas en même temps aux registres.

Pour le limiter, les horloges sont routé sur un réseau particulier : **clock tree**.

Attention : il ne faut pas ajouter de logique combinatoire sur le signal d’horloge, car sinon on rajoute de la latence, et on obtient des erreurs pour la suite. Il faut donc utiliser des PLL pour modifier le signal d’horloge.

# Entrées/sorties dans FPGA

Les entrées/sorties sont regroupées par régions : Bank

Un seul composant peu rassembler une multitude d’interface, avec différentes tension d’alimentation sur une même carte.

Composant de proximité : permet de rassembler divers information sur un seul « câble ».

Ainsi pour avoir cet flexibilité, plusieurs standards doivent pouvoir être accessibles par les broches. Il est ainsi possible de configurer les pin d’entrée et de sortie en fonction de certains standards.

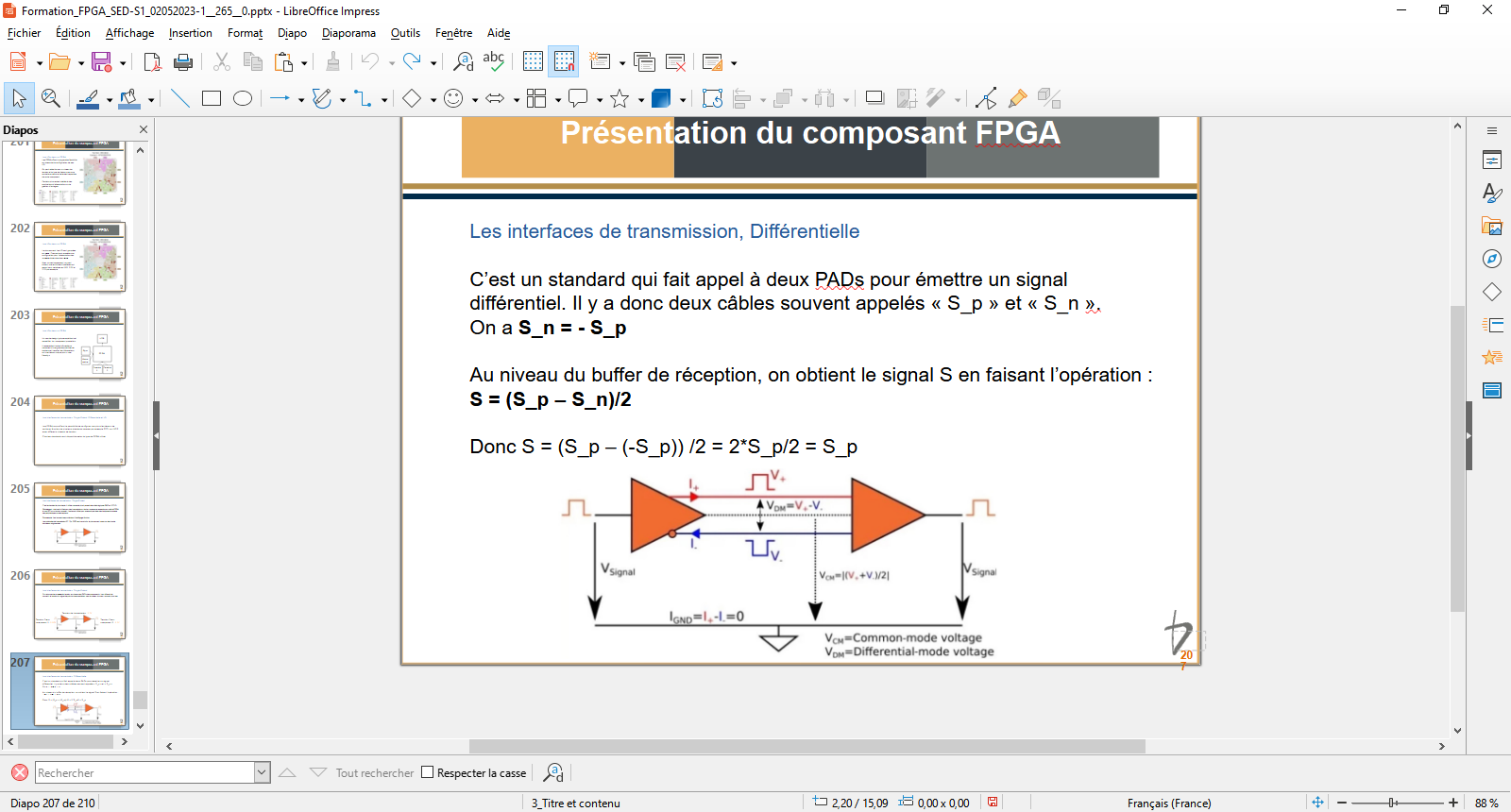
## Single ended

Standard le plus simple, permet de 2 états logiques : gnd et Vcc. Utilisé plutôt pour les transmissions « lentes ».

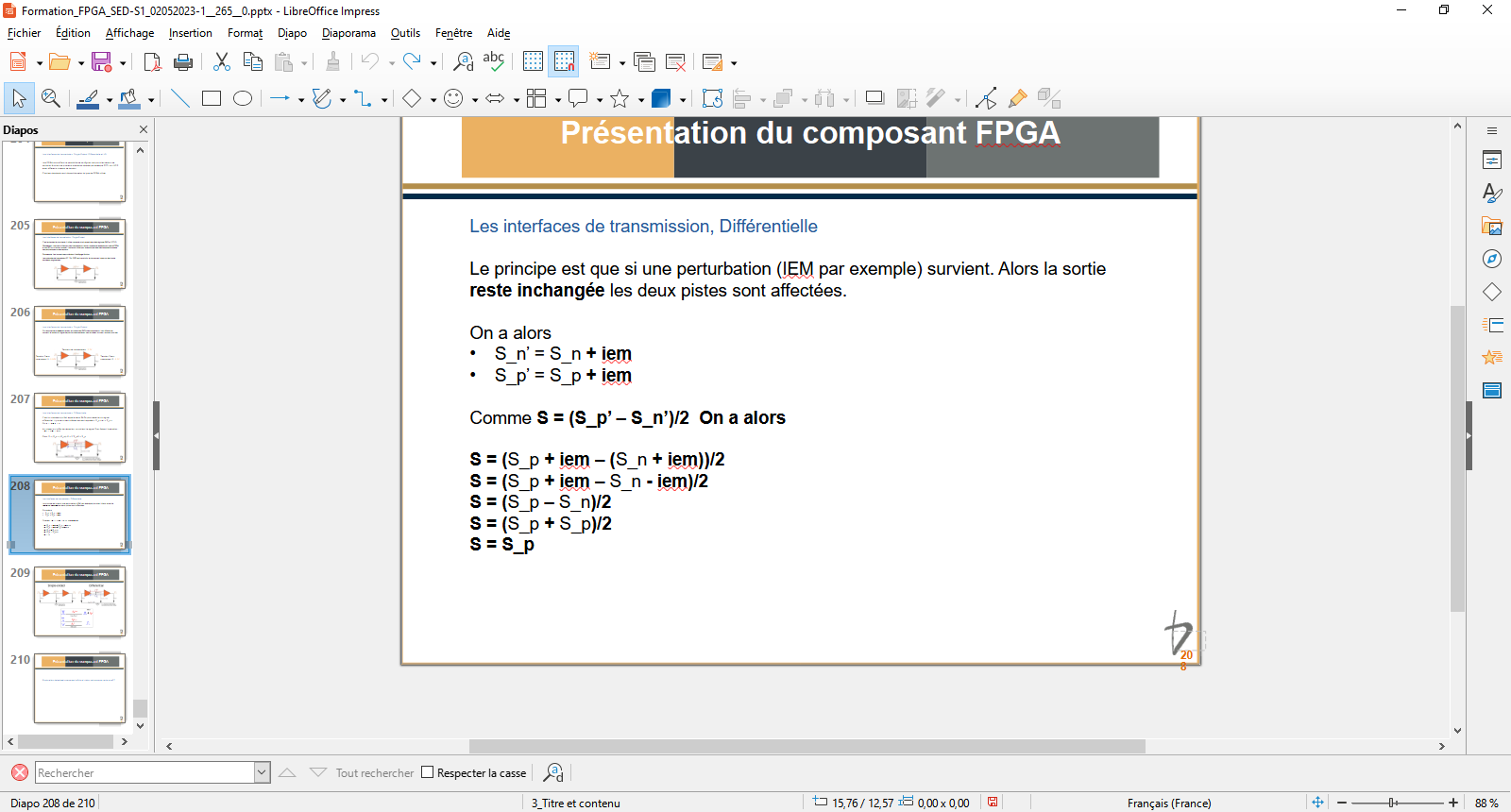
Nous y retrouvons des buffers, qui permettent de faire la transition entre la tensions du signal extérieur vers le signal de coeur (plus petit pour des raisons de consommation du composant, économie d’énergie).

## interface de transmission, différentiel

Des signaux single ended qui sont apparairé, deviennent alors différentiels.



Son avantage : lorsqu’une perturbation arrive, elle frappe les 2 câble, S\_p et S\_n. Et si on développe l’équation, on découvre qu’alors les perturbation s’annulent : C’est un interface beaucoup plus résilient. L’obligation est de coller un maximum S\_p et S\_n, pour que si une perturbation arrive sur un câble ne soit pas reportée sur l’autre.



Autre règle de design : les 2 câbles doivent être de la même longueur, car il y a alors un décalage entre S\_p et S\_n, ce qui enlève l’intérêt et le bon fonctionnement du différentiel.

Plus rapide, plus résilient, donc pour les interface rapide, c’est le mode préféré, et donc très commun.