formation POEI FPGA

samory.diaby

April 2023

Table des matières

1 Intro

1.1 Representation des chiffres en nombres binaires/décimal/Héxa

TD 18 -> 0001 0010 -> 0x12 40 -> 0010 1000 -> 0x28 64 -> 0100 0000 -> 0x40

1.2 Chronogramme

chronogramme -> vue analytique de la donnée liée à une simulation mesure -> vue physique de la donnée liée à une mesure en labo

1.3 Opération logiques basiques

- Chaque opérateur logique possède : - un schéma Register Transfer Level (RTL) : niveau le plus bas pour effectuer un calcul (portes logiques) - une table de vérité (Look Up Table)

1.3.1 Portes basiques

- AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

TABLEAU 1 – AND gate

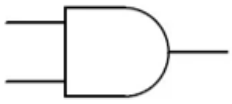


FIGURE 1 – AND gate RTL representation

- OR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

TABLEAU 2 – OR gate

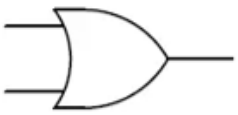


FIGURE 2 – OR gate RTL representation

- XOR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

TABLEAU 3 – XOR gate

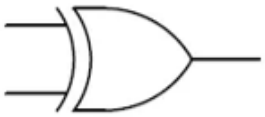


FIGURE 3 – XOR gate RTL representation

- NAND

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

TABLEAU 4 – NAND gate

FIGURE 4 – NAND gate RTL representation

- NOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

TABLEAU 5 – NOR gate

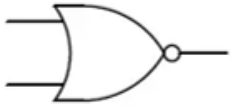


FIGURE 5 – NOR gate RTL representation

- XNOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

TABLEAU 6 – XNOR gate

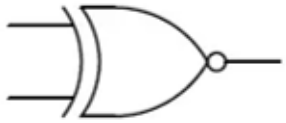


FIGURE 6 – XNOR gate RTL representation

1.4 TD 2 : exos portes logiques

consigne :

On suppose que A vaut 0 et B vaut 1 quelle est la sortie des opérations suivantes ?

- NOT((A AND B) OR (B AND B))
- (B XOR B) AND (A OR B)

1) : NOT((A AND B) OR (B AND B)) : rés -> **0**

cheminement :

eq1 : A AND B -> 0
eq2 : B AND B -> 1
eq3 : **eq1** OR **eq2** -> 1
output : NOT **eq3** -> 0

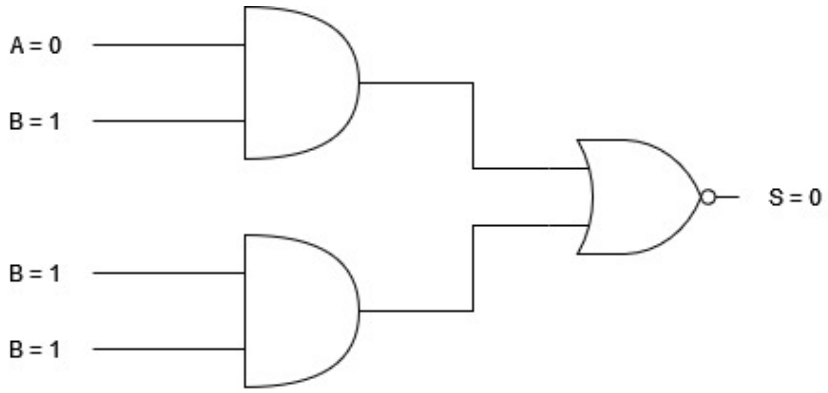


FIGURE 7 – Représentation RTL

2) : (B XOR B) AND (A OR B) : rés -> **0**

cheminement :

eq1 : B XOR B -> 0
eq2 : A OR B -> 1

output : **eq1** OR **eq2** -> 0

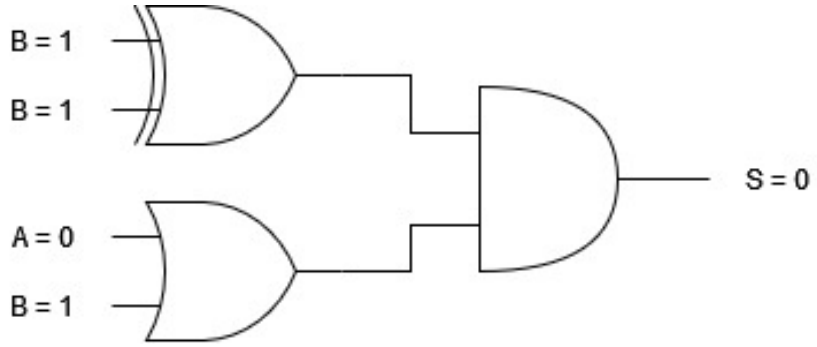


FIGURE 8 – Représentation RTL

1.5 TD3 : adder + full adder

1.5.1 Additionneur binaire

A	B	Output(base10)	Output (base2)[1] (Cout)	Output (base2)[0] (S)
0	0	0	0	0
1	0	1	0	1
0	1	1	0	1
1	1	2	1	0

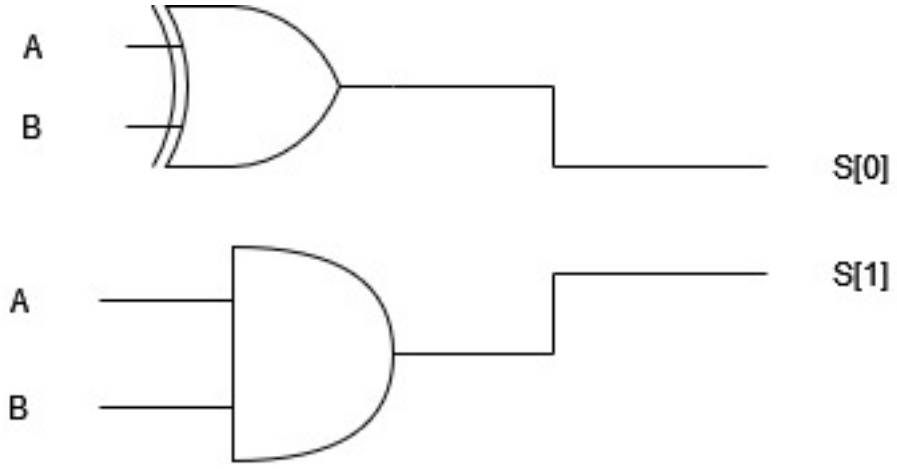


FIGURE 9 – Schéma logique de l'additioneur

Correction

$S = (A \oplus B) \oplus Cin$
$Cout = (A.B) + ((A \oplus B).Cin)$

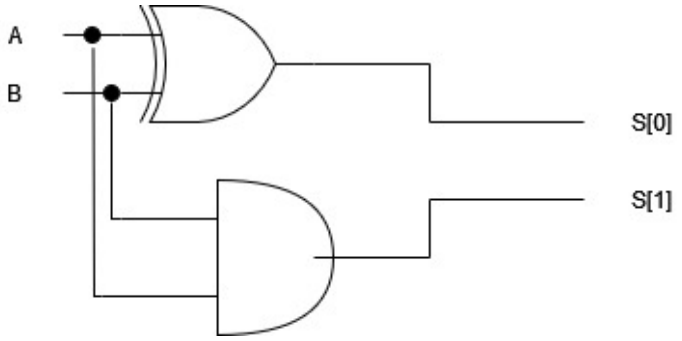


FIGURE 10 – Schéma logique de l'additioneur (*Correction*)

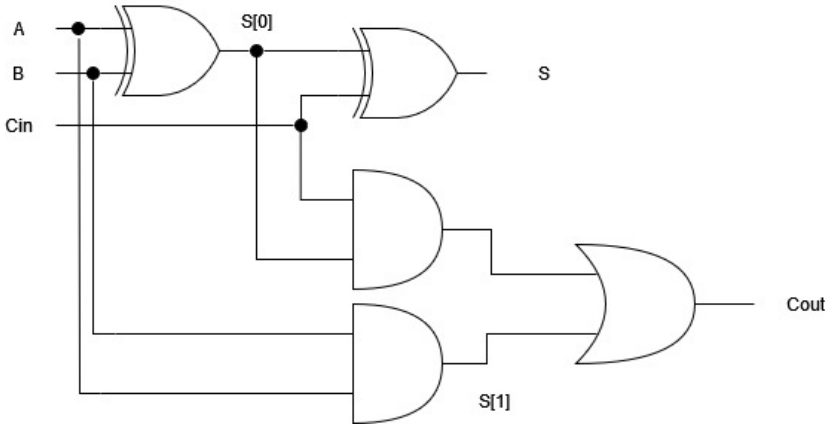


FIGURE 11 – Schéma logique du full adder (*Correction*)

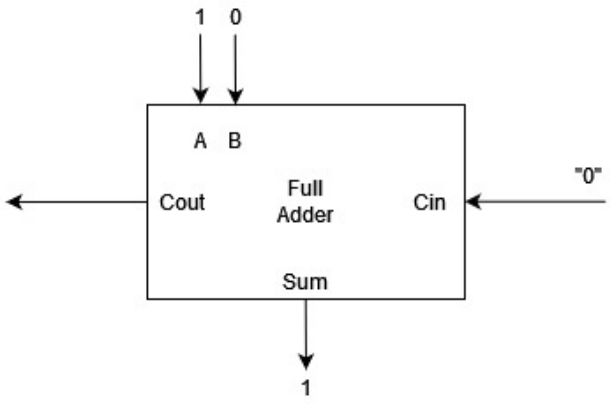


FIGURE 12 – 1 full adder

— A, B : entrées de l'additionneur (1 bit pour A, 1 bit pour B)
— Cin : entrée de la retenue (1 bit)
— Sum : Sortie de l'addition entre A et B (1 bit)
— Cout : Sortie de la retenue (1 bit)

Nouvelle consigne : combiner des full adder pour additionner des chiffres sur 4 bits

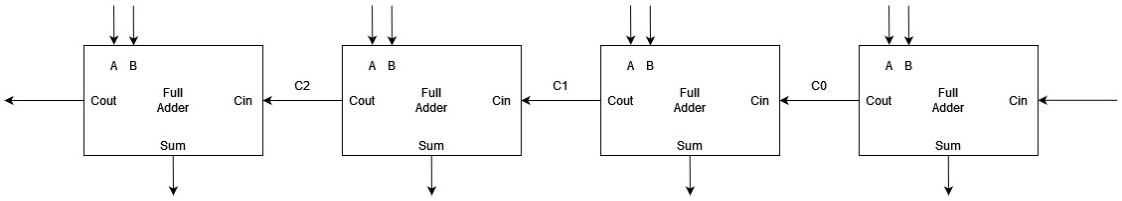


FIGURE 13 – Full adder 4 bits

2 Segment 1

2.1 Récapitulatif session d'intro

utilisation de symboles pour la logique booléenne :

Boolean	Name
X = A.B	AND
X = A+B	OR
X = $\overline{A.B}$	NAND
X = $\overline{A+B}$	NOR
X = $A \oplus B$	XOR
X = $\overline{A \oplus B}$	XNOR
X = $\overline{A}$	NOT

Pour reprendre l'example de l'additionneur complet :

$C_0 = (\overline{A}.B.c_{in}) + (A.\overline{B}.C_{in}) + (A.B.\overline{C_{in}}) + (A.B.C_{in})$
$C_0 = C_{in}(\overline{A}.B + A.\overline{B}) + A.B(\overline{C_{in}} + C_{in})$

Simplification C_{in}

C_{in}	$\overline{C_{in}}$	C_{in} or $\overline{C_{in}}$
0	1	1
1	0	1

Ca correspond à une fonction toujours vraie -> elle peut être symbolisée par un 1

On a donc au final : $C_0 = C_{in}(A \oplus B) + A.B.1$

On fait pareil pour S :

$S = C_{in}(\overline{A}.\overline{B} + A.B) + \overline{C_{in}}(\overline{A}.B + A.\overline{B})$

A	B	$\overline{A}.\overline{B} + A.B$
0	0	1
0	1	0
1	0	0
1	1	1

$S = C_{in}(\overline{A \oplus B}) + \overline{C_{in}}(A \oplus B)$

appelons $Y = A \oplus B$ pour transformer l'équation en : $\overline{C_{in}}(Y) + C_{in}(\overline{Y})$

$\overline{A \oplus B}$	$A \oplus B$
1	0
0	1
0	1
1	0

C_{in}	Y	$\overline{C_{in}}$	$\overline{Y}$
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	0

C_{in}	Y	$\overline{C_{in}}$	$\overline{Y}$	$\overline{C_{in}}.Y$	$\overline{Y}.C_{in}$	$\overline{C_{in}}.Y + \overline{Y}.C_{in}$
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

$S = C_{in} \oplus Y$ $S = C_{in} \oplus A \oplus B$

2.1.1 Table de Karnaugh

Pour chaque combinaison ou x est vrai dans la table de vérité, on inscrit 1 dans le tableau de Karnaugh

Exemple 1 On cherche à trouver la porte logique associé à cette table de vérité :

a	b	X
0	0	1
0	1	1
1	0	1
1	1	0

On construit la table de Karnaugh associée :

X	$\overline{A}$	A
$\overline{B}$	1	1
B	1	0

FIGURE 14 – Table de Karnaugh

- S'il existe des cases à "1" adjacentes, alors une simplification est possible. Ici on voit que les cases "$\overline{A}.\overline{B}$" et "$\overline{A}.B$" sont à 1, le résultat dépend donc de $\overline{A}$. La prmière partie du résultat est donc : $X = \overline{A}$. Il reste donc 1 case à "1" , on a alors : $X = \overline{A} + A.\overline{B}$

a	b	$\overline{A}$	$\overline{B}$	$A.\overline{B}$	X
0	0	1	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	1	0	0	0	0

On observe une porte "NAND" grace à la table de vérité de X

Dans le cas où les cases ne sont pas adjacentes et qu'aucun regroupement n'est possible, les opérations sont dites "exclusives"

X	$\overline{A}$	A
$\overline{B}$	0	1
B	1	0

L'équation associé est alors $A \oplus B$

Si on trace la table de vérité de X on obtiens :

a	b	X
0	0	0
0	1	1
1	0	1
1	1	0

Récap : - cases à "1" adjacentes horizontales ou verticales : porte "OR" - cases à "1" adjacentes diagonales : porte "XOR"

Example 2 : exo full adder en partant de la table de Karnaugh
On essaye de construire la table de karnaugh de la sortie S de l'additionneur complet :

A	B	C_{in}	$C_{in} \oplus A \oplus B$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

	$\overline{BC_{in}}$	$\overline{B}C_{in}$	BC_{in}	$B\overline{C_{in}}$
$\overline{A}$	0	1	0	1
A	1	0	1	0

$$S = A.\overline{B}.\overline{C_{in}} \oplus \overline{A}.\overline{B}.C_{in} \oplus A.B.C_{in} \oplus \overline{A}.B.\overline{C_{in}} \quad (1)$$
$$= C_{in}.(\overline{A}.\overline{B} \oplus A.B) \oplus \overline{C_{in}}.(A.\overline{B} \oplus \overline{A}.B) \quad (2)$$
$$= C_{in}.(\overline{A \oplus B}) \oplus \overline{C_{in}}.(A \oplus B) \quad (3)$$
$$S = A \oplus B \oplus C_{in} \quad (4)$$

On construit maintenant la table de Karnaugh pour la sortie Cout :

	$\overline{BC_{in}}$	$\overline{B}C_{in}$	BC_{in}	$B\overline{C_{in}}$
$\overline{A}$	0	0	1	0
A	0	1	1	1

$$S = B.C_{in} + A.\overline{B}.C_{in} + A.B.C_{in} \quad (5)$$
$$= B.C_{in} + A.(\overline{B}.C_{in} + B.\overline{C_{in}}) \quad (6)$$
$$= B.C_{in} + A.(B \oplus C_{in}) \quad (7)$$
$$= B.C_{in} + A.B \oplus A.C_{in} \quad (8)$$

2.2 TD : multiplication par 2 d'une entrée sur 3 bits

On effectue un décalage vers la gauche

entrée : A[0, 1, 2] sortie : B[0, 1, 2, 3]

On peut utiliser directement des câbles pour effectuer le décalage vers la gauche. Ex :

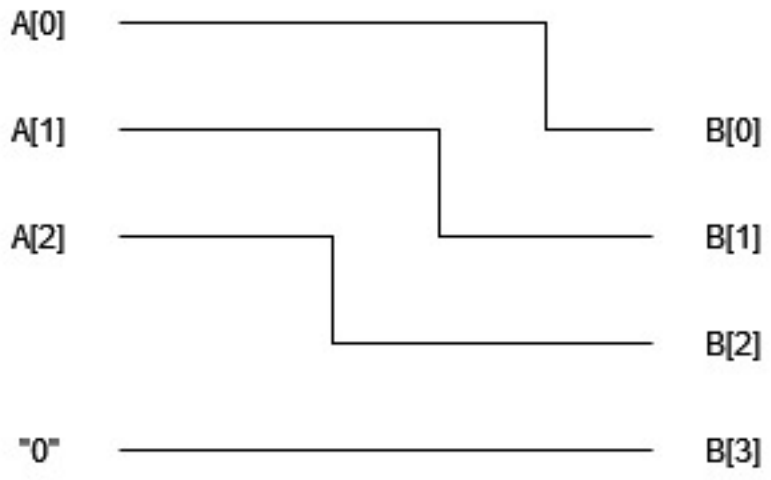


FIGURE 15 – Schéma RTL multiplication par 2

Avec "A[0]" et "B[0]" les Most Significant Bit des 2 entiers.

3 Présentation du FPGA

Base d'architecture d'un calculateur, mémoire et *Process Unit* (PU)

Tout calculateur, peu importe sa taille, comporte au moins un élément de mémorisation et un élément de calculs.

↪ généralement ils sont placés très proches pour éviter les corruptions de signaux
↪ il faut limiter le plus possible la distance entre la PU et la mémoire

distance moins élevée → moins de conso de courant

→ carte cora Z7 : PU + memory

FPGA → reconfigurable au niveau matériel
↪ Intéressant pour récupérer des signaux en simultanés (de plusieurs capteurs) ASIC → non reconfigurable

BRAM → emBedded Random Access Memory

Synchrone vs Asynchrone

Analyse de chemin critique
Principaux composants du FPGA : cellules logiques, mémoires BRAM, I/O
Accélérateur matériel → qqc qu'on ajoute au système pour accélérer des temps de traitement

Les opérations d'un calculateur type CPU, GPU ou micro-contrôleur

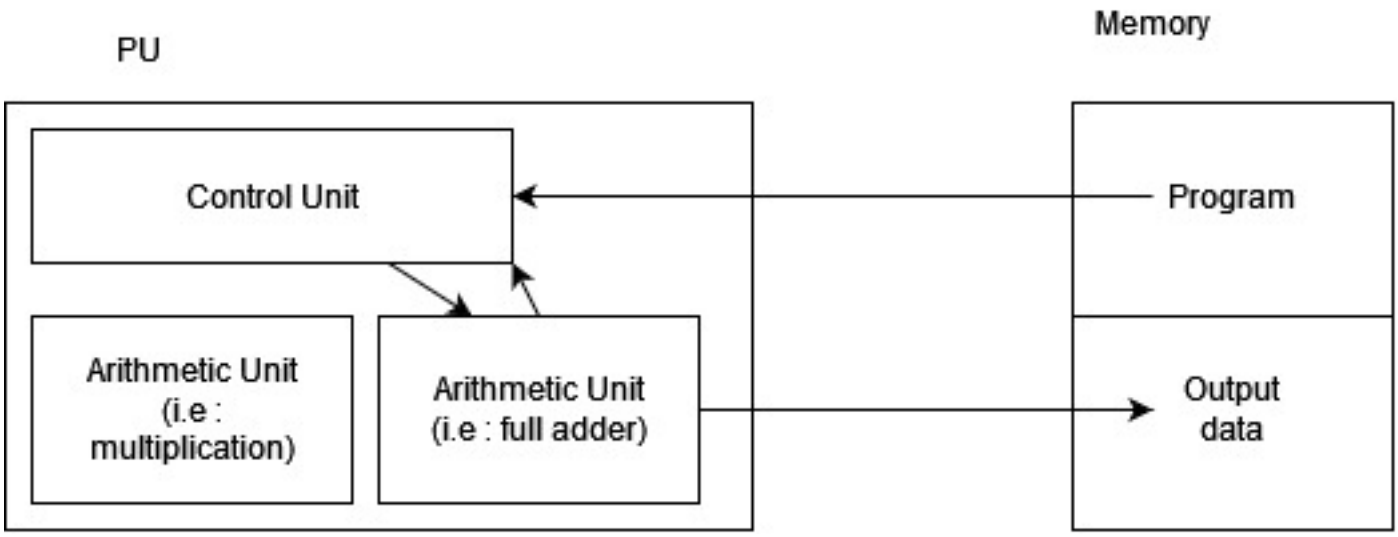


FIGURE 16 – schéma simplifié CPU

Instructions enregistrées dans la mémoire → programmation
Instructions exécutées dans la PU → inférence

⚠Les constructeurs font en sorte que les signaux sortant des composants (PU, mémoire, ...) soit plus "fort" que nécessaire pour éviter des parasites lors des transferts de données.

Etude de cas : addition de 2 chiffres par un CPU

- cf. schéma

- inférence (temps de traitement) d'une instruction : 9µs

temps mis par le programme pour exécuter 1 addition : 9 x 4 = 36µs
x 500 itérations = 18ms

Si on ajoute un autre programme pour l'addition :
- il n'y a qu'un full adder et 1 Control Unit
- les ressources sont partagées
- ça va entraîner des latences et des incertitudes sur les temps de traitement des tâches

Dans un FPGA la logique est différente, nous ne sommes pas soumis à cette séquence d'étapes qui peut être longue

Certaines fois il faut de l'électronique dédiée pour effectuer certaines tâches critiques qui prendrait trop de temps en logiciel
↪ cf. schéma slide 80
↪ problème : une fois configuré, le FPGA n'est bon que pour faire ce qu'on lui a demandé

TD : temps de propagation pour une addition sur FPGA

Porte logique : 5ps
chemin le plus long C_{out} : 15ps (3 portes)

S = 10 ps

pour 500 itérations : $C_{out} = 10ns; S = 5ns$

chemin critique → chemin le plus long du circuit logique
⚠A et B arrivent en parallèle pour chaque full adder

/*26/04/2023*/

3.1 VHDL

Langage de description matériel :

- Hardware Description Language (HDL)
- description des entrées, sories et du comportement d'une architecture numérique
- 3 princiqpaux langages : VHDL, Verilog et System Verilog

ex : porte "AND"

VHDL :
↪ couramment utilisé dans les systèmes complexes
↪ plus d'outils (pour la simulation, ...)

Verilog : plus pour le bas niveau

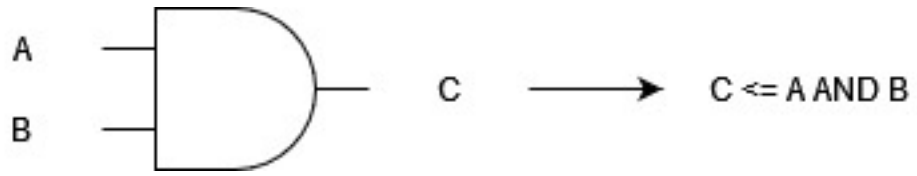


FIGURE 17 – VHDL porte "ET"

System Verilog : Extension du verilog

Choix du langage HDL : plus par affinité mais VHDL est "le mieux"

Verilog et System Verilog sont plus souvent utilisé dans les pays anglo saxons

System Verilog : Principalement utilisé pour des tests complexes , ajout d'orienté objet dans les tests,et fonctionnalité de plus haut niveau que le Verilog

Démo Vivado

Run synthesis → première étape pour passer de la partie code à la partie architecture

Run implementation → generate Bitstream → ce qui va être envoyé sur la carte

synthesis : "crée" l'architecture décrite par le code
↪ implémentation : place les élément de l'archi générée sur la carte
↪ generate Bitstream → crée un binaire à placer sur la carte
↪ open hardware manager → auto-connect
↪ dans le panneau flow navigator
↪ Program device

3.1.1 Structure d'un code VHDL

entity : décrire l'architecture de la boite
architecture : décrit le comportement de l'entité

- On ne peut pas modifier les entrées
- On ne peut pas lire les sorties

STD_LOGIC : 1 bit *STD_LOGIC_VECTOR* : Vecteur de STD_LOGIC

- And : `C <= A and B`
- Or : `C <= A or B`
- Xor : `C <= A xor B`
- Not : `C <= not A`

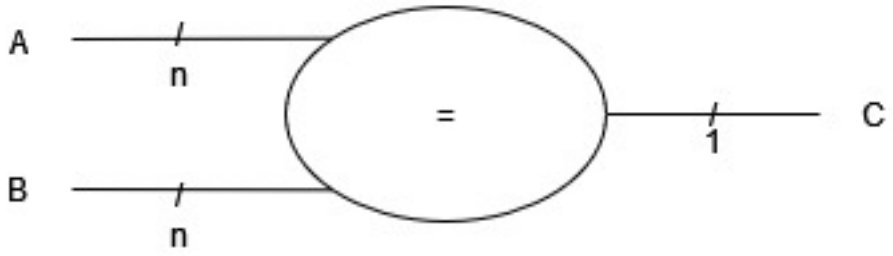


FIGURE 18 – Exemple condition VHDL

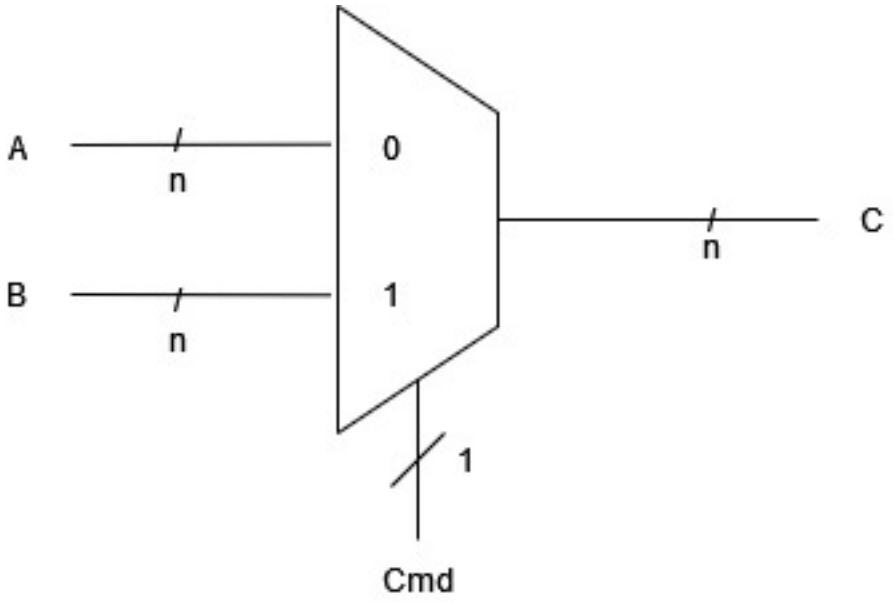


FIGURE 19 – Exemple multiplexeur

Tests et sélections

Conditions (=, <, >)

Si A = B alors C = 1, sinon C = 0
"n" : nombre de bits des entrées A et B

Multiplexeurs

Si Cmd = 0 alors C = A sinon C = B

3.1.2 Les différents types de logiques

Séquentielle

— implique l'utilisation d'une horloge
— Registres, compteurs, machines à états
— la sortie dépend des états précèdent et actuel des signaux

Combinatoire

— La sortie dépends uniquement des états actuels des signaux
— Portes logiques, multiplexeurs, comparateurs, etc.

Instruction séquentielles et concurrentes

Instructions concurrentes :
↪ pas d'ordre de priorité entre 2 instructions concurrentes
↪ en dehors du process
↪ Si 2 instructions tentent d'affecter le même signal, cela provoque un conflit (valeur 'X')

Instructions séquentielles :
↪ Dans le corps d'un process
↪ instructions interprétées successivement
↪ Si 2 instructions tentent d'affecter le même signal → fonctionne car les instructions sont interprétées successivement → dernière affectation prise en compte

Process
Les signaux sont mis à jour uniquement à la fin du process

ex : `process(clk, resetn)`

(clk, resetn) -> liste de sensibilité (stimuli). Le process serra réveillé, lorsqu'un ou plusieurs de ces signaux aura un changement de valeur.

Registre

Élément de mémoire
previous_data prendra la valeur de data uniquement sur le front montant de clk

Si data est une entrée sur plusieurs bits : à la synthèse on aura n registres pour les n bits de data

3.1.3 Machine à états (Finite State Machine, FSM)

Permet de représenter les différents modes de fonctionnement du système

machine à états → séquentielle

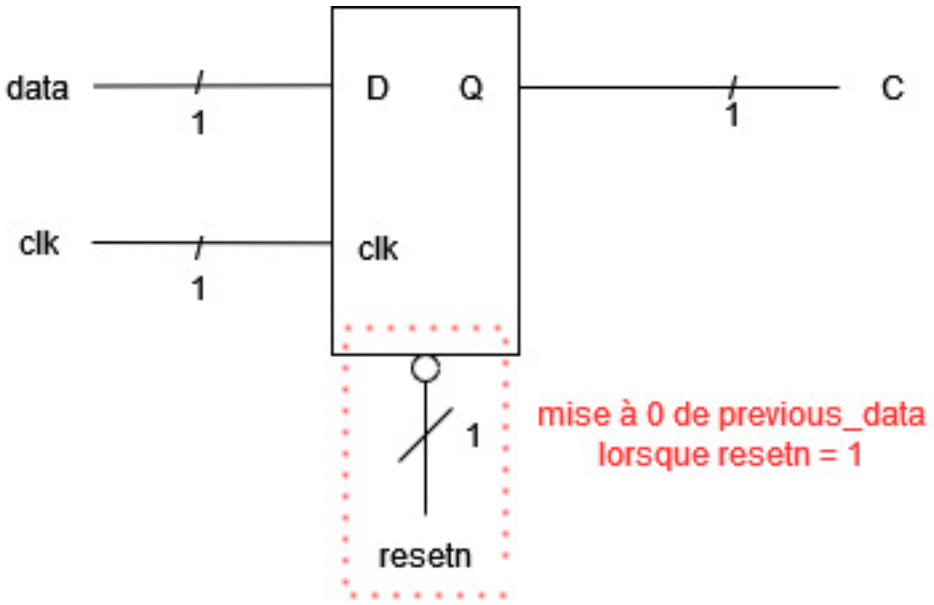


FIGURE 20 – Registre

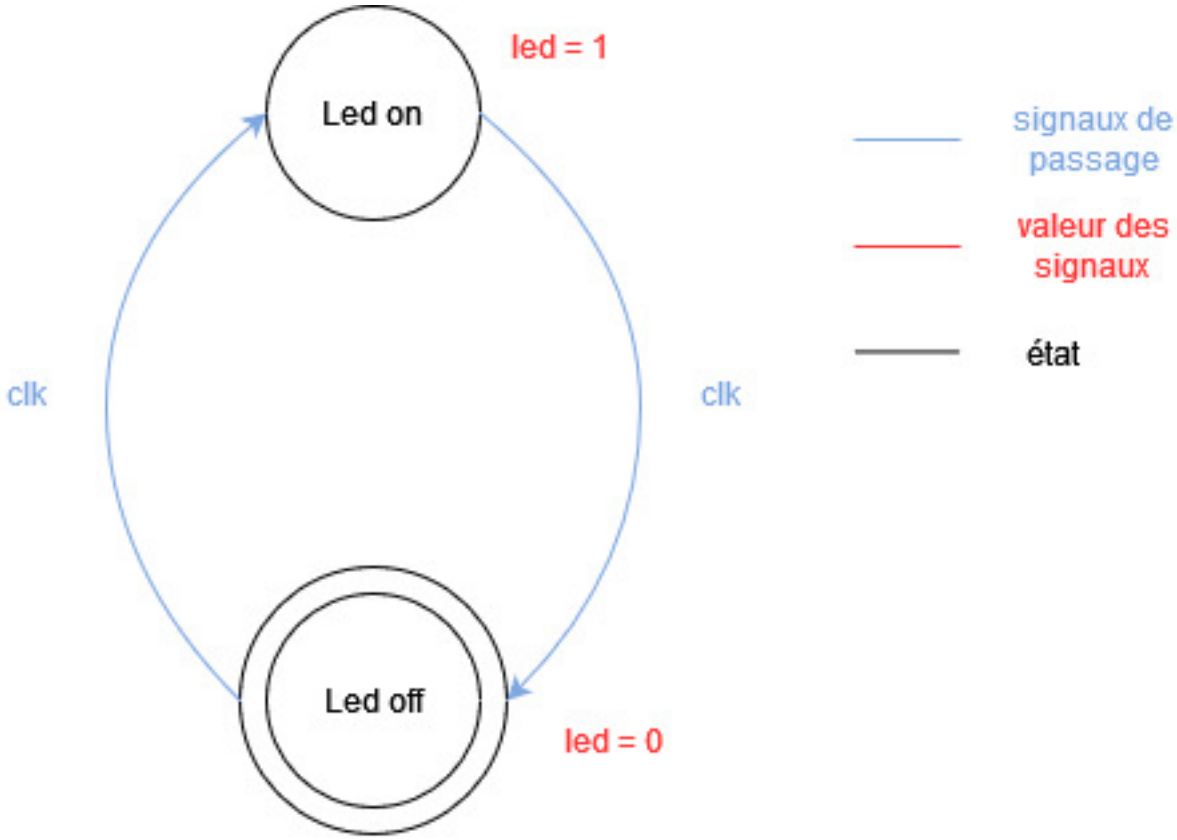


FIGURE 21 – Exemple de machine à états

3.1.4 Retour sur process

Process synchrone → registres

Process combinatoire :

La liste de sensibilité doit contenir tous les signaux d'entrée de ce bloc combinatoire

On utilise un process quand on veut effectuer une action lorsqu'un signal change

3.1.5 Décalage

- vers la droite avec insertion

```
output <= bit_msb & data[n-1:1]
```

ex : data : 100 1110 et bit_msb = 1 -> output : 1100 111

- vers la droite avec rotation

⚠l'opérateur '' est utilisé pour la concaténation en VHDL

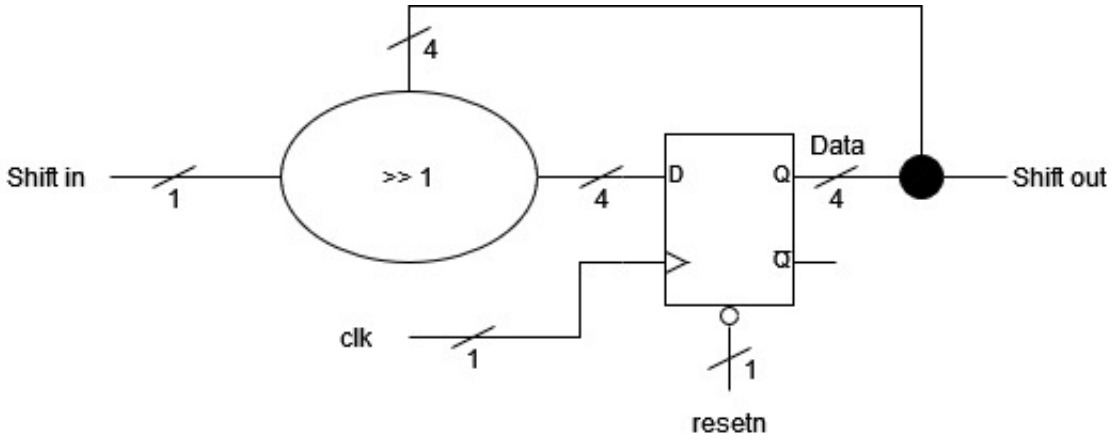


FIGURE 22 – Exemple de registre à décalage 4 bits

3.1.6 TP : Full adder en VHDL

3.1.7 Synoptique

Schéma avec plusieurs niveaux d'abstractions

flot de dév :

1. Commencer par le schéma RTL / synoptique
2. Retranscrire le schéma RTL en code VHDL
3. Fixer les contraintes du système

4. Vérifier le comportement du système en simulation
5. Étudier la synthèse
6. Placer des sondes avec l'ILA
7. Étudier le placement routage (Place And Route -PAR-)
8. Vérifier le comportement du système sur carte

Essayer de toujours avoir un schéma RTL en accord avec le code VHDL associé.
↪ Si le code ne fonctionne pas et on fait des modifs → modifier le schéma RTL

Si le rapport de synthèse ne correspond pas au schéma RTL → problème lors de l'écriture du code
↪ possibilité de ne pas avoir le comportement attendu

ILA → interface dédiée au debugage sur carte

3.1.8 Simulation VHDL (TestBench)

Cf. slide 24

Permet de tester le comportement de l'archi lors d'une simulation
Pilote les entrées du design
Peut réaliser des tests automatiques

↪ TestBench → process sans liste de sensibilité
↪ exécuté en boucle à partir de $t = 0$

Test auto avec 'assert'
↪ ex : assert output = '1' -> le signal output vaut-il '1' à cet instant ?

'report' -> pour voir où il y a une erreur (un peu comme un print)

mettre un 'wait' à la fin du process -> permet de l'empêcher de boucler

possible de déclarer des signaux internes au testbench qui seront utilisé pour les tests

'assert' fonctionne en paire avec 'report'

TP 1 : full adder en VHDL

/*27/04/2023*/

3.2 Retour sur le composant FPGA

Outils de description matériels -> HTL coder de Matlab

rétro engineering -> parfois il y a des attaquant qui cherchent des failles en cherchant à recréer le code de description (ex : VHDL) à partir du bitstream

mécanisme de défense (sécurité) -> chiffrement du bitstream

projet open source : YosysHQ/yosys (GitHub)

Retour sur l'étude de cas Full adder : Temps de Propagation

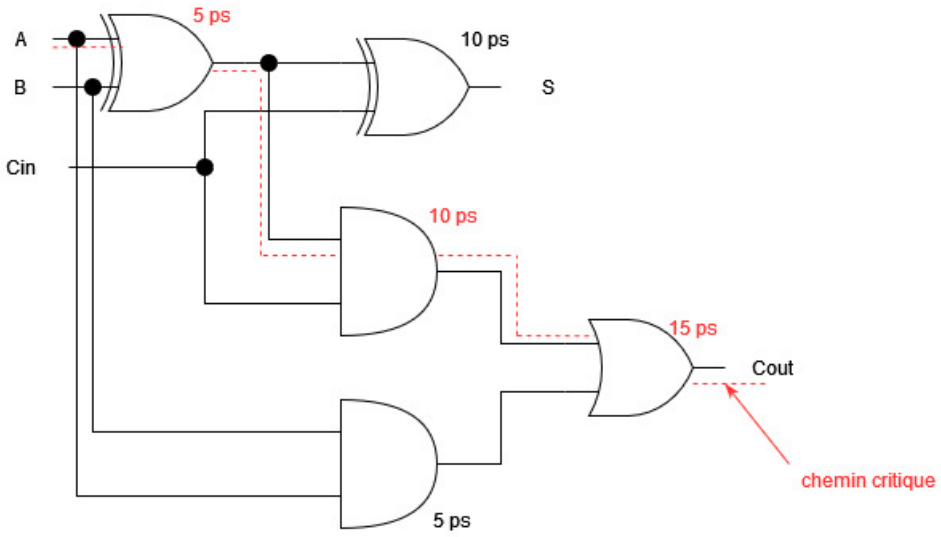


FIGURE 23 – Chemin critique full adder

Limite de fonctionnement du système -> les signaux doivent s'établir dans un temps plus long que le temps de propagation des portes logiques

système complexe -> on fait un "arbitrage" en fonction de la criticité des tâches que le système doit accomplir

FPGA -> utilise des tables de routage

Net matching -> garantir que tous les câbles arrivent en même temps à leur destination (ex : bus de données)

Full adder :
latence sur C_0 -> entraîne de la latence sur C_1
↪ C_1 -> entraîne de la latence sur C_2
↪ C_2 -> entraîne de la latence sur C_{out}

On dit que C_0 se transfert du "full adder 0" au "full adder 1"
↪ latence de $S1$ et C_1 : $S1$ = 20 ps ; C_1 = 25 ps
↪ latence de $S2$ et C_2 : $S2$ = 30 ps ; C_2 = 35 ps
↪ latence de $S3$ et C_3 : $S3$ = 40 ps ; C_3 = 45 ps

Autre cas : On souhaite effectuer 2 calculs d'additions
↪ on parallélise les traitements
↪ On utilise 2 fois plus de Process Units, aussi appelées Logical Elements dans le contexte d'un FPGA

- fichier de contrainte :
↪ placement d'I/O
↪ placement de timing

Buffer (électronique) : permet de passer de niveau de tension dit "de coeur" (interne au FPGA) à un niveau dit "d'interface" (I/O)

On peut configurer une I/O en tant qu'entrée et sortie en même temps

On ne travaille pas avec des tensions de 3,3V/5V dans le FPGA ↪ plus de l'ordre de 0,95V/0,75V (tension < 1V)

3.2.1 Slice

Logical Element (LE) / parfois aussi Logiciel Cell (LC)
↪ regroupement de composant logiques bas niveau
↪ chaque FPGA va avoir sa propre archi de "slice" lui conférant des performances différentes

ex : "Carry4"" permet de construire de manière plus "optimale" des compteurs (voir *fpga4fun*)

- Look Up table (LUT) : permet de reproduire le comportement des portes logiques

- Multiplexeur : permet de faire de l'aiguillage des signaux

- flip flop (registre) : élément de mémorisation.

- LUT : latence plus "élevée" qu'une porte logique

Info : site *mouser.com* pour les datasheets des composants

3.3 Électronique numérique fondamentale, le MOSFET

le transistor MOS -> base de tout circuit numérique

Le transistor laisse passer le courant si Vth(threshold) est plus petit que Vgs (tension à la gate)

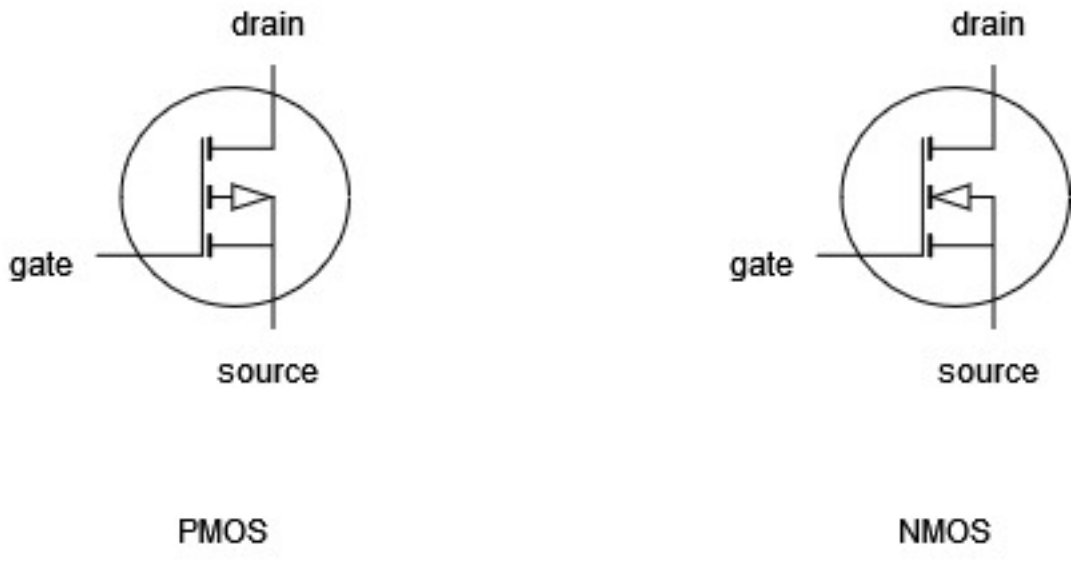


FIGURE 24 – transistors MOSFET P et N

MOS N :
- Vgs >= Vth -> le courant passe (interrupteur fermé) entre drain et source
- Vgs < Vth -> le courant ne passe pas (interrupteur ouvert) entre drain et source

MOS P :
- Vgs >= Vth -> le courant ne passe pas (interrupteur ouvert) entre drain et source
- Vgs < Vth -> le courant passe (interrupteur fermé) entre drain et source

TD : Mosfet sut LTSpice (Modélisation et simulation de portes logiques)

Cf. Td sur cahier + fichiers LTSpice

3.3.1 Look Up Table

On peut associer la LUT à une mémoire qui possède un bus d'adresse sur lequel on connecte nos entrées

peut s'apparenter à de la RAM

plus pratique de raisonner en terme de porte logique qu'en terme d'adresses

L'outil de synthèse s'occupe de la configuration de la LUT

la LUT induit "beaucoup de latence"

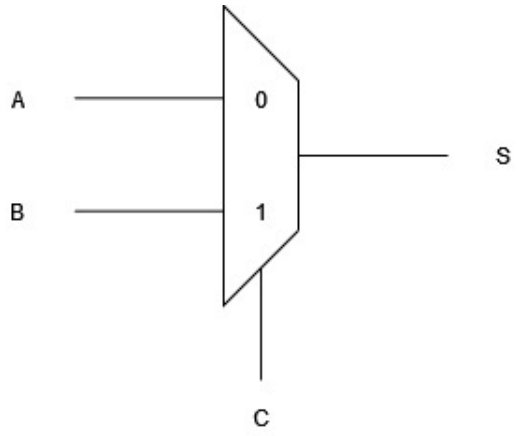
ASIC : Application Specific integrated Cirtcuit

Synopsys -> propose des services de desgin FPGA

Prendre en compte le temps d'initialisation des LUT Possible de reconfigurer les LUTs à la volée (sujet de recherche)

schéma représentatif de la LUT slide 130
↪ Cascade de multiplexeur -> permet de définir la sortie

Comment réaliser un multiplexeur (à 2 entrées) avec des portes logiques?



	$\overline{AB}$	$\overline{A}B$	AB	$A\overline{B}$
$\overline{C}$	0	0	1	1
C	0	1	1	0

$S = A.\overline{B}.\overline{C} + A.B.\overline{C} + \overline{A}.B.C + A.B.C$
$= \overline{C}.(A.\overline{B} + A.B) + C.(\overline{A}.B + A.B)$
$= \overline{C}.(A.(\overline{B} + B)) + C.((\overline{A} + A).B)$
$= \overline{C}.A + C.B$

-> Au vu de la structure de la LUT :
↪ cascade de multiplexeur
↪ chaque multiplexeur à une latence
↪ la latence se cumule sur tous les multiplexeurs de la cascade

Doit-on toujours utiliser la RAM pour écrire les LUT?

-> Pas forcément -> on peut aussi utiliser de la flash [1]

-> SRAM (Static RAM) -> se comporte comme de la RAM

SRAM (besoin que d'1 initialisation) DRAM (besoin de rafraîchissement)

1. https://www.microsemi.com/document-portal/doc_download/129911-designing-for-performance-on-flash-based-fpgas

Flash -> plus lente et plus "grande" (prend plus de place sur la puce) qu'une cellule RAM (-> RAM - grosse et + rapide)

Temps de programmation -> mise en service du FPGA

Il existe des automates gravés dans le silicium (dont on a pas la main) et qui s'occupent de charger le design depuis la flash dans le FPGA (un peu à la manière d'un bootloader pour un OS ?)

Il existe des FPGA à base de fusibles
↪ quasi impossible de faire du rétro engineering dessus
↪ on perd le côté reconfigurable
↪ peut être utilisé dans des satellites
↪ ne peut pas être corrompue par "les radiations" (ou difficile à corrompre)

3.3.2 Les types de mémoires modernes

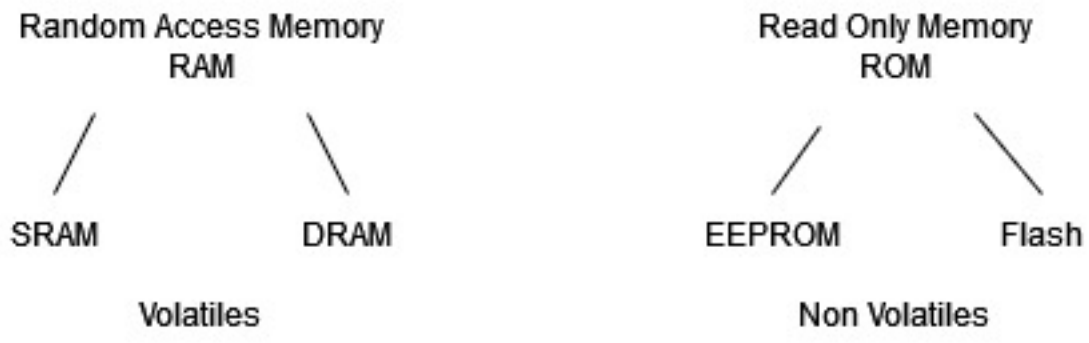


FIGURE 25 – Types de mémoires

RAM		ROM	
⊕	⊖	⊕	⊖
densité : quantité de Logical Element (LE) que l'on peut placé sur la puce		moins sujet aux attaques matérielles	cadencé moins vite
accès : plus rapide d'accès que la ROM		pas besoin de stockage externe	

TD : Construire un full substractor 4 bits

(Cf. slide 144)

Table de vérité sur excel (à ajouter)

équation logique :

$$D = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B_{in}} + A.\overline{B}.\overline{B_{in}} + A.B.B_{in} \quad (9)$$
$$= B_{in}.(\overline{A}.\overline{B} + A.B) + \overline{B_{in}}.(\overline{A}.B + A.\overline{B}) \quad (10)$$
$$= B_{in}.(\overline{A \oplus B}) + \overline{B_{in}}.(A \oplus B) \quad (11)$$
$$= A \oplus B \oplus B_{in}(Cf.fulladder) \quad (12)$$
$$(13)$$

$$B_{out} = B_{in}.(\overline{A \oplus B}) + \overline{A}.B \quad (14)$$

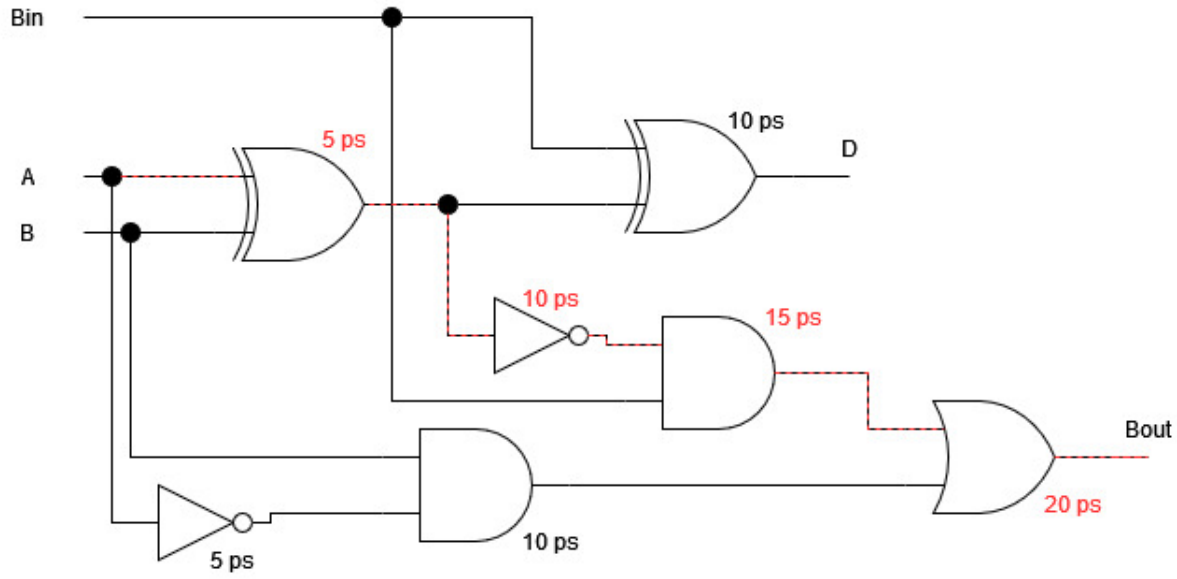


FIGURE 26 – correction full substractor + chemin critique

/* 02/05/2023*/

3.4 Logique numérique synchrone

- La bascule (latch)
- Le registre (flip-flop)
- Les compteurs et registres à décalage
- Machines à états (Finite State Machin / FSM)

3.4.1 La bascule (Latch)

C'est un élément de mémorisation dit "latché"

Les états stables d'un latch sont appelés "set" et "reset"

Si les entrées (R et S) sont à "0", la sortie ne change pas (conserve sa valeur précédente)

-> latching(locker)

S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

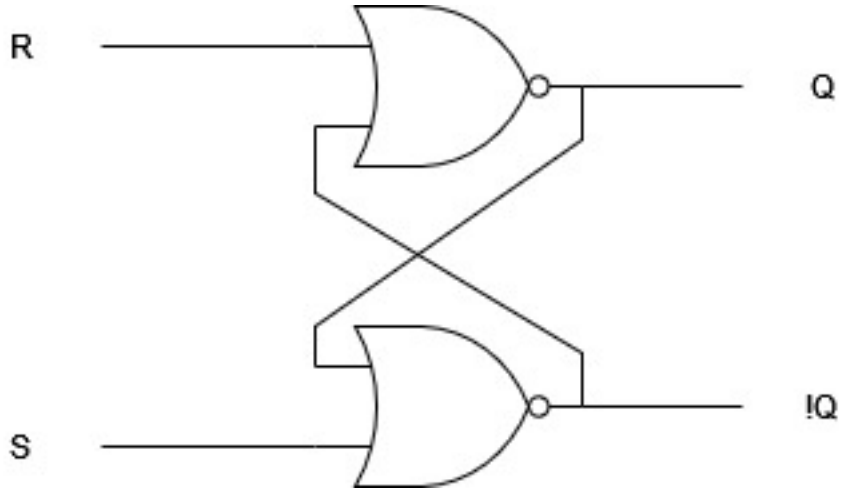


FIGURE 27 – Schéma RTL bascule

⚠on considère le cas S = 1 et R = 1 illégal
↪ on l'écrit dans la table de vérité pour décrire l'ensemble des cas mais c'est redondant avec le cas : S = 0 et R = 0

TD : Réaliser une bascule sur LTSpice

nom du composant sur LTSpice : *SRflop*

2 cas à gérer :

- S = 0 et R = 1 puis S = 0 et R = 0
- S = 1 et R = 0 puis S = 0 et R = 0

observer Q dans chaque cas

/// mettre les captures ici?////

3.4.2 Le registre (flip-flop)

Les registres ajoutent de la synchronicité au *Latch* avec des portes "AND" et un signal "clk" (horloge)

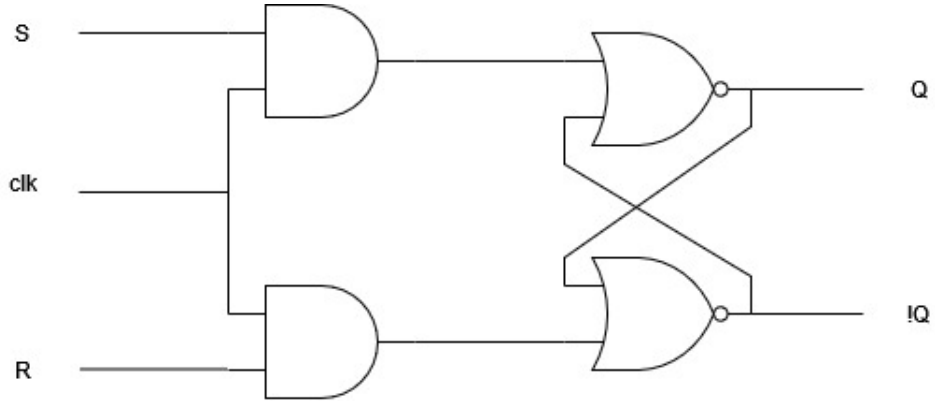


FIGURE 28 – Caption

TD : Réaliser un registre sur LTSpice

nom du composant sur LTSpice : *dflop*

Le reset d'un registre est dit "asynchrone"
↪ on peut clear le registre sans se soucier de l'horloge

Notes

Analyse de timing :
-> l'outil de placement fait exprès d'induire des décalages pour éviter les "glitchs"

boites de routage -> pour induire des "déphasages" léger

resetn -> plus populaire pour des raisons de perturbations de signaux moins fréquentes quand on passe de l'état bas à l'état haut

Par convention on travail sur front montant d'horloge

front montant + front descendant -> Double Data Rate (DDR, ex : DDR-RAM)

3.4.3 Le registre T-flop

Variante de la flip-flop (d-flop)

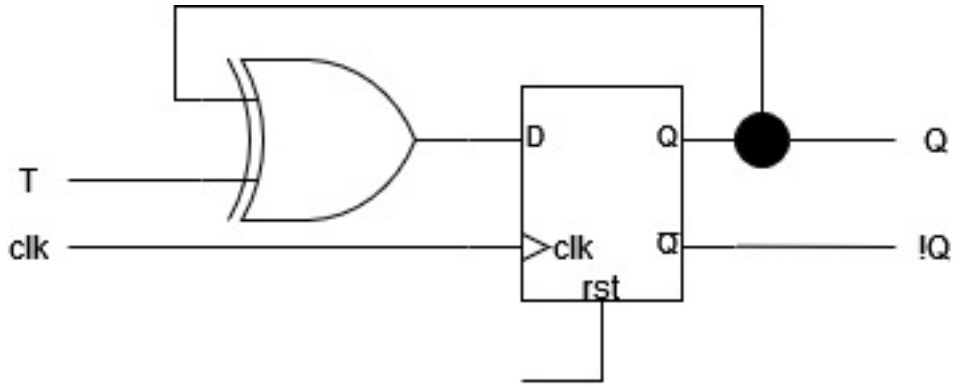


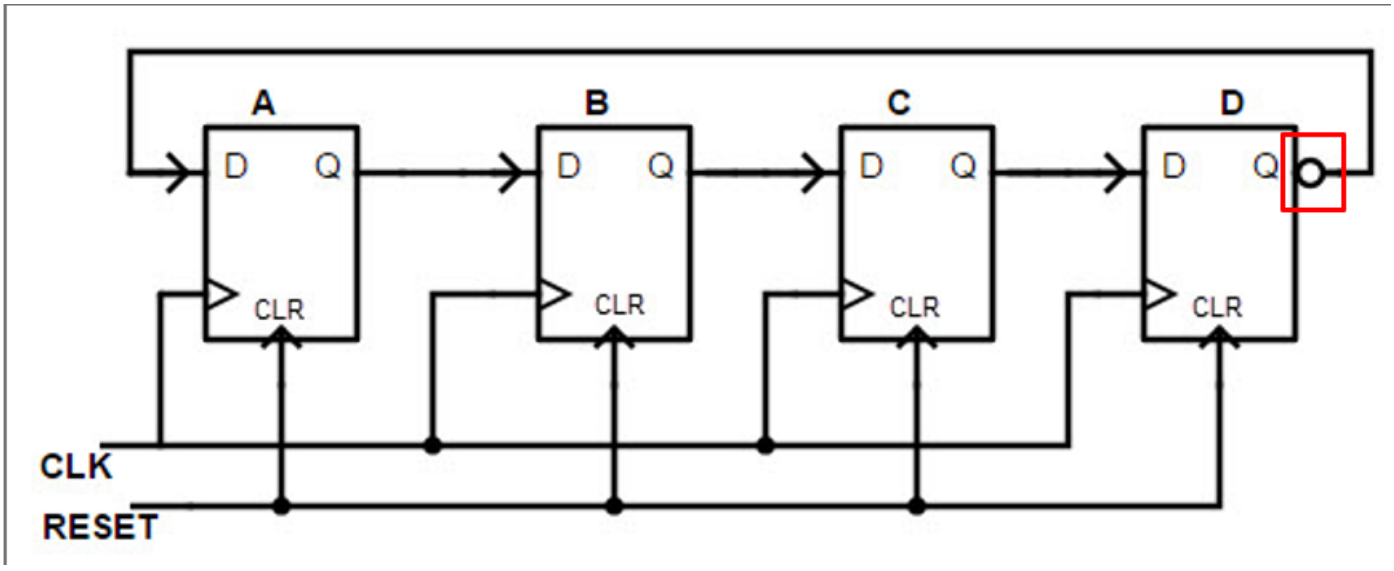
FIGURE 29 – Schéma RTL T-flop

clk	T	Q_{n+1}
↑	0	Q_n
↑	1	Q'_n

3.4.4 Les compteurs

-> très récurrents dans les designs numériques -> ex : permet de réaliser des *watchdogs* -> pour interrompre un process qui ne réponds pas au bout d'un certain temps

Exo1 : Compteur 4 bits, schéma RTL et chronogramme



C'est un **ring counter** ou **compteur de johnson**

Table de vérité du compteur de Johnson :

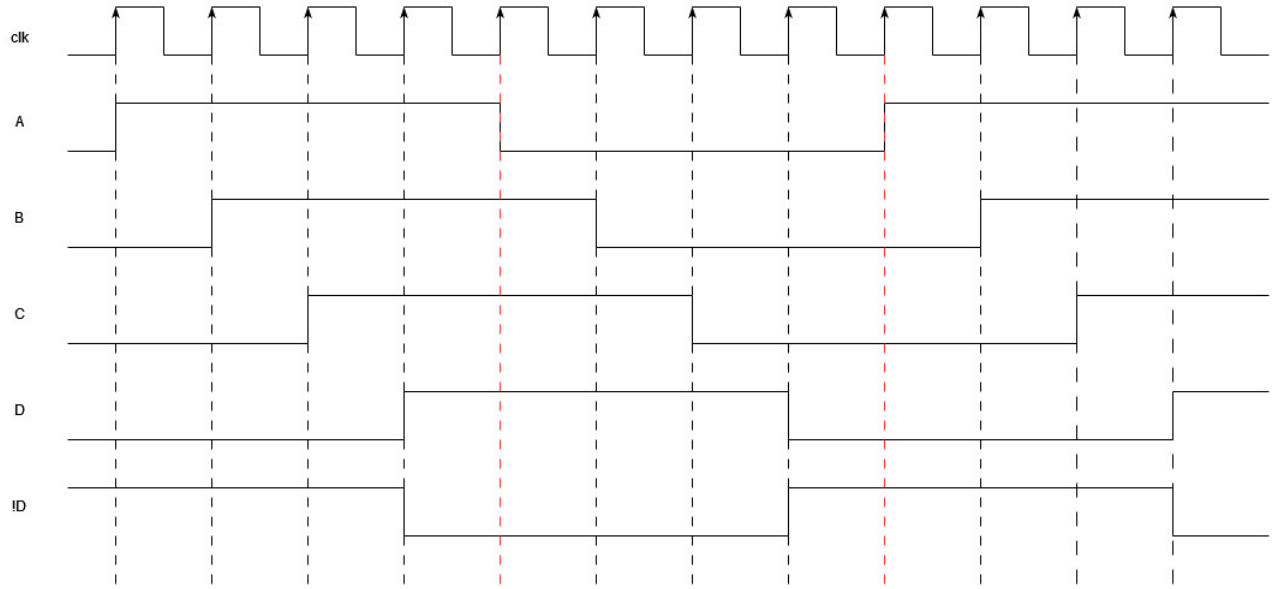


FIGURE 30 – Chronogramme compteur 4 bits

Q_A	Q_B	Q_C	Q_D
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
repeat			

représentation RTL pour un compteur de Johnson

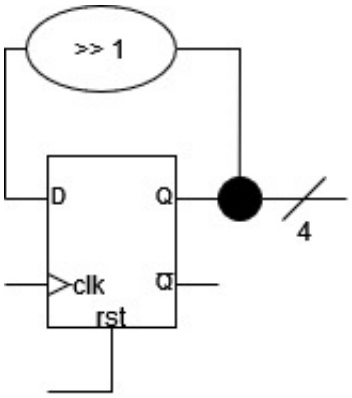


FIGURE 31 – Schéma RTL compteur de Johnson

Ex2 : compteur 4 bits entiers naturels (0 à 15)

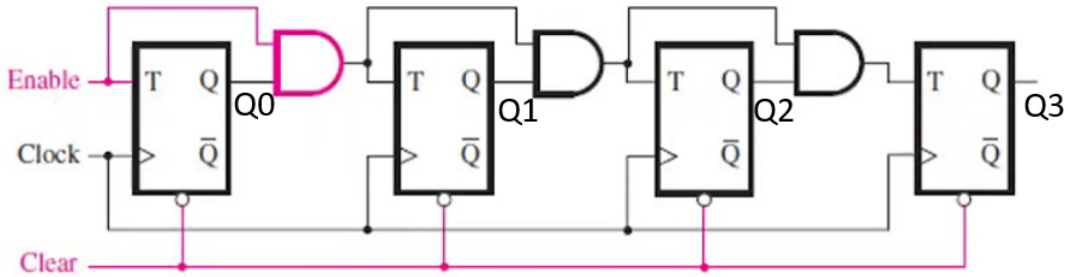


FIGURE 32 – Compteur 4 bits entiers naturels

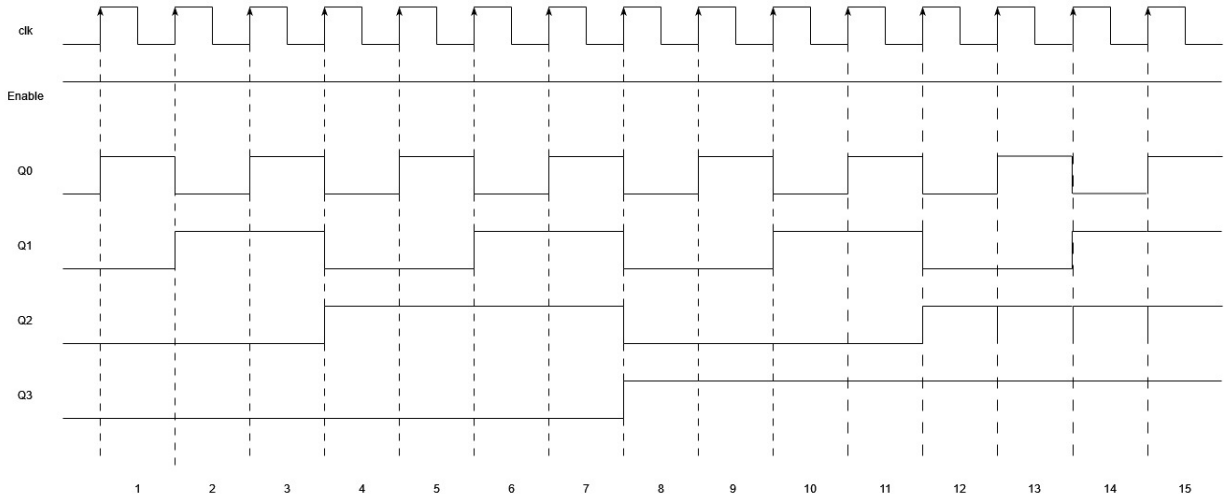


FIGURE 33 – Chronogramme compteur 4 bits entiers naturels

Correction :

Si T = 1 ; $Q_0 = \overline{Q_0}$ -> général pour la bascule T

On a alors :

$$T_1 = Q_0 \tag{15}$$
$$\tag{16}$$

$$T_2 = Q_1.T_1 \tag{17}$$
$$= Q_1.Q_0 \tag{18}$$
$$\tag{19}$$

$$T_3 = Q_2.T_2 \tag{20}$$
$$= Q_2.Q_1.Q_0 \tag{21}$$
$$\tag{22}$$

Notes : éviter d'utiliser plus de 80 % des ressources utilisable dans le FPGA pour un design (LUT, registres, slices, BRAM, etc.)

3.4.5 Machines à états (Finite State Machin / FSM)

Généralement définies par un ensemble d'états finis possibles
↪ chaque état peut avoir un ensemble de sorties associées qui représentent l'action ou le comportement du système à un moment donné

Ex : Feu rouge

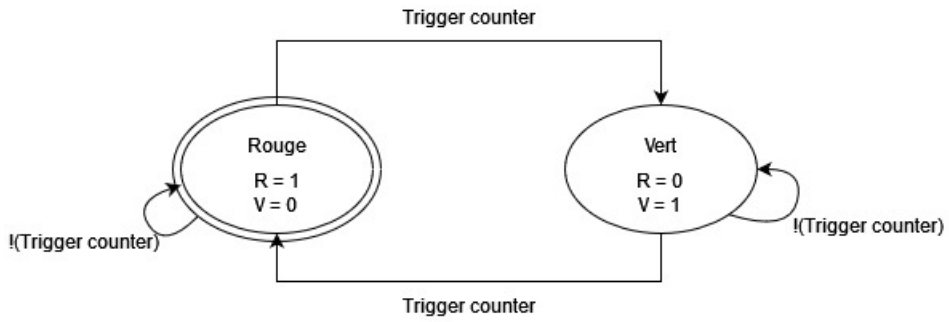


FIGURE 34 – Diagramme d'état Feu rouge

— double cercle : pour indiquer d'où on part dans le circuit
— R et V : signaux sur 1 bit qui viendront piloter les LEDS
— Quand la FSM est établie on peut faire le synoptique (Cf. synoptique)

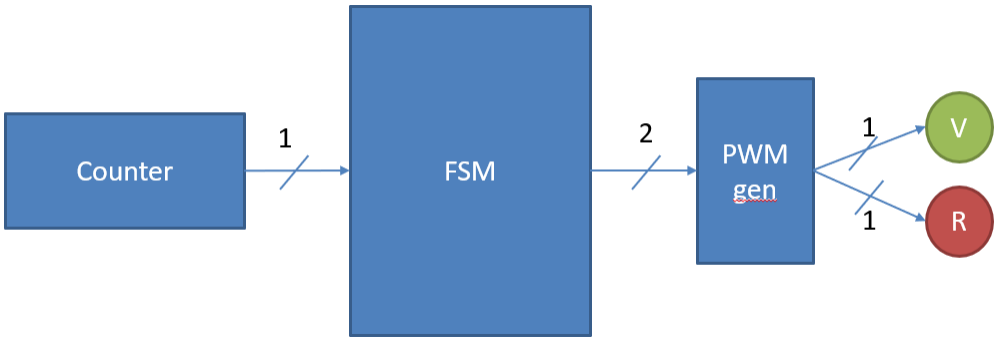


FIGURE 35 – Synoptique exemple Feu rouge

Remarques :
-> Compteur de Johnson -> bien pour faire du PWM
-> Compteur d'entier naturels -> bien pour créer un compteur de changement d'état

Il y a 2 types de FSM :
-> Machine de Moore : l'état suivant dépend uniquement de l'état courant
-> Machine de Mealy : l'état suivant dépend de l'état courant et des entrées

Il est possible d'avoir plusieurs machines à états pour un même problème

TD : machine à états

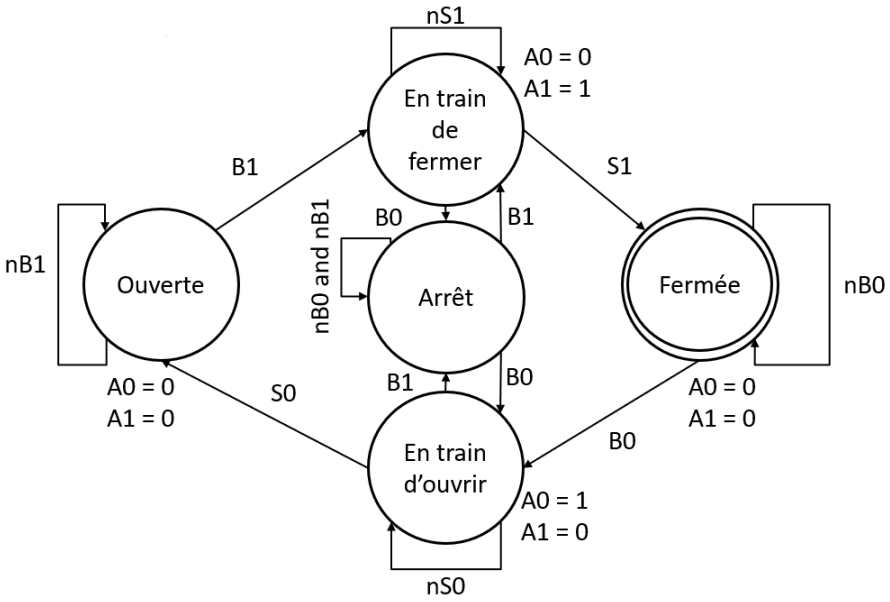


FIGURE 36 – Correction machine à états

On utilise un registre à décalage pour stocker la valeur de la séquence de 4 digits

Le bit *valid* de commande est utilisé dans le multiplexeur pour changer la valeur du registre quand un digit est entré par l'utilisateur

Le registre à décalage est utilisé pour stocker les digits entrés par l'utilisateur

On a pas besoin de *clear* les entrées à chaque nouveau code car à chaque appui on décale les digits de 1 dans le registre

Pour la vérification -> on place un comparateur entre les digits du registre à décalage

Pour l'attente de 10s :
↪ on génère un signal *trigger counter*
↪ on doit attendre 1 000 000 000 de cycle d'horloge pour atteindre 10s
↪ un signal *EnableCount* va permettre de démarrer et de stopper le compteur
↪ un signal *EnableCount* va permettre de démarrer et de stopper le compteur

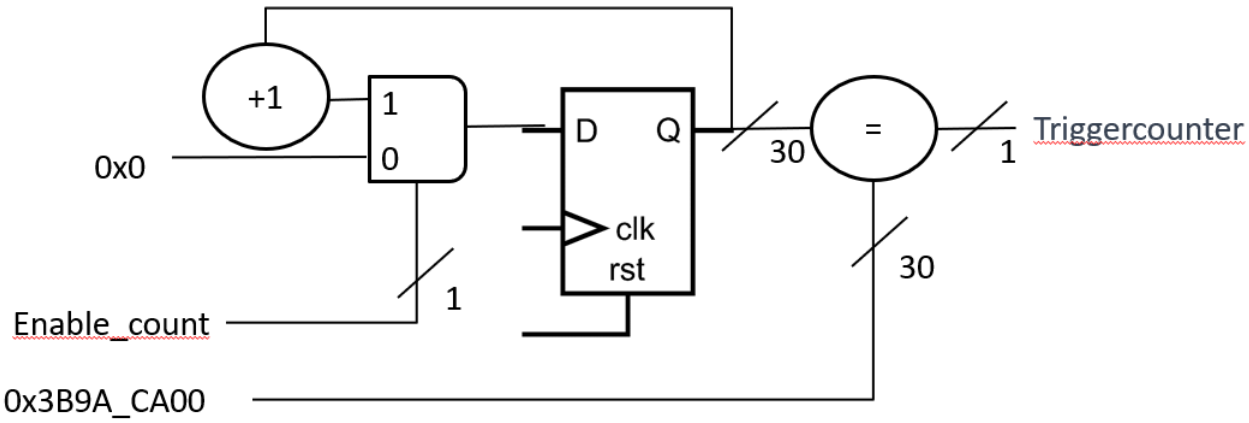


FIGURE 37 – Compteur TD1 FSM

3.5 Gestion de timing

Souvent la logique combinatoire est associée avec un logique synchrone -> que se passe-t-il si la logique combinatoire est très grande?

On reprend le cas du TD1 des FSM :

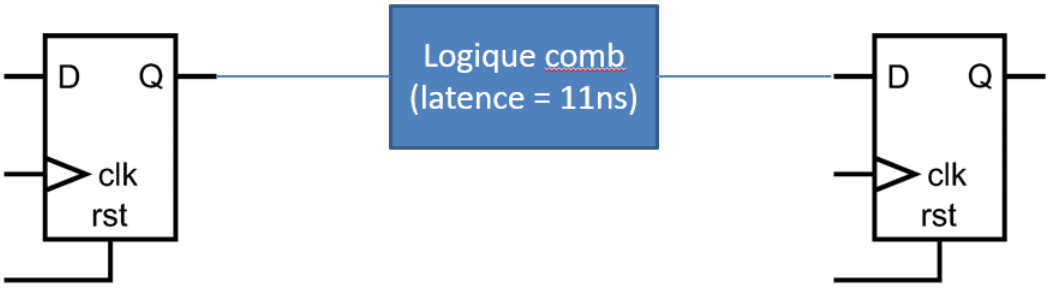


FIGURE 38 – latence TD1 FSM

-> clk = 100MHz -> on a des période de 10ns
-> la latence de la logique combinatoire est de 11ns
↪ la latence est trop grande par rapport à la période
↪ on va "rater" des fronts montant d'horloge -> ça décale les signaux

La plupart du temps c'est l'outil de *CAO* qui s'occupe des problèmes de timing

Les constructeurs ont donnés des spécifications pour résoudre ces problèmes de timing
↪ 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 d'horloge (la plage est donnée par le constructeur)

Il y a plusieurs phases pour établir la plage :
↪ avant le front montant d'horloge -> setup

↪ après le front montant d'horloge -> hold

-> *slack* -> temps "bonus" avant la phase de setup (ou de hold)
↪ peut être négatif si le signal est en retard

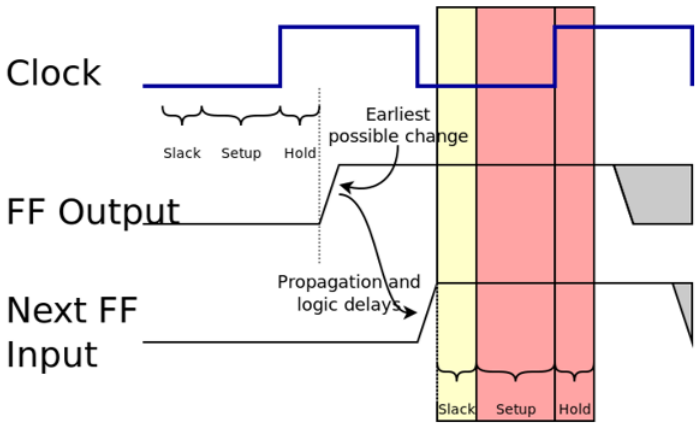


FIGURE 39 – Schéma du setup/hold/slack

Note :
les FPGAs peuvent être cadencé au max aux alentours de 400/500 MHz
↪ peut de jitter

Le signal d'horloge est amené à chaque registre "au même moment" grâce à un chemin induisant de latence

⚠Ne jamais installer de la logique combinatoire sur un signal d'horloge (sinon des désynchronisation vont apparaître dans le circuit)

Le signal d'horloge doit toujours arriver en même temps aux registres -> le logiciel de CAO s'occupe de ça

Le *clock skew* (horloge désynchronisée)
↪ les signaux d'horloge n'arrivent pas en même temps aux registres (quand il y a **plusieurs** horloges)
↪ *skew* -> delta sur l'émission des signaux d'horloges

Le *Clock Tree*
↪ spécialement conçu pour limiter le clock skew
↪ réseau particulier conçu pour de la faible latence

-> Quand on crée un design pour un FPGA
↪ on a pas à prendre en compte le clock skew qui est normalement déjà "gérer" par le constructeur
↪ on peut assumer que les signaux arrivent en même temps

⚠Ajouter de la latence au niveau des horloges peut entraîner des problèmes que le logiciel de CAO n'indique pas dans les rapports d'analyses de timing

3.6 Analyse de timing

L'analyse de timing est une méthode utilisée pour résoudre les problèmes liés au timing dans les circuits

-> Quand on a un changement d'état "pile au moment" d'un front montant on a un problème de *métastabilité* -> le signal est dans un état non logique

La métastabilité peut se propager

plus on augmente la fréquence -> plus on contraint le système -> plus on a de chances d'avoir de la métastabilité

double registre -> permet de régler des problèmes de métastabilité

règle de design pour éviter la métastabilité
↪ bonne pratique : utiliser la PLL pour gérer 2 horloges

Cross Clock Domain (CCD) : quand on a plusieurs domaines d'horloges

Clock Enable (CE) -> permet d'avoir plus de contrôle au niveau des registres

/* 03/05/2023 */

Globalement, qu'est ce qui augmente la résilience d'un système aux erreurs de timings?

↪ avoir un chemin critique le plus court possible
↪ une fréquence de fonctionnement basse
↪ un silicium de bonne facture (-4C ou -1C)

Nouvel exo : Ripple Counter

Nouvel exo : Machine à états

3.7 Les I/O dans un FPGA

-> FPGA -> offrent une grande flexibilité au niveau de la config des I/O

-> *Plots/balling* -> sous la puce FPGA. Petites broches qui servent pour les I/O

Certains pins restent dédiés à l'alim ou à la gestion d'horloge

⚠Il faut que la tension de coeur soit la même partout dans le FPGA (bien répartir la tension) -> normalement pris en charge par les constructeurs

Les I/O sont regroupées par *bank*
↪ chaque *banque* possède une config pour l'alim des buffers situés dans les PADs

3.8 Gestion de timings, cas particuliers et résolutions

Il est possible d'utiliser 2 horloges dans un système (très fréquent quand on conçoit une interface)

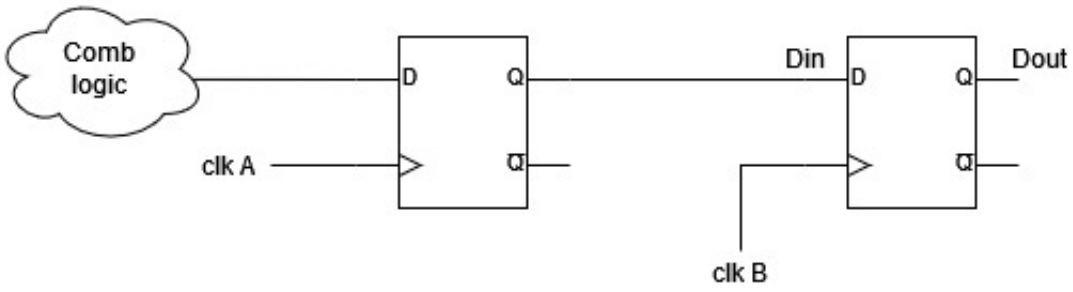


FIGURE 40 – Exemple de Cross Clock domain

Dans l'exemple ci-dessus, le $2^{ème}$ registre aura une métastabilité au niveau de D_{in} à cause de l'échantillonnage de **clk B**.

Quand on a un Cross Clock Domain, on ajoute un registre derrière le registre qui aura la métastabilité :

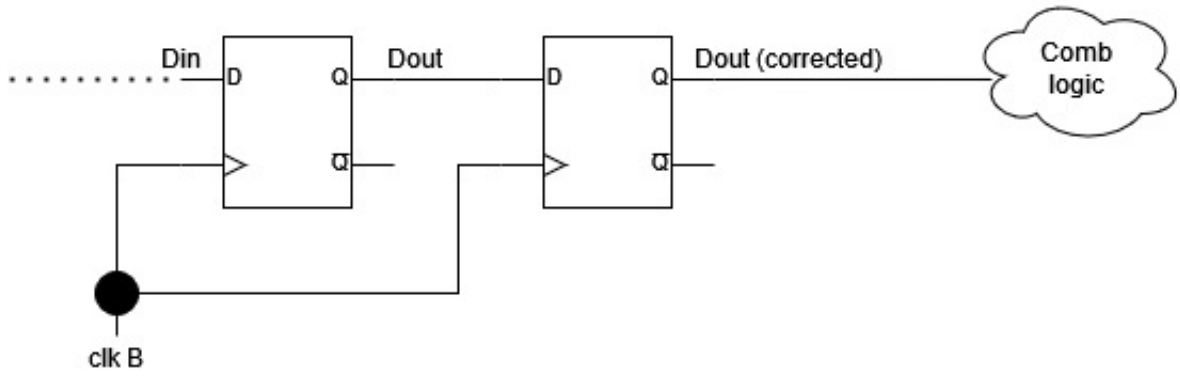


FIGURE 41 – Exemple de Cross Clock domain corrigé

double registre -> appelé "double rideau"

Dans le cas d'un CCD sur un bus de données, on peut aussi faire appel à une FIFO (attention à l'équilibre des flux de données)
↪ On peut utiliser 2 horloges : 1 pour l'écriture et 1 autre pour la lecture

3.9 Retour sur Les I/O dans les FPGA

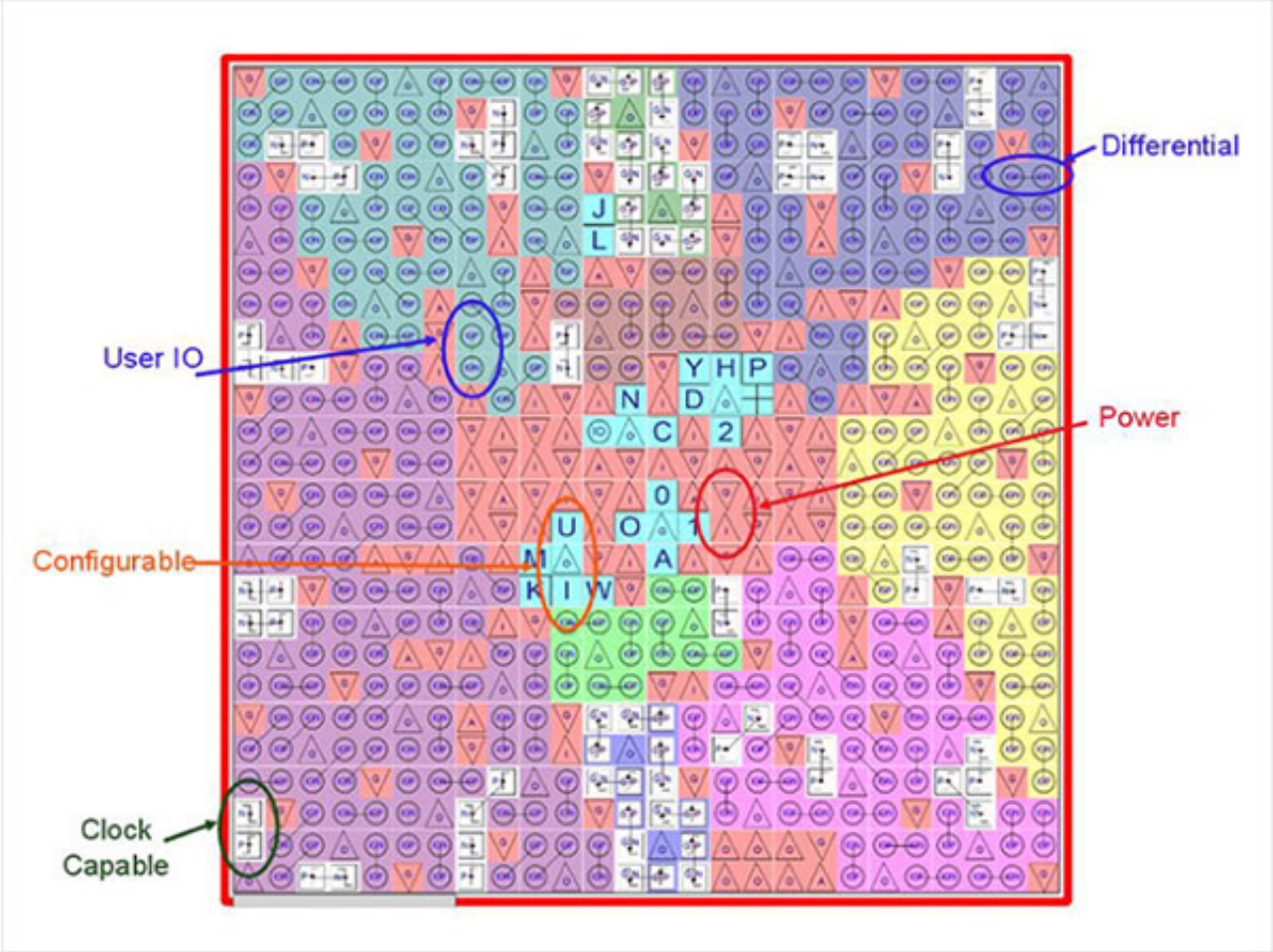


FIGURE 42 – Exemple d'agencement des plots d'un FPGA

Il existe des **composants de proximité** qui rassemblent les infos provenant de plusieurs capteurs et transferts ces infos sur 1 seul liaison plus "user friendly"

Il y a des port **VCC** et **GND** un peu partout sur la carte pour uniformiser le passage du courant.

On peut configurer les entrées / sorties selon plusieurs standards (les datasheets des fondeurs indiquent les standard supportés par les I/O des FPGAs)

3.10 Les interfaces de transmission

3.10.1 Single Ended

Utilisé dans les transmissions "lente" -> pour communiquer avec des composants comme des périphériques ou des capteurs (ex : on utilise des buffers au niveau des PADs).

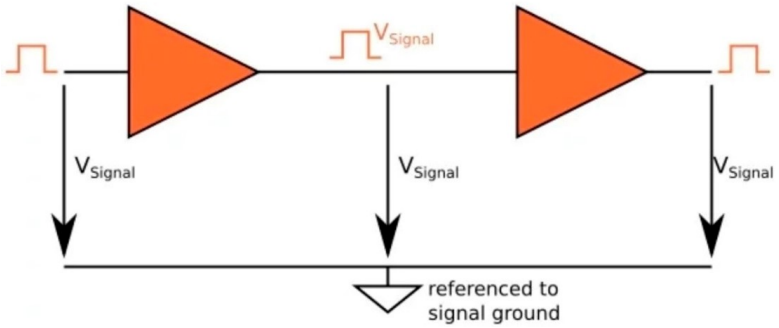


FIGURE 43 – Schéma transmission single ended

3.10.2 Différentielle

Utilise 2 PADs pour émettre un signal différentiel
↪ Il y a 2 câbles souvent appelés : S_p et S_n
↪ On a : $S_p = -S_n$
↪ Au niveau du buffer de reception on obtient le signal S en faisant :

$$S = \frac{(S_p - S_n)}{2}$$

On a alors :

$$\begin{aligned} S'_n &= S_n + item \\ S'_p &= S_p + item \end{aligned} \tag{23}$$

Comme $S = \frac{(S'_p - S'_n)}{2}$

$$S = \frac{(S_p + item - (S_n + item))}{2} \tag{24}$$
$$= \frac{(S_p + item - S_n - item)}{2} \tag{25}$$
$$= \frac{(S_p - S_n)}{2} \tag{26}$$
$$= \frac{(2S_p)}{2} \tag{27}$$
$$= S_p \tag{28}$$

Principe de la pair différentiel : ↪ liaison plus résiliente aux perturbations (ex : perturbations IEM)

⚠Il faut que les signaux S_p et S_n **arrivent en même temps** pour que la reconstruction du signal de sortie soit correcte.

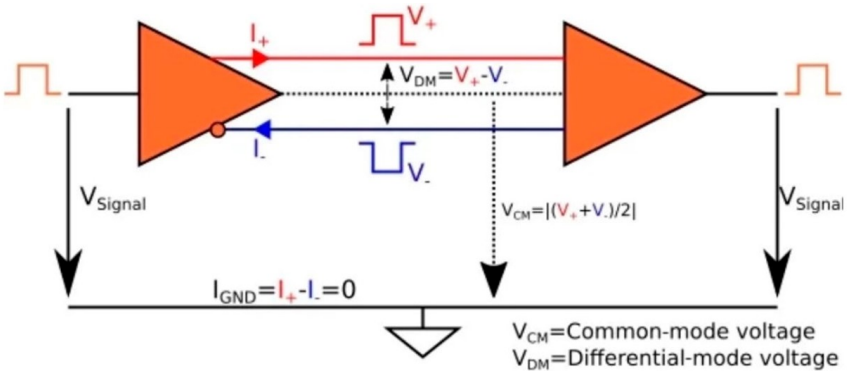


FIGURE 44 – Schéma transmission différentielle

Appairage de ligne :
- gérer les formes des lignes pour gérer la latence ou mise à niveau des câbles / composants pour que tous les composants arrivent en même temps

(Note : Pour de l'imagerie (ou autre cas) on utilise de la mémoire externe)

4 Segment 2 : Méthode de dév d'un système logique FPGA

Note : La première partie de cette section est la sous partie *VHDL* de la section 1

4.1 Tests sur carte

-> Placer des sondes dans le design pour observer le comportement des signaux sur carte

-> ILA -> permet de placer des sondes de debug sur la carte et voir les infos sur vivado ↪ ensemble des câbles sur lesquels sont placés les signaux
↪ le bitstream est généré après l'ILA car on ajoute des élt logiques pour le debug (ce qui alourdit l'archi)

-> Pour de grosses archis -> on essaye de choisir les signaux de debug pour ne pas alourdir le design "pour rien"

Il peut arriver que le testbench passe mais qu'il y ait un problème lors du déploiement sur carte -> lié à des problèmes de routage -> on utilise l'ILA pour debug ces cas

4.2 Rapport de synthèse (sous vivado)

-> Tous ce que l'outil a trouvé dans notre design -> détecte des machines à états (la section n'apparaît pas s'il n'y a pas de FSM) -> élt de logique -> détecte les ressources utilisées pour le design

FDCE -> registre à reset asynchrone

FDXX -> registres

4 types de registres :

- registres à reset asynchrone
- registres à reset synchrone
- registres à set asynchrone
- registres à set synchrone

Fichier *Urilisation* -> autre rapport "de synthèse" pour voir les ressources utilisées.

4.3 Rapports de timming

Static Timing Analysis (STA) (dans le fichier Timing summary - Route Design)

-> donnes des infos sur la "forme" de l'horloge

ex : Ici changement d'état toutes les 0 et 5 ns avec une période de 10 ns

```
------------------------------------------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------------------------------------------

Clock        Waveform(ns)       Period(ns)      Frequency(MHz)
-----        ------------       ----------      --------------
sys_clk_pin  {0.000 5.000}      10.000          100.000
```

FIGURE 45 – STA clock summary

on observe un rapport cyclique de 50% (section waveform)

Dans la section Design Timing Summary, on retrouve :

- Total Negative Slack (TNS) -> si pas à '0' -> violation de setup
- Total Hold Slack (THS) -> si pas à '0' -> violation de hold

Dans la section Max Delay Paths on retrouve le chemin critique de notre design (1^{er} chemin donné dans la section) (On peut aussi voir les chemins dans la vue *schematic* de la synthèse)

S'il y a trop de sondes on peut avoir des violations de setup/hold, il y a 2 solutions dans ce cas :

- supprimer des sondes (on préfère cette solution)
- ralentir l'horloge