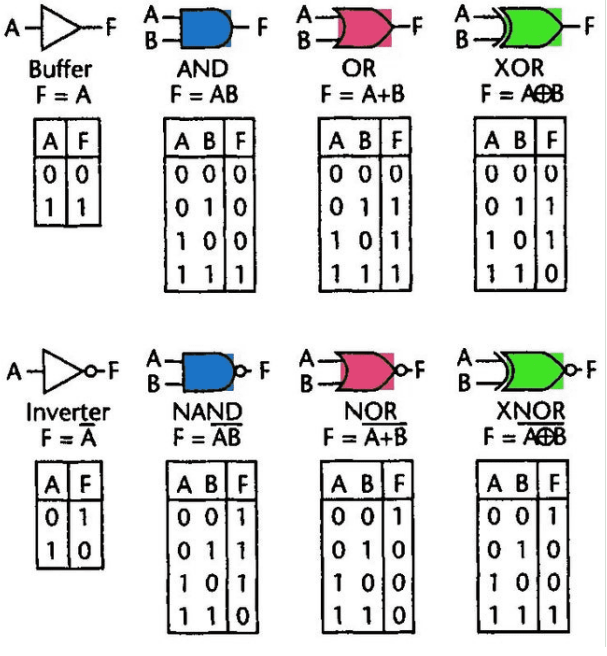
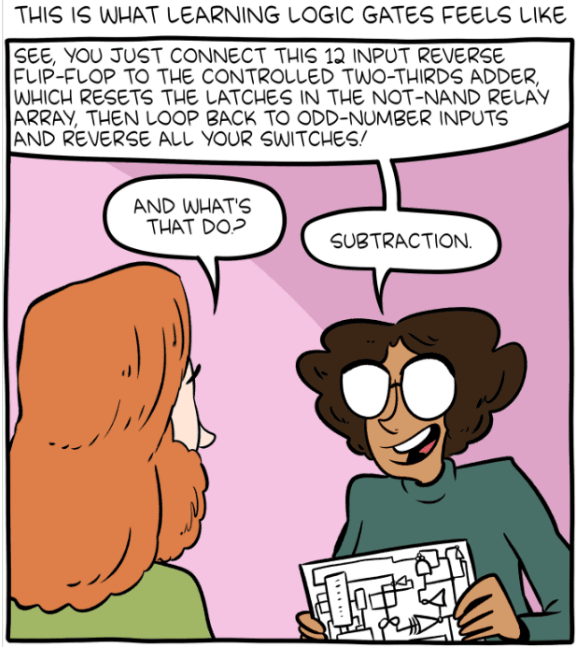
Faculté des Sciences & Techniques — Université de Limoges

Master 2nde année





FPGA et «*soft core*» RiscV — P-F. Bonnefoi

Version du 2 novembre 2025

Objectif et Moyens

But du module d'enseignement

Réalisation de **circuits électroniques** contrôlés par

- ▷ **circuits logiques** : maitriser la construction de circuit logique séquentiel ;
- ▷ **intégrer un circuit digital** dans un SoC ;
- ▷ **utiliser** ce circuit en programmation «*bare metal*» en C.

Moyens pratiques

Utilisation de la suite «*Open Source*» logicielle pour :

- ▷ découvrir la **programmation de circuit logique** avec Verilog ;
- ▷ utiliser un **simulateur de circuit** pour vérifier le comportement de son circuit ;
- ▷ utiliser un **outil de synthèse** pour réaliser le circuit **physiquement** à l'intérieur du FPGA cible ;
- ▷ programmer le FPGA et interagir avec le circuit défini :
  - ◇ **carte de développement** Black ICE II avec un FPGA Lattice iCE 40 hx8k :
  - ◇ port série, LEDs, boutons, broches d'E/S, mémoire ;

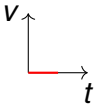
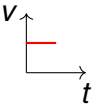
Utilisation du PicoSoc :

- ▷ **processeur RiscV** PicoRV32 ;
- ▷ circuit de chargement de *firmware* par **port série** ;
- ▷ création d'un circuit digitale réalisant un **LFSR** et **intégration** dans le SoC ;
- ▷ programmation d'un *firmware* en C utilisant ce LFSR et un mécanisme de «*timer*».

Mais un bit, c’est quoi au juste ?

Qu'est-ce qu'un bit, «*binary digit*» ?

Un **bit** représente un système à **deux états** possibles :

- «*allumé*»,  ou «*éteint*»,  ;
- «*allumé*», (I) ou «*éteint*», (O), d'où le symbole présent sur les interrupteurs poussoir :  ou 
- «*Vrai*» ou «*Faux*» ;
- un **voltage** «*bas*»  ou un voltage «*haut*»  (où «*v*» est le voltage et «*t*» le temps) ;
- une **magnétisation** de sens nord-sud N|S ou de sens sud-nord S|N sur un support magnétique ;
- la **valeur** «*1*» ou la valeur «*0*».

Qu'est-ce qu'un bit de mémoire dans un ordinateur ?

«*Un bit est juste un emplacement de stockage d'électricité :*

- ▷ *s'il n'y a pas de charge électrique alors le bit est 0 ;*
- ▷ *s'il y a une charge électrique*  *alors le bit est 1*

La seule chose que l'ordinateur peut mémoriser est si le bit est à 1 ou 0.»

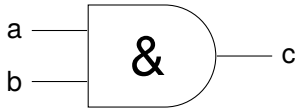
Chaque bit de mémoire correspond à une case dans laquelle on peut stocker un bit de données, soit la valeur 1, soit la valeur 0.

Et un circuit logique, c'est quoi ?

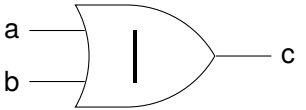
des opérateurs logiques...

et des composants électroniques !

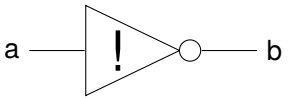
Le «*et*» : $c = a \,\&\, b$ ou $c = a \wedge b$



Le «*ou*» :$c = a|b$ ou $c = a \vee b$

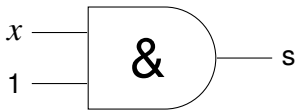


Le «*non*» : $b = !a$ ou $b = \neg a$

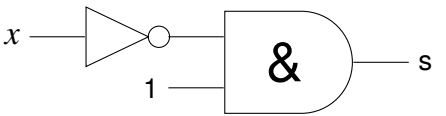


Créer des opérations

L'opération $x = 1$



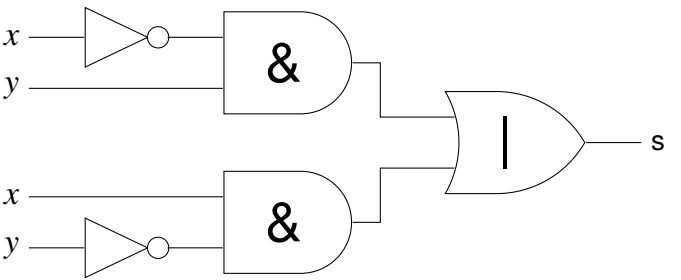
L'opération $x = 0$



$\Longrightarrow$ **L'opérateur «*xor*»**

Table de vérité du *xor*

a	b	$\oplus$
0	0	0
0	1	1
1	0	1
1	1	0

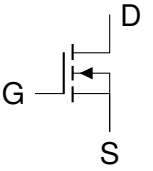


On constate que le *xor* est vrai si $a \neq b$

Et en électronique ?

Le transistor

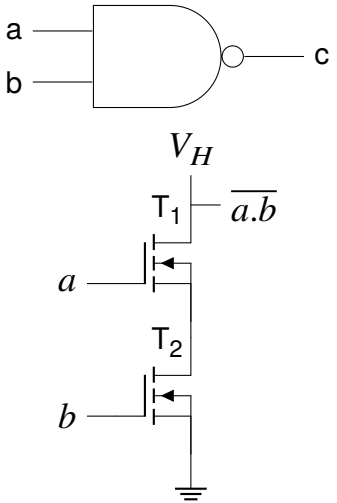
Le **transistor** agit comme un **interrupteur** : le courant peut circuler de la «*Source*» vers «*le Drain*» uniquement si une tension est présente sur la «*Gate*» :



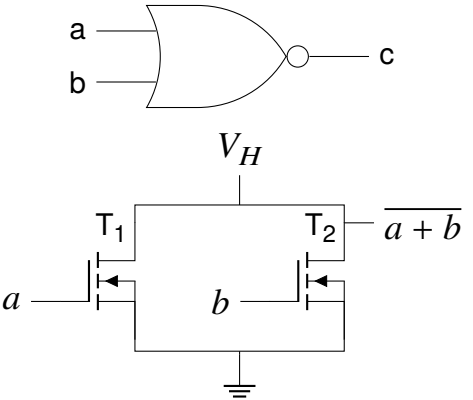
Simuler les portes logiques ?

Il est «*plus simple*» de construire des portes logiques **négatives** :

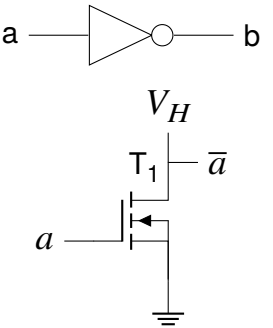
Le «*non-et*» ou NAND :



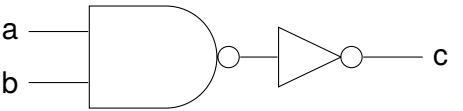
Le «*non-ou*» ou NOR :



Le «*non*» :



Et si on voulait une porte logique «*positive*», comme un «*OU*» ?

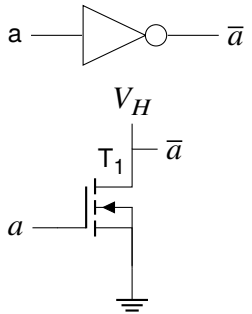


Et en vrai ?

Passer d'un **valeur logique** à un autre revient à **changer le voltage** :

- ▷ $0v$ pour le «*0*» logique $\Longrightarrow$ ce qui est relié directement au «*ground*» ;
- ▷ V_H pour le «*1*» logique $\Longrightarrow$ ce qui est relié directement à V_H ;

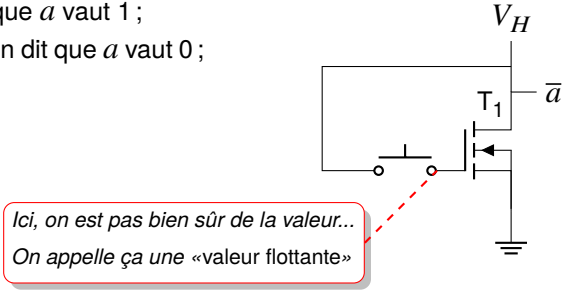
Exemple sur le «*non*» :



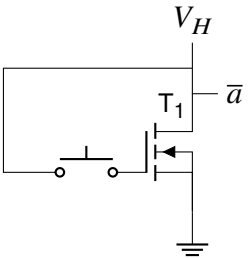
Si l'entrée a est connectée à

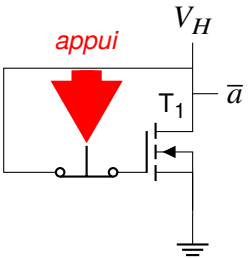
- ▷ V_H alors on dit que a vaut 1 ;
- ▷ «*ground*» alors on dit que a vaut 0 ;

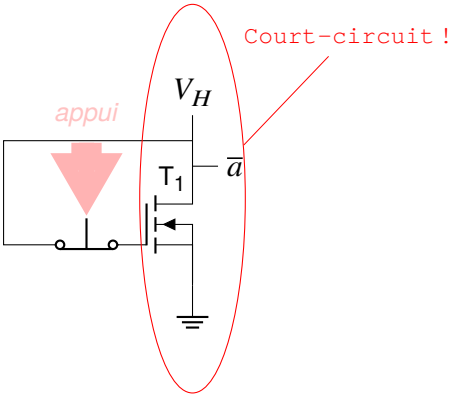
Ce qui donne :



Et électriquement, ça marche ?







Électricité : Rappels

Le **courant électrique** se comporte comme un liquide dont le **flot** circule du «*plus*» vers le «*moins*» :

- le **voltage**, exprimé en volts, qui exprime la «*pression*» du flot ;
- la **résistance**, exprimée en ohms, qui mesure la résistance opposée à ce flot ;
  *On notera également qu'une **chute de voltage** se produit à la sortie d'une résistance comme pour un liquide où une haute pression en entrée d'un obstacle donne une plus faible pression en sortie*
- l'**intensité**, exprimé en ampères, qui indique la quantité de liquide qui circule.
  En réalité, le nombre de charges électriques circulant dans le flot (électrons).

En général, c'est l'intensité du courant, son ampérage, qui entraîne des problèmes dans un circuit.

Loi d'Ohm $U = R * I$, ou «*volts et résistance crée l'ampérage*»

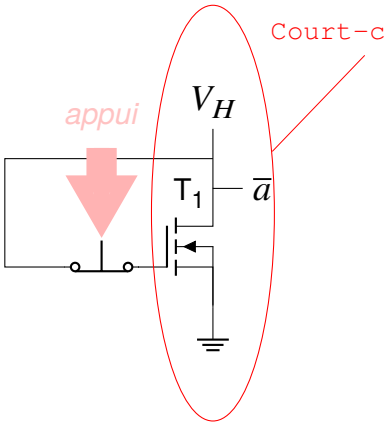
$\boxed{\frac{V}{\Omega} = A}$ ce qui se traduit pour un voltage constant par :
$\Longrightarrow$ l'ampérage ↗ quand la résistance ↘
$\Longrightarrow$ l'ampérage ↘ quand la résistance ↗

Ce qui permet de distinger **3 situations de panne** dans un circuit :

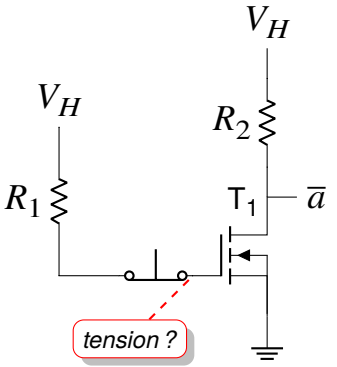
- ▷ le **circuit ouvert** où il n'y a pas de circulation $\Longrightarrow$ la **résistance** est infinie et le flot est nul ;
- ▷ le **court-circuit** où le flot va directement vers le «*ground*» ce qui entraîne trop de flot $\Longrightarrow$ la **résistance** est très proche de zéro et l'ampérage tend vers l'infini $\Longrightarrow$ les composants brûlent !
  Ils libèrent la fumée magique qui les faisait fonctionner...
- ▷ **pas assez de flot de courant** pour que le circuit fonctionne correctement $\Longrightarrow$ la **résistance** est trop élevée.

On remarque que chaque panne est liée à un changement de résistance...

Et finalement ?



Il faut ajouter des résistances pour limiter l'intensité du courant, c-à-d son ampérage.



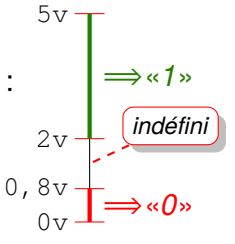
On évite deux court-circuits possibles avec R_1 et R_2.

Par contre, on ne connait pas la tension à l'entrée de la «gate» du transistor...

Comment distinguer un «*0*» et un «*1*» ?

Pour des circuits électroniques «*standards*» (TTL) :

- ▷ de $5v$ à $2v \Longrightarrow$ «*1*» ;
- ▷ de $0,8v$ à $0v \Longrightarrow$ «*0*» ;
- ▷ de $0,9v$ à $1,9v \Longrightarrow$ «*indéfini*» ou «*flottant*» ;



Autre usage de ces résistances

Elles garantissent une tension :

- ▷ **Pull up** resistor : garantie une tension proche de V_H, c-à-d un «*1*» logique, *ici, R_1 et R_2* ;
- ▷ **Pull down** resistor : garantie une tension proche de 0, c-à-d un «*0*» logique, *ici, il n'y en a pas !*

Et finalement ?

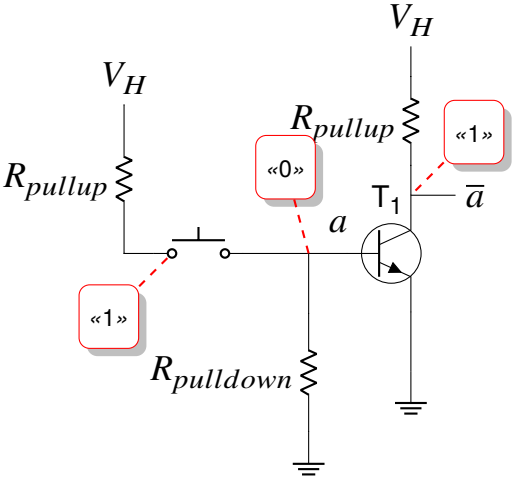
On rajoute des résistances de «*pull up*» pour :

- ▷ **forcer une tension** interprétable comme un «*1*» logique ;
- ▷ **éviter un court circuit** en cas d'utilisation d'interrupteur pour ouvrir/fermet le circuit ;

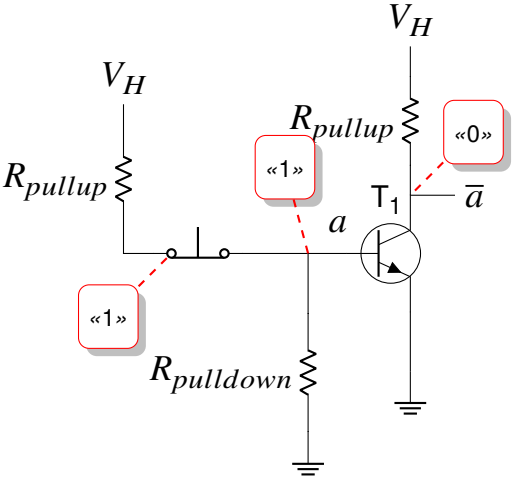
On rajoute des résistances de «*pull down*» pour :

- ▷ **forcer une tension** interprétable comme un «*0*» logique ;
- ▷ **éviter un court circuit** en cas d'utilisation d'interrupteur pour ouvrir/fermet le circuit ;

D'où le circuit final :

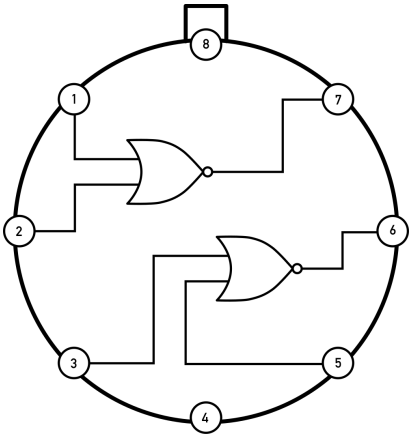


Si $a = 0$ alors $\overline{a} = 1$



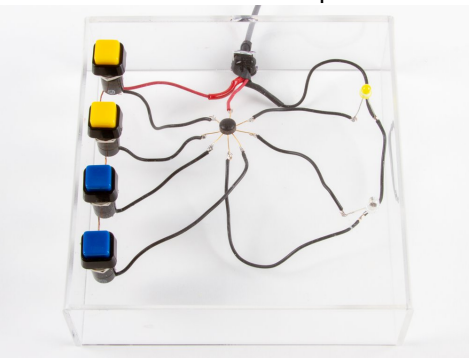
Si $a = 1$ alors $\overline{a} = 0$

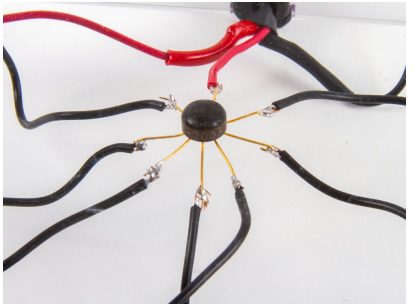
Et en réalité ? Exemple du composant µL914 de la société Fairchild



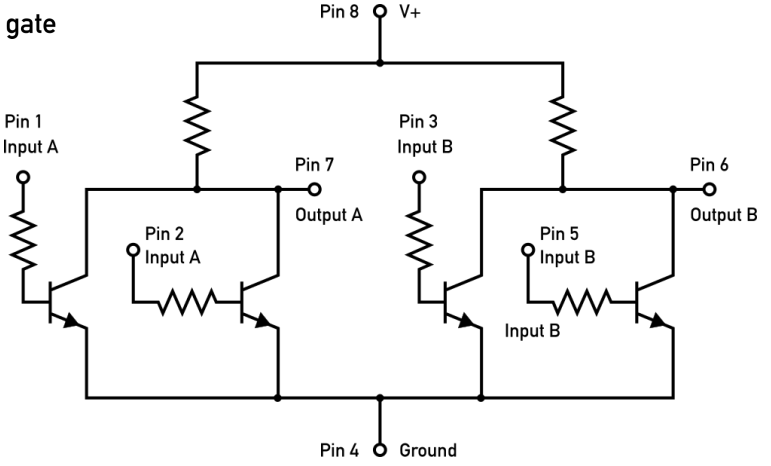
µL914
Dual 2-input NOR gate

Le composant est au centre, connecté à des boutons poussoirs.



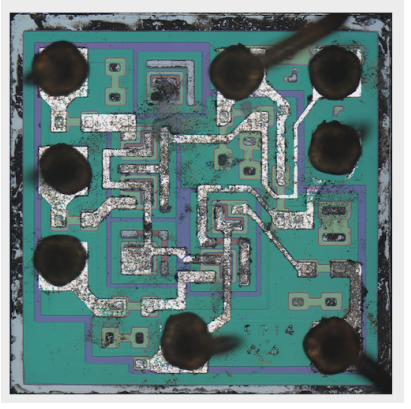


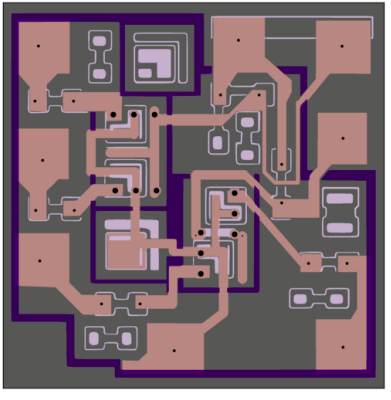
Agrandissement du composant.

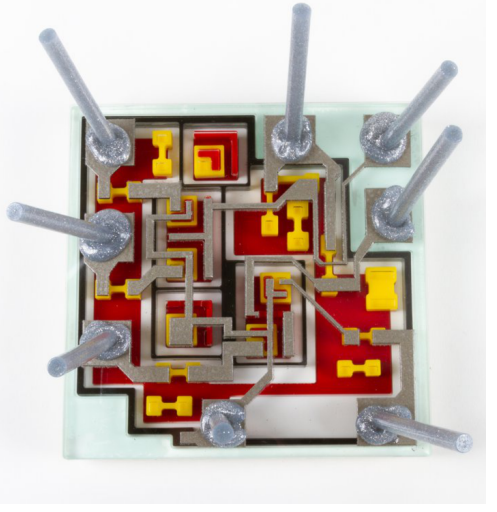


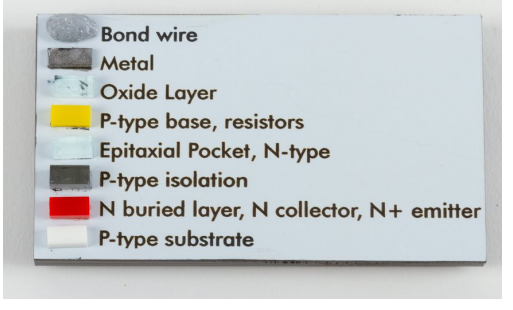
Et à l'intérieur ?

Le composant a été «*décapé*» : sa coque de protection a été enlevée par abrasion et utilisation d'acides :





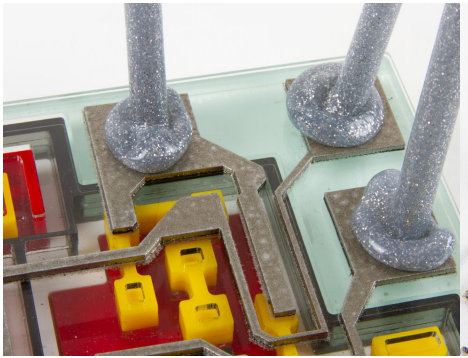




Le composant est constitué de différentes couches de matériaux différents, superposées les unes sur les autres.
On obtient chaque couche par dépôt de substrat ou par gravure (creusement d'une couche).

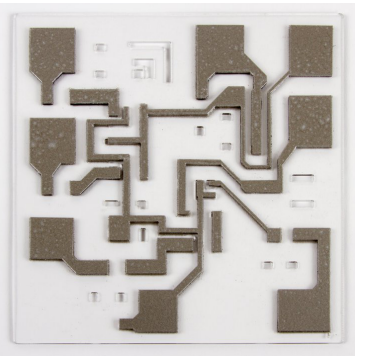
Et à l'intérieur ?

Fils de connexion vers l'extérieur :

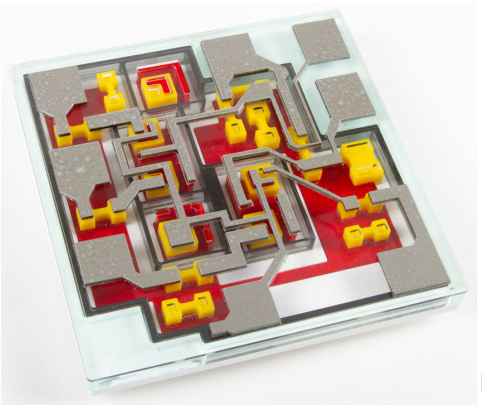


Connexion sur la couche conductrice :

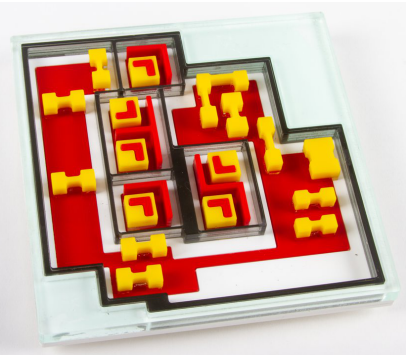
Couce conductrice :



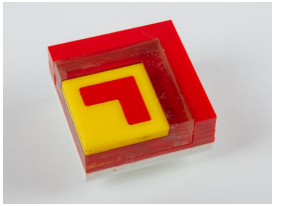
Toutes les couches :



Sans la couche conductrice :



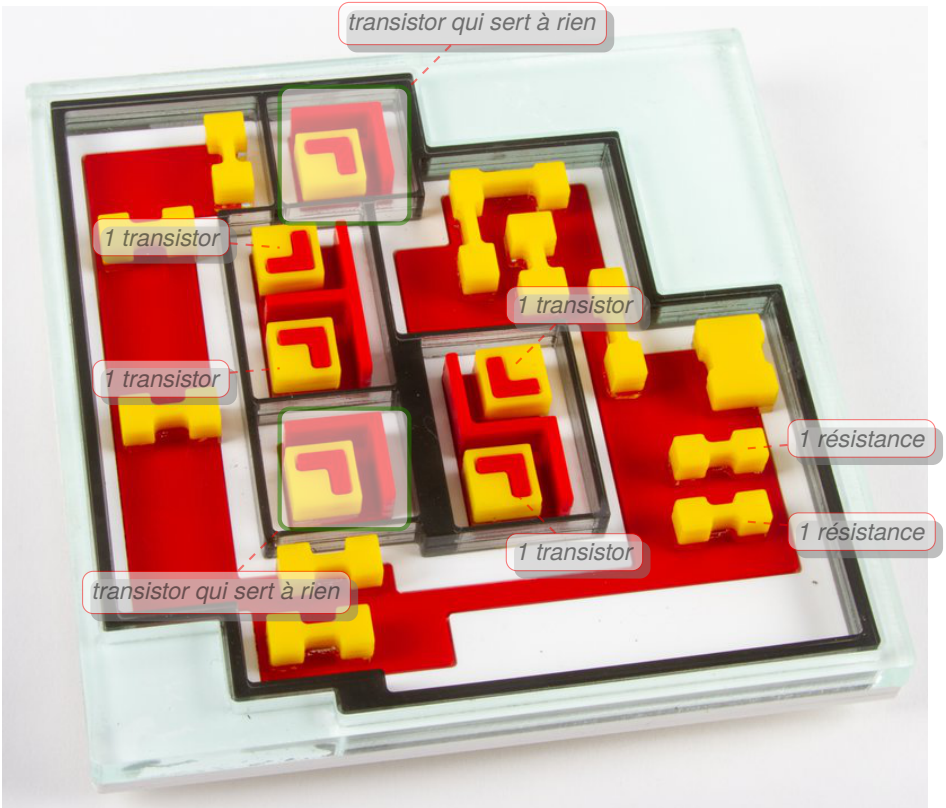
Les Transistors :



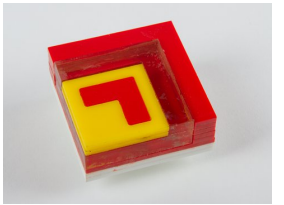
Les Résistances :

Deux transistors ne servent à rien.

Si on analyse...



Les Transistors :



Les Résistances :

On retrouve chaque transistor et résistance du circuit. Certains transistors ne servent à rien : ils ont été gravés/déposé mais ne sont pas connectés par la couche conductrice.

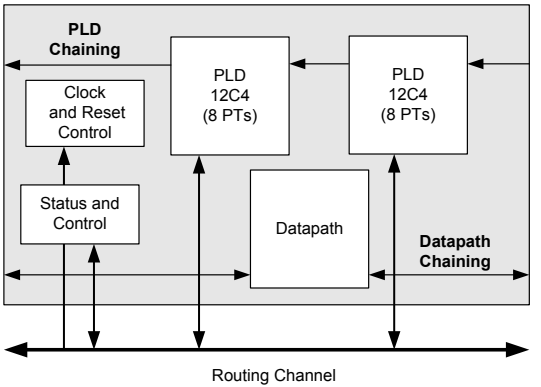
Mais des circuits logique reconfigurables

Ca existe pas déjà avec les CPLDs ?

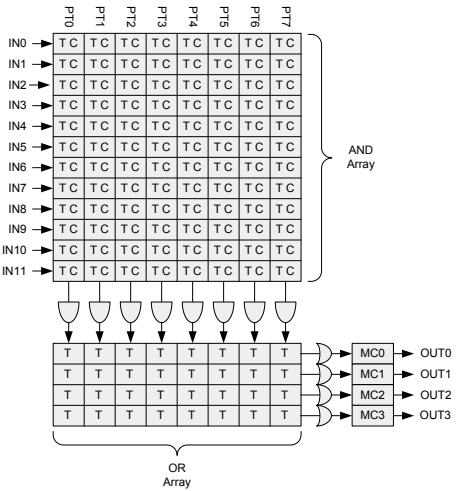
D'après la documentation

PSoC implements programmable logic through an array of small, fast, low-power digital blocks called Universal Digital Blocks (UDBs). PSoC devices have as many as 24 UDBs. A UDB consists of two small programmable logic devices (PLDs), a datapath module, and status and control logic.

UDB



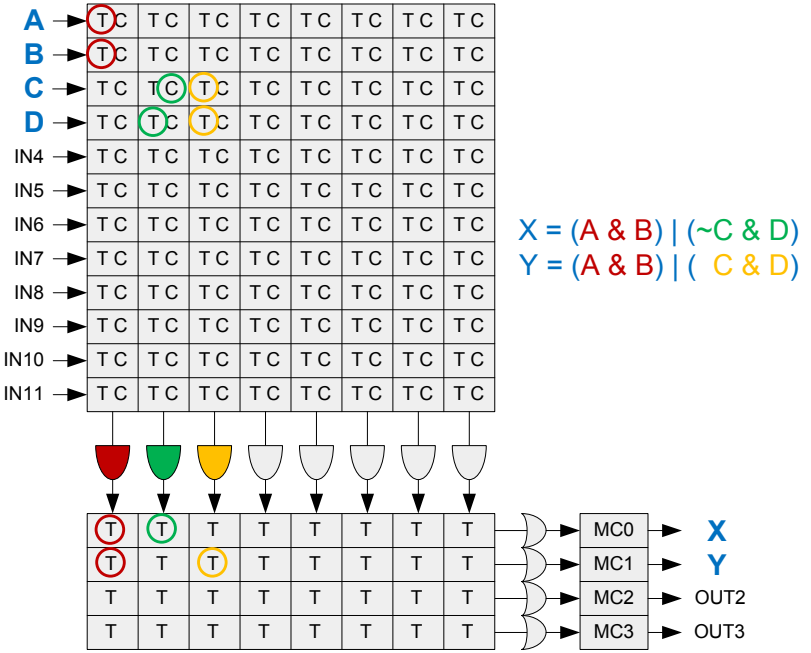
PLD



D'après la documentation

PSoC PLDs, like most standard PLDs, consist of an AND array followed by an OR array, both of which are programmable. There are **12 inputs** which feed across **eight product terms** (PTs) in the AND array. In each PT, either the true (T) or complement (C) of the input can be selected. The **outputs of the PTs** are inputs into the **OR array**.
The outputs of the OR gates are fed to **macrocells** (MC). Macrocells are **flip-flops** with additional combinatorial logic.

Exemple de circuit simulant une fonction logique

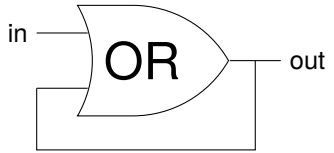


D'après la documentation

PSoC PLDs, like most standard PLDs, consist of an AND array followed by an OR array, both of which are programmable. There are **12 inputs** which feed across **eight product terms** (PTs) in the AND array. In each PT, either the true (T) or complement (C) of the input can be selected. The **outputs of the PTs** are inputs into the **OR array**.
The outputs of the OR gates are fed to **macrocells** (MC). Macrocells are **flip-flops** with additional combinatorial logic.

Mais comment les bits sont mémorisés ?

Que se passe-t-il si on branche la sortie en entrée d'une porte logique ?

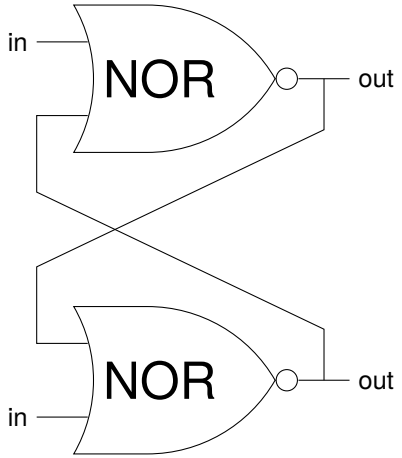


a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

Ici, que se passe-t-il ?

- ▷ *au début on peut imaginer que :*
  - ◇ *l'entrée* «in» *est à zéro ;*
  - ◇ *la sortie* «out» *est à zéro ;*
- ▷ *Mais dès que l'entrée* «in» *passe à un alors la sortie reste bloquée à un !*

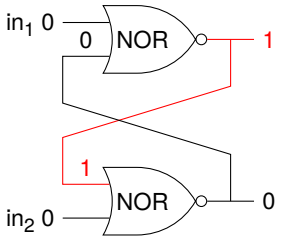
Comment faire pour «*l'éteindre*» ?



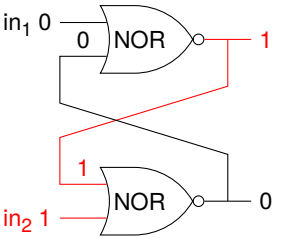
a	b	NOR
0	0	1
0	1	0
1	0	0
1	1	0

À l'allumage du circuit que se passe-t-il ?

❶ À l'allumage, les entrées sont à zéro :
⟹ on obtient :

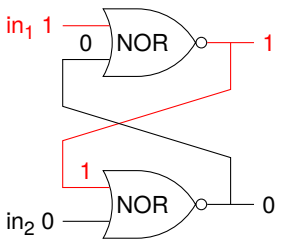


❷ Si on mets l'entrée «in_2» à un :
⟹ l'état du circuit ne change pas :

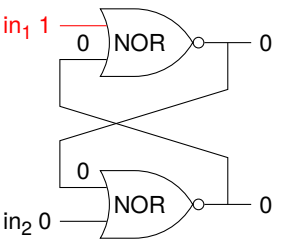


a	b	NOR
0	0	1
0	1	0
1	0	0
1	1	0

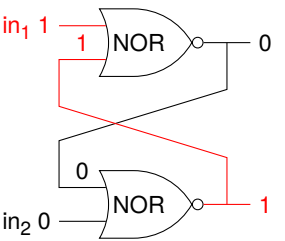
❸ Si on mets l'entrée «in_1 *à un*» :
⟹ on obtient :



❹ Si on mets l'entrée «in_1» à un :
⟹ l'état du circuit change vers :

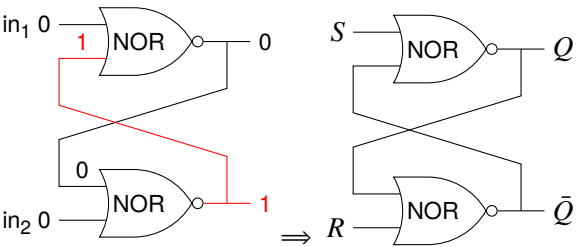


❺ Si on mets l'entrée «in_1» à un :
⟹ l'état du circuit se stabilise sur :



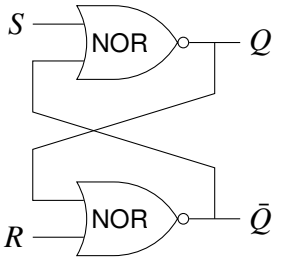
❻ Si on remets l'entrée «in_1» à zéro :
⟹ le circuit ne change pas ;
⟹ on est dans l'image inversée de l'état précédent où l'état ne changeait plus...

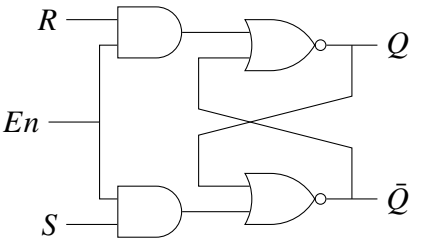
⟹ **On vient de construire une «*S-R Latch*», une «*set/reset*» latch ! : un bouton «*set*» et l'autre «*reset*»**
⟹ **On vient de mémoriser un état !**



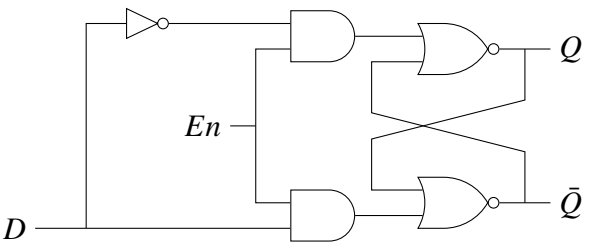
Améliorer le circuit ? Ajouter une entrée «*enable*»

On ajoute le «*enable*» qui permet d'activer ou non la SR-latch :





Si on va plus loin : mémorisation d'un bit «*à la demande*»



À chaque fois que :

- ▷ l'entrée «*enable*» est active : la sortie «Q» reproduit la valeur de l'entrée «D» ;
- ▷ l'entrée «*enable*» est inactive : la sortie «Q» conserve sa valeur quelque soit la valeur de D ;

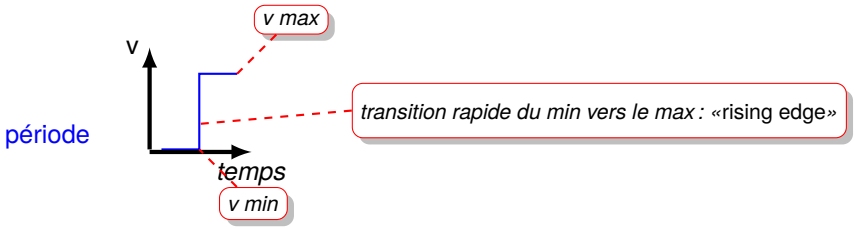
On «*copie*» la valeur de D en activant le «*enable*».

$\Longrightarrow$**On peut mémoriser un bit d'information à la demande !**

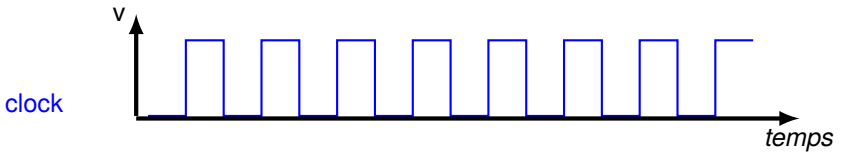
Qu'est-ce que l'horloge ?

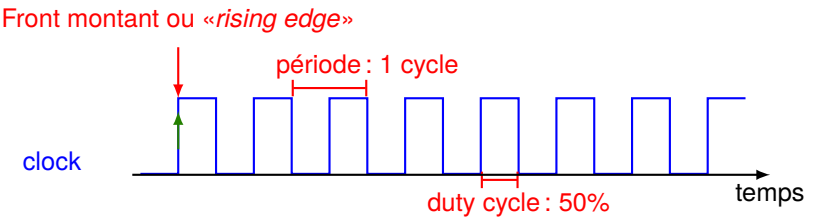
Un **signal** électrique :

- ▷ une variation de tension entre 0v et 3,3v ou 5v (en fonction des micro-contrôleurs par exemple) ;
- ▷ périodique : la période est une portion du signal qui se reproduit indéfiniement :



*où le temps passé avec le voltage maximal est égal au temps passé avec le voltage minimal (*duty cycle *de 50%).*





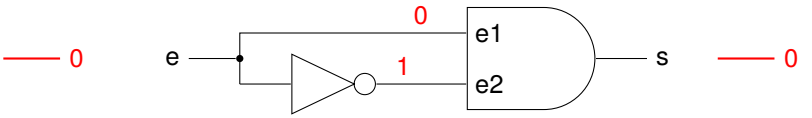
- ▷ *en résumé* :

⟹ servir de **référence globale** dans le circuit ;

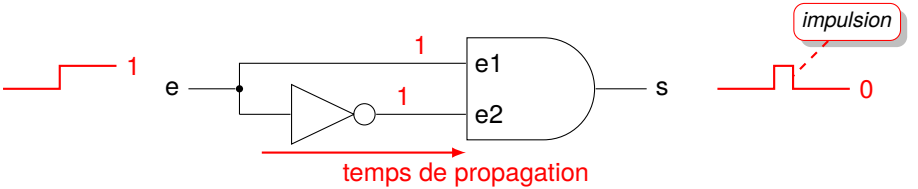
⟹ **synchroniser** les différentes parties de ce circuit.

En général, on utilise le front montant ou «rising edge» *pour effectuer cette* ***synchronisation***.

Introduction d'un délai : détection du front montant, «*rising edge*», de l'horloge

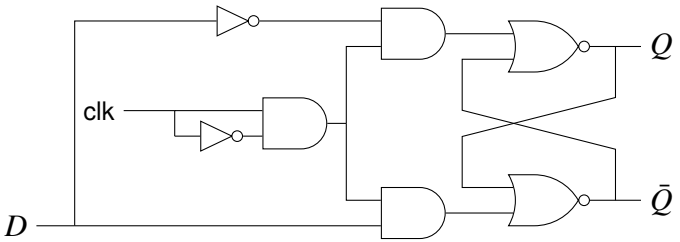


*La traversée de la porte logique «*NOT*» introduit un délai qui permet au «*ET*» de sortir un 1 pendant un court instant.*



*Pendant un court instant, les entrées «*e1*» et «*e2*» sont vraies ⟹ une impulsion en sortie.*

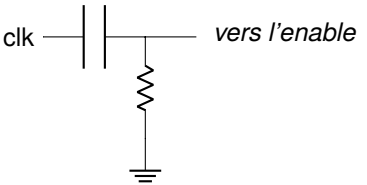
Combinaison avec le circuit de mémorisation à la demande : la «*D flipflop*»



Si on utilise l'horloge pour l'entrée «*enable*» on obtient une **mémoire synchronisée** sur l'horloge !

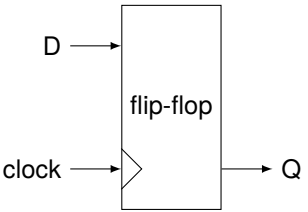
⟹ la «*D-flipflop*»

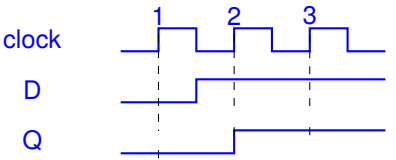
La valeur D est mémorisée/accessible au moment où l'horloge change ⟹ on peut synchroniser différents circuits entre eux.



On peut également utiliser un condensateur pour générer l'impulsion :

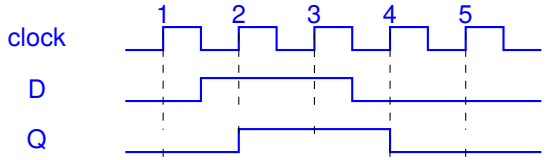
La D flip-flop





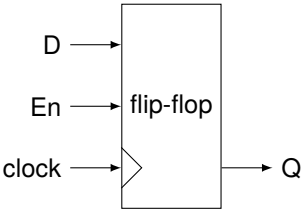
▷ le signal «*D*» devient actif, il vaut «*1*», entre les cycles d'horloges 1 et 2, c-à-d avant le front montant de 2
⟹il est ignoré par la flip-flop ⟹le signal «*Q*» est inactif, il vaut «*0*» ;

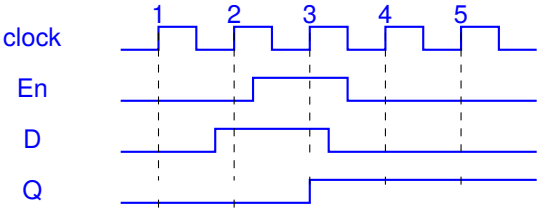
▷ lors du front montant 2, la flip-flop enregistre, «*register*», le changement du signal «*D*»
⟹«*Q*» devient actif.



De même, lorsque le signal «*D*» passe à 0, la flip-flop ne l'enregistrera qu'au prochain front montant en 4.

On ajoute une entrée «*Enable*» à la D flip-flop :





▷ lorsque le signal «*En*» n'est pas actif⟹pas de modification de Q. C'est seulent en 3 que Q enregistre D.

Ce sont ces D-Flip-flop qui sont intégrées dans les FPGAs : elles enregistrent D aussi longtemps que voulu.

Et les circuits logique dans tout ça ?

LUT : «*Look Up Table*»

Une fonction binaire de n entrées vers un bit en sortie peut être simulée par une LUT :

Fonction binaire

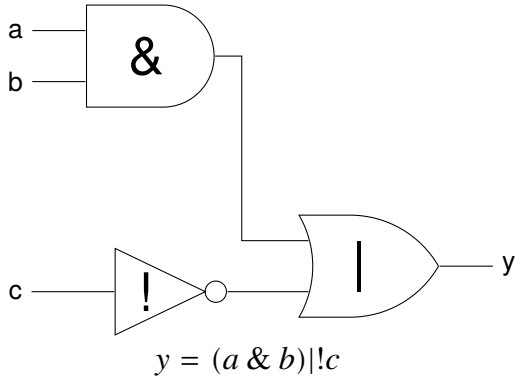
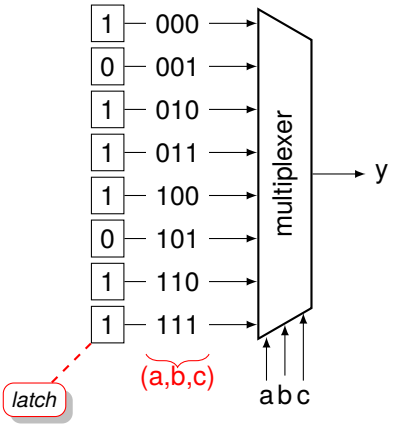


Table de vérité

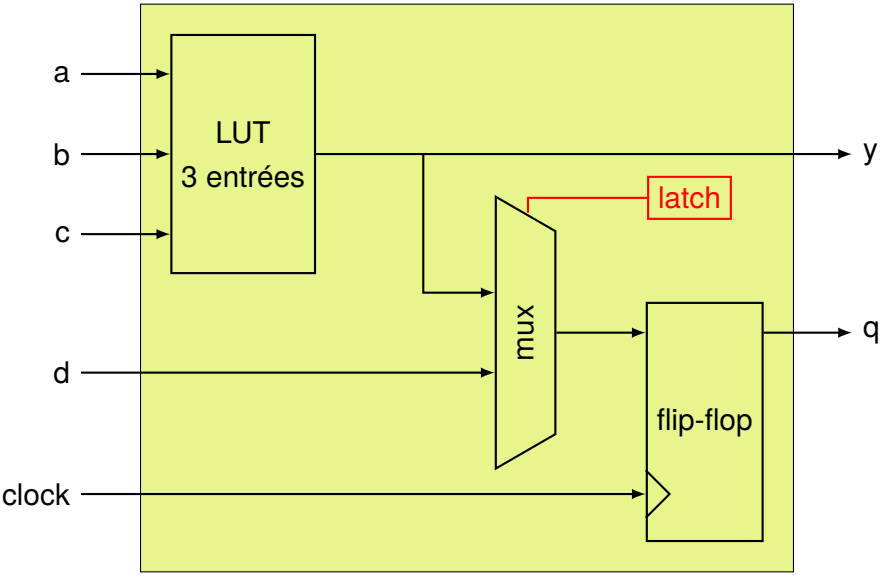
a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

LUT



- ▷ pour chaque combinaison des variables d'entrées (a, b, c), une valeur de sortie y est **mémorisée** :
  - ◇ la fonction binaire **n'est pas implémentée** sous forme de portes logiques ;
  - ◇ le **temps de propagation** nécessaire à la traversée de chaque portee logique en série est **minimisé** ;
- ▷ la sélection de la valeur de sortie par rapport aux valeurs en entrée est faite par un **multiplexeur** :
  - ◇ suivant le nombre d'entrées du multiplexeur, il peut être nécessaire d'en combiner plusieurs si on a besoin de plus d'arguments ou de plus de bits en sortie $\Longrightarrow$ de la BRAM, «*block ram*» peut être utilisée en remplacement.
- ▷ la configuration de chaque valeur de sortie y dans le multiplexeur est :
  - ◇ définie dans le «*bitfile*» lors de la synthèse du design ;
  - ◇ mémorisée dans un *latch* lors de la programmation du FPGA.

Associer une LUT et une Flip-Flop



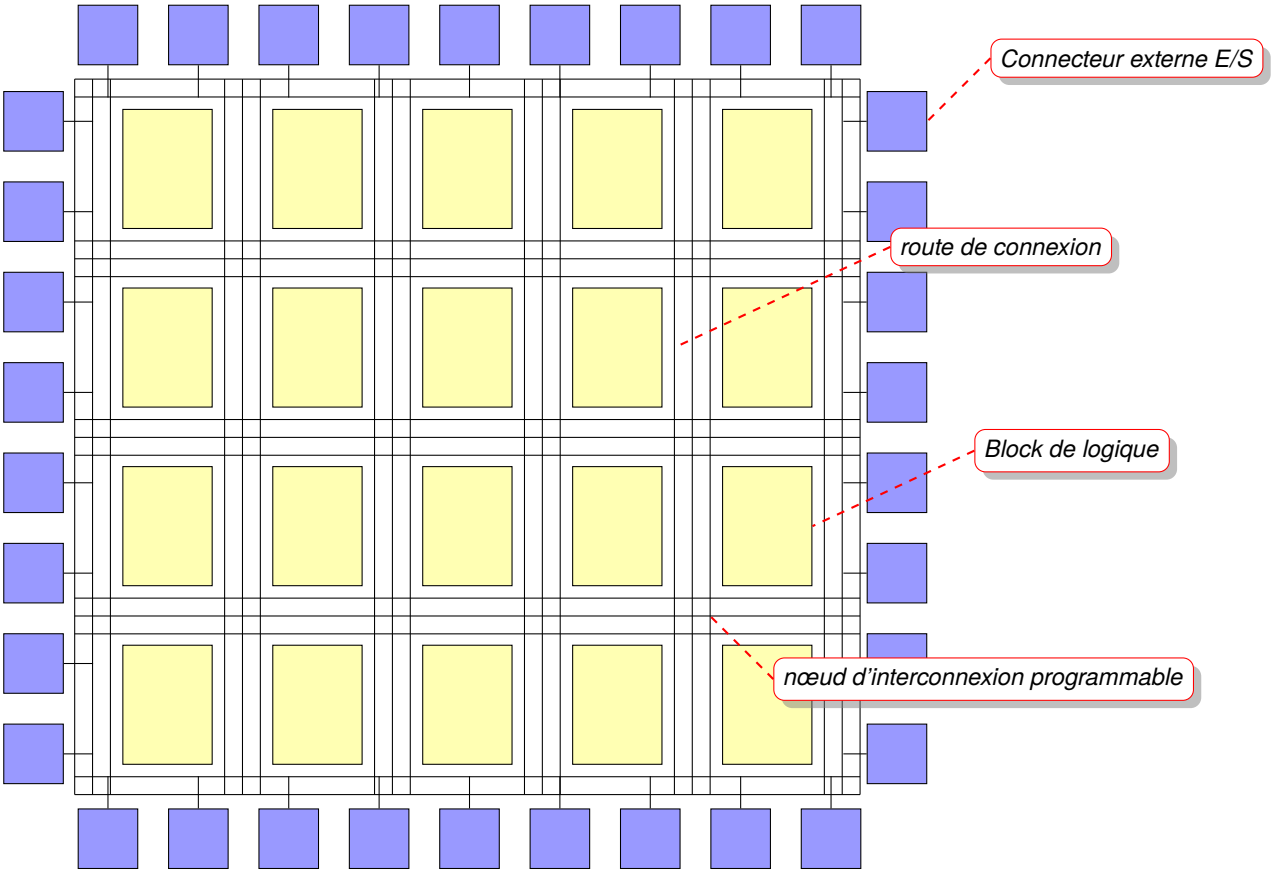
La sortie :

- ☐ «*y*» sert au circuit logique **séquentiel** ;
- ☐ «*q*» sert au circuit logique **combinatoire**.

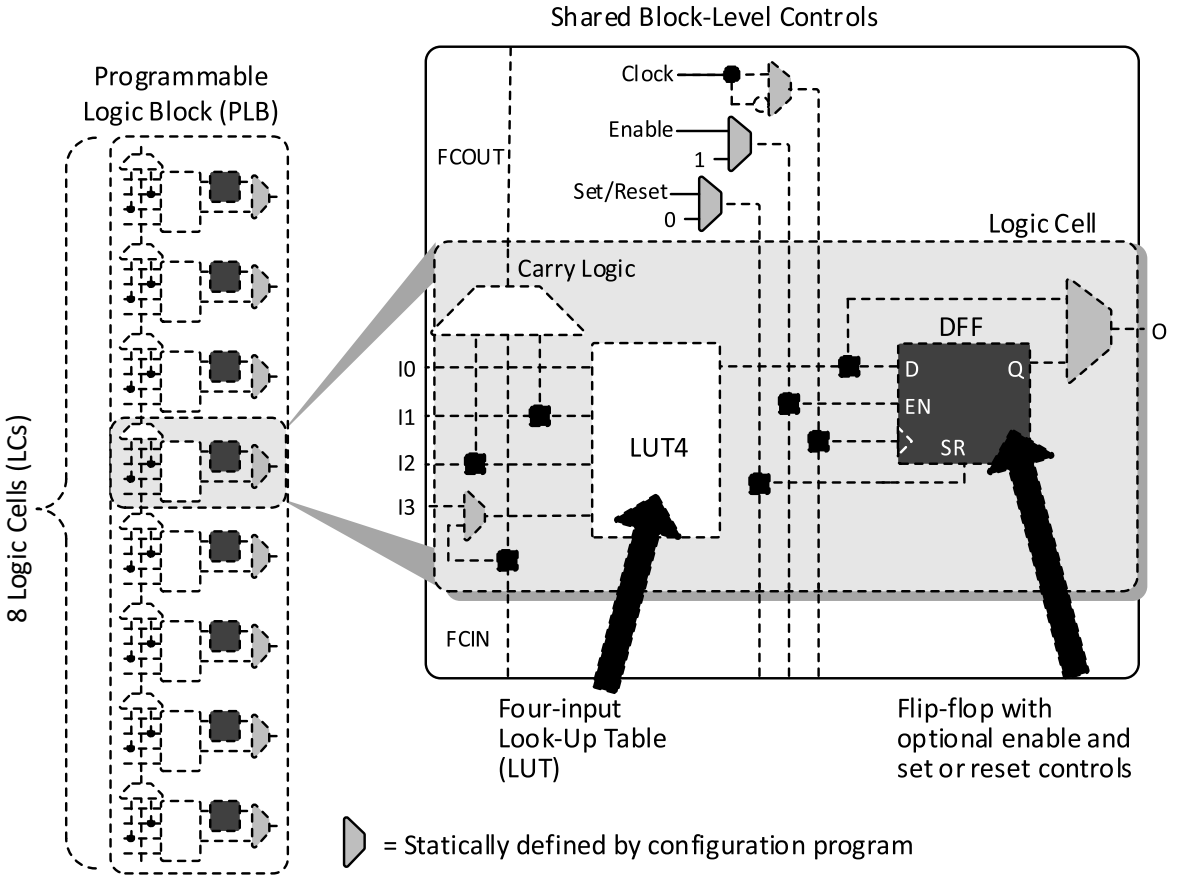
Le «*latch*» :

- ▷ sert à sélectionner la sortie de la LUT ou celle de la Flip-Flop ;
- ▷ est configuré dans le «*bitfile*» lors de la programmation du FPGA.

Schéma fonctionnel d'un FPGA : milliers de blocs logiques interconnectés



Et pour le Lattice iCE40 hx8k



Donc, programmer un FPGA :

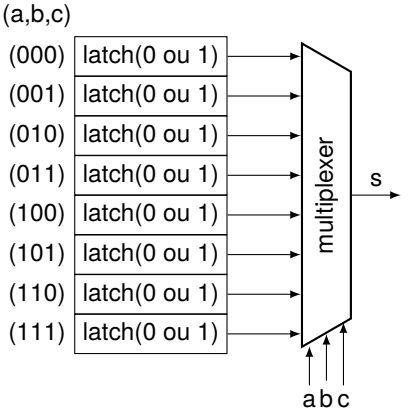
c’est choisir une interconnexion de blocs logiques ?

Sauver cette configuration dans un fichier «*bitstream*»

Et c’est tout ?

Configuration de la LUT dans le «*bit stream*»

- ▷ chaque combinaison de (a, b, c) est associée à un bit configuré par la valeur contenue dans une «*latch*» ;
- ▷ la configuration de toutes les «*latches*» d'un multiplexeur est faite dans le fichier de configuration du FGPA : le «*bit stream*» ;
- ▷ cette configuration est **permanente** jusqu'à la reconfiguration/reprogrammation du FPGA (elle ne peut pas être changée durant l'exploitation du design programmé dans le FPGA).



Des multiplexeurs peuvent être utilisés pour servir de ROM, «Read Only Memory»*, où la configuration d'un bit peut être consultée mais pas modifiée pendant l'utilisation du FPGA.*

En particulier, on peut ***grouper*** *plusieurs LUTs :*

- ▷ *pour stocker des* «données» *sur plusieurs bits : même sélection par (a, b, c) mais pour un bit à la fois : 8 muliplexeurs par exemple pour stocker un octet.*
- ▷ *en* ***cascade*** *pour augmenter le nombre de sélecteur :*
  - ◇ *une LUT à 3 entrées utilisée sur une seule entrée pour la sélection de deux autres LUTs à 3 entrées : on passe à une LUT à 4 entrées.*

⟹***Le nombre d'entrées pour une LUT doit être choisie de manière optimale.***

⟹***Dépend du choix du constructeur :*** *pour le iCE40 de Lattice, ce sont des LUTs de 4 entrées qui ont été choisies.*

Mais...

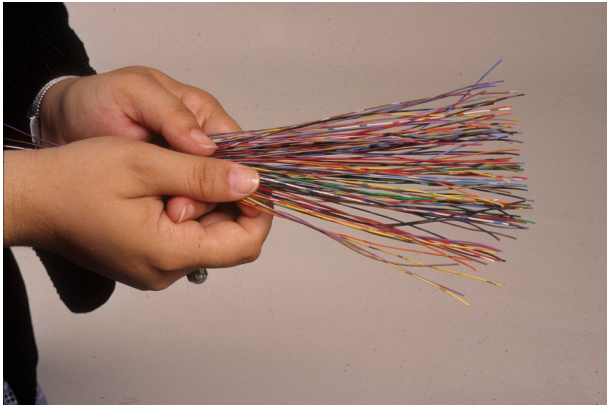
il y aurait pas des problèmes tout de même ?

Les processus de placement et routage doivent garantir que :

▷ le circuit est **viable** ;

⟹ les composants sont exploités à une vitesse qu'ils peuvent supporter sans erreur.

Une **évaluation** est faite par l'outil de «*pnr*», pour déterminer le temps maximal de propagation, c-à-d le temps pour un signal d'arriver de la source à sa destination.



Un signal voyage en une *nanoseconde*, 10^{-9} ou $1GHz$, sur une distance de 30cm environ.

Photo ci-contre de l'illustration d'une «nanoseconde» *pat Grace Hopper.*

▷ les **routes mise en place** peuvent être assez longues ⟹ le **temps de propagation** à travers ces routes **augmente** ;

▷ **chaque élément logique traversé augmente le temps de propagation** :

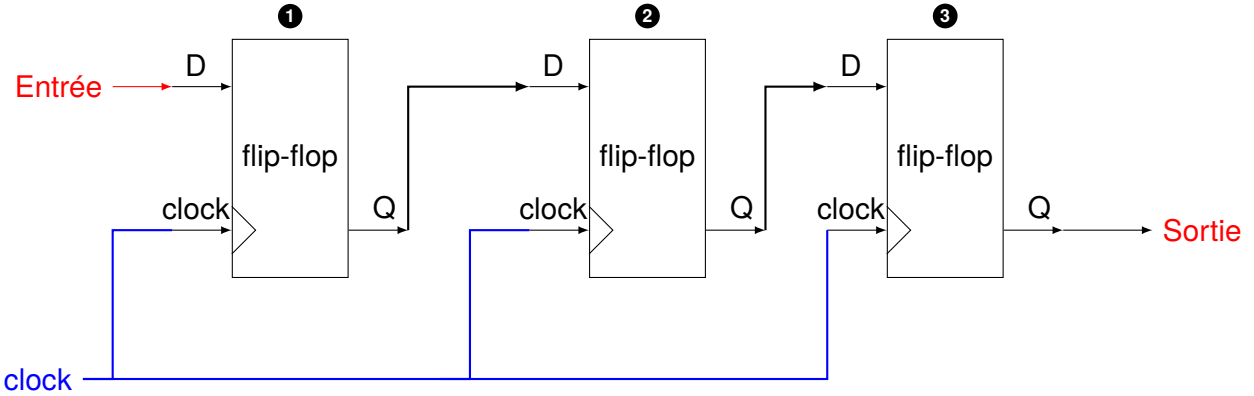
- ◇ LUT ;
- ◇ flip-flop ;
- ◇ BRAM, «*Bloc RAM*»

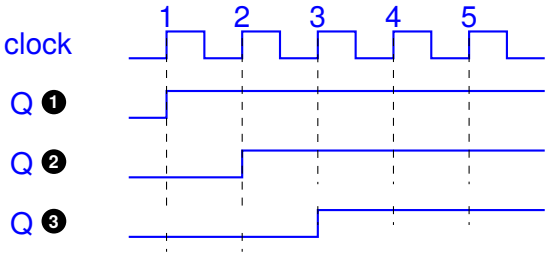
...

Les processus de placement et routage, «*pnr*», doivent tenir de ces **contraintes** de propagation :

⟹ ils recherchent le **meilleur placement possible** qui **minimise le temps de propagation** lié au routage !

Exemple de circuit : Registre à décalage ou «*Shift register*»



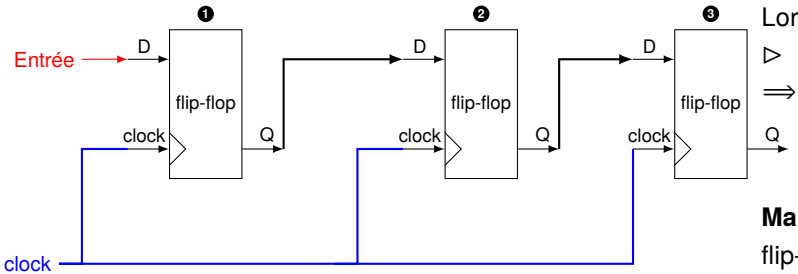


À chaque cycle d'horloge, le signal d'entrée se propage dans le circuit :

- ▷ *au cycle 1, la flip-flop ❶ enregistre le signal en entrée puis le positionne en sortie ;*
- ▷ *au cycle 2, c'est au tour de la flip-flop ❷ ;*
- ▷ *au cycle 3, c'est enfin le tour de la flip-flop ❸ ;*

⟹Il faut **3 cycles d'horloge** pour propager le signal à travers le circuit.

Mais comment ça fonctionne ce «décalage» ?

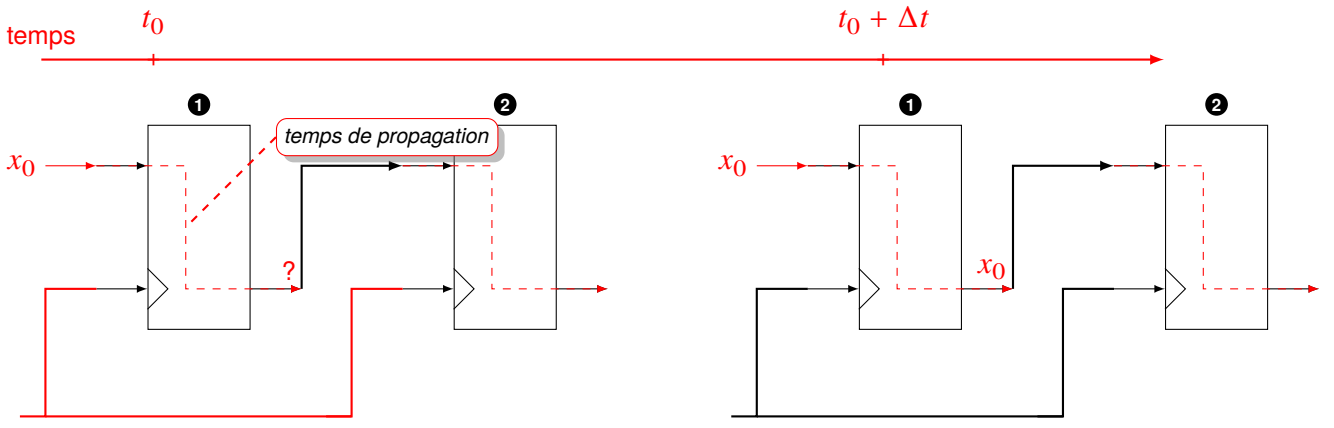


Lorsque le «front montant» de l'horloge se produit :

▷ chaque flip-flop reçoit **simultanément** ce signal ;

⟹ chaque flip-flop enregistre la valeur D (entrée) et la fournit en Q (sortie)

Mais, il y a le **temps de propagation** au travers de la flip-flop !



□ à t_0:

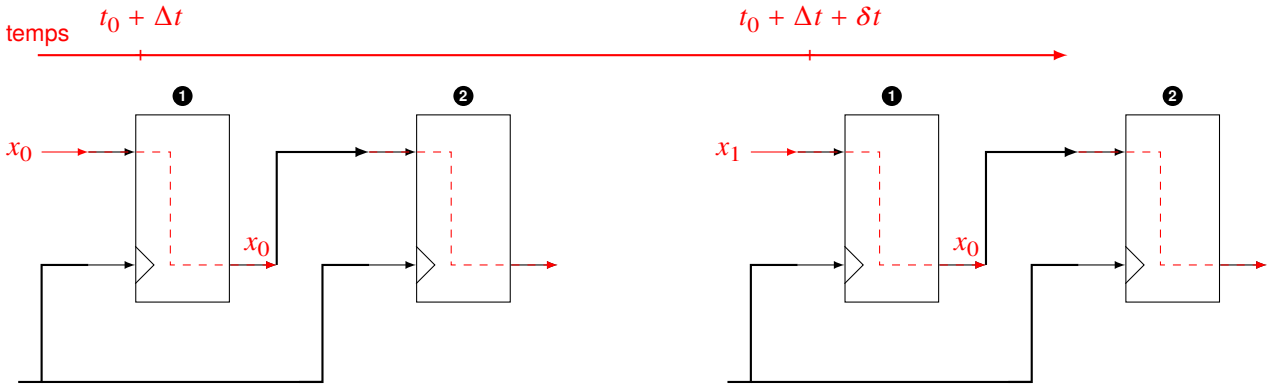
⟹ impulsion de l'horloge ;

▷ x_0 est présent à l'entrée de la flip-flop ❶ ;

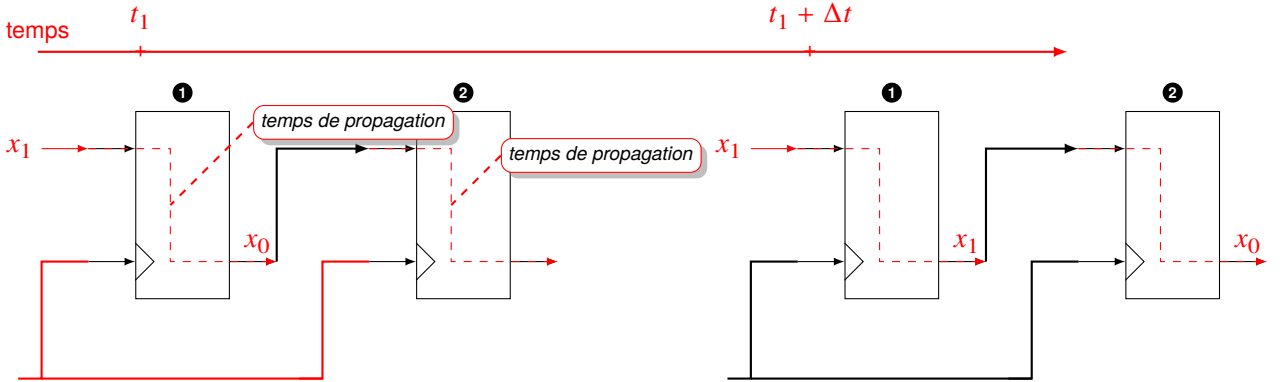
▷ la flip-flop ❶ enregistre x_0 ;

□ à $t_0 + \Delta t$, x_0 est propagé à la sortie de ❶ ;

Mais comment ça fonctionne ce «décalage» ? – la suite



- à $t_0 + \Delta t$, x_0 est à la sortie de ❶ et en entrée de ❷ ;
- à $t_0 + \Delta t + \delta t$, x_1 est présent à l'entrée de ❶ ;



- à t_1 :
  ⟹ impulsion de l'horloge ;
  - ▷ x_1 entre dans ❶ et x_0 dans ❷ ;
- à $t_1 + \Delta t$, x_1 est à la sortie de ❶ et x_0 à celle de ❷ ;

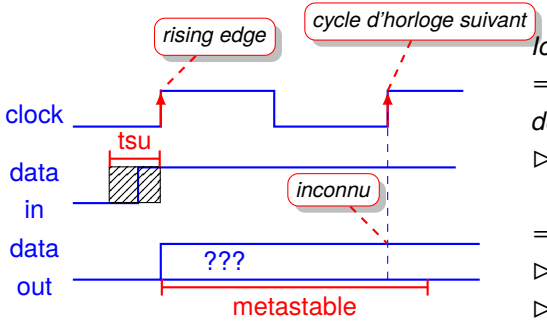
Ok, mais au niveau électronique
il y aurait pas aussi des soucis ?

Contraintes liées aux flip-flops

- «*tsu*», «*Time to Set Up*» : la durée pendant laquelle l'entrée **doit rester stable avant** le front montant de l'horloge pour que la flip-flop puisse **enregistrer la donnée en entrée** et la fournir sur sa ligne de sortie ;
- «*th*», «*Time to Hold*» : la durée pendant laquelle l'entrée doit **rester stable après** le front montant de l'horloge pour que la flip-flop puisse **conserver correctement** sa valeur en sortie jusqu'au **prochain cycle d'horloge**.

ATTENTION : la flip flop peut être dans un état inconnu au prochain cycle d'horloge

Si l'entrée change pendant le «*tsu*» ou le «*th*» alors la flip-flop peut se retrouver dans un état «*metastable*» :



Ici, l'entrée change de 0 vers 1 pendant le «time to setup» *de la flip-flop.*
⟹*le temps que la sortie de la flip-flop se* ***stabilise*** *sur la valeur de l'entrée devient* ***indéterministe*** *:*

- *il est* ***probabiliste****, il peut* ***varier*** *et surtout* ***dépasser le temps attendu****, c-à-d pour être stable au cycle d'horloge suivant.*

⟹***au prochain cycle d'horloge*** *:*

- *la valeur en sortie de la flip-flop est* ***imprédictible*** *!*
- *il peut ou non être en accord avec la valeur en entrée.*

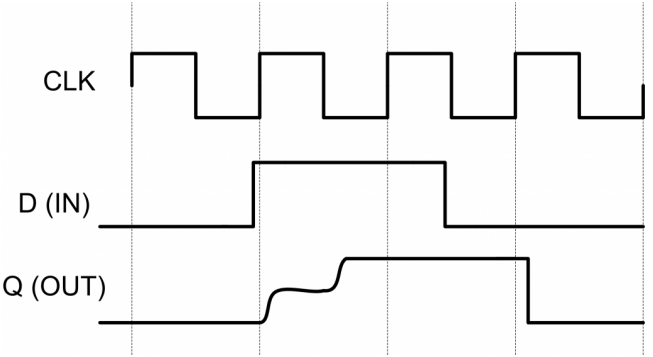
⟹**La sortie de la flip-flop au cycle d'horloge suivant est instable :**
soit 0, soit 1 ou «*une valeur*» intermédiaire (elle atteindra la bonne valeur qu'après le cycle d'horloge attendu).

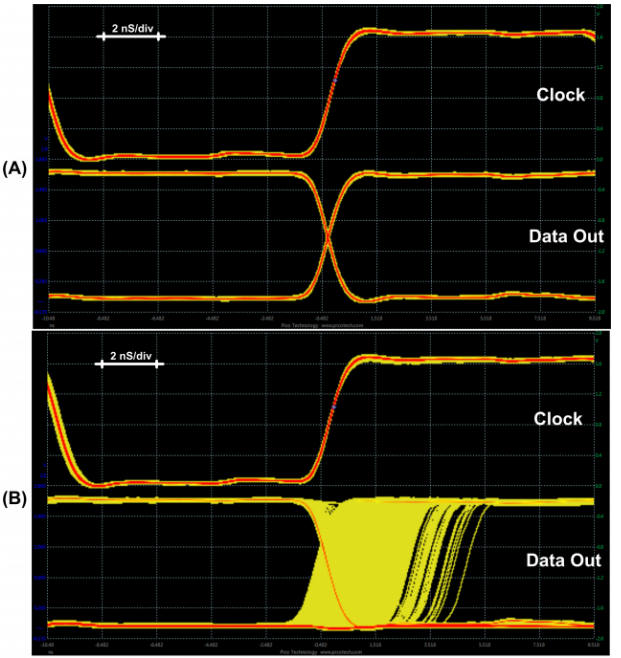
t_{SU}	Clock to Data Setup - PIO Input Register	iCE40LP384	–0.08	—	ns
		iCE40LP640	–0.33	—	ns
		iCE40LP1K	–0.33	—	ns
		iCE40LP4K	–0.63	—	ns
		iCE40LP8K	–0.63	—	ns
t_H	Clock to Data Hold - PIO Input Register	iCE40LP384	1.99	—	ns
		iCE40LP640	2.81	—	ns

Parameter	Description	Device	Min.	Max.	Units
		iCE40LP1K	2.81	—	ns
		iCE40LP4K	3.48	—	ns
		iCE40LP8K	3.48	—	ns
General I/O Pin Parameters (Using Global Buffer Clock with PLL)[3]					

Ici, on voit que le t_{su} vaut $-0,63ns$, cela veut dire que le signal peut arriver un peu après le «cycle d'horloge» *et être encore correctement enregistré par la flip-flop.*

Méta-stabilité mesurée





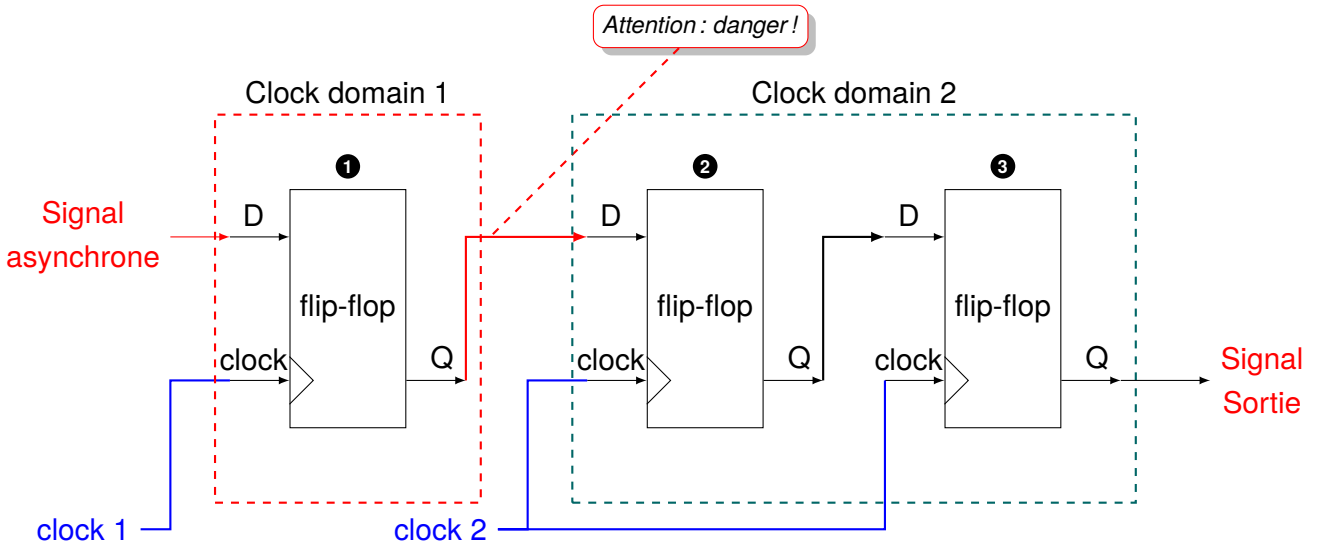
Sur cette mesure :

- ▷ *on superpose la trace de centaines modification de la valeur de sortie de la flip-flop ;*
- ▷ *en (A), la transition de 0 à 1 ou de 1 à 0 se passe correctement au cycle d'horloge si on respecte le t_{su} et le t_h ;*
- ▷ *en cas de non respect du t_{su} ou t_h, on subit de la* «méta-stabilité» *et on voit qu'il faut jusqu'à $5ns$ pour que la valeur de sortie de la flip-flop atteigne une valeur stable !*

`https://colinoflynn.com/2020/12/experimenting-with-metastability-and-multiple-clocks-on-fpgas/`

Du coup, pas de solution possible ?

Une solution pour passer d'un domaine d'horloge à un autre



Ici :

- ☐ *le* «clock domain 1» *est synchronisé sur l'horloge* «clock 1» *;*
- ☐ *le* «clock domain 2» *est synchronisé sur l'horloge* «clock 2».

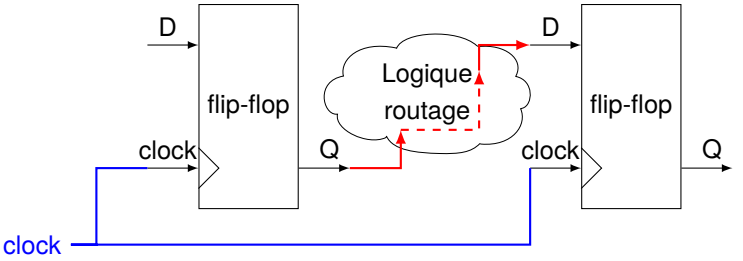
▷ *il y a un* ***risque de méta-stabilité*** *dans la flip-flop* ❶ *;*

⟹ *on utilise les flip-flop* ❷ *et* ❸ *pour* ***synchroniser*** *dans le* «domain clock 2».

⟹ *les flip-flps* ❷ *et* ❸ *sont appelées un* «synchronizer».

Donc les flip-flops induisent
des contraintes sur notre FPGA ?

Exemple de circuit



- le circuit contient deux flips-flops avec un signal circulant de la sortie de la première vers l'entrée de la seconde ;
- ce signal doit traverser un nuage de route/LUTs où va se produire des délais de propagation : plus le nuage contient de LUTs/route à traverser, ↗ plus le délai de propagation ↗
- les deux flip-flops sont pilotées par la même horloge ;
- si la première flip-flop enregistre son entrée D et la propage vers sa sortie Q :

⟹ le signal dispose d'**un seul cycle d'horloge** pour se propager de la première flip-flop vers la seconde ;

- si le signal **arrive à temps**, alors le **circuit fonctionne** ;
- si le routage et les éléments logiques à traverser induisent un **délai de propagation trop important**

⟹ il va y avoir un «*timing error*» car le design est incapable d'atteindre les objectifs fixés.

⟹ C'est à l'outil de «*pnr*» d'analyser chaque chemin et de remonter à l'utilisateur quel est le pire des chemins en terme de délai de propagation.

Le circuit a **moins d'un cycle d'horloge** à cause du «*tsu*» de la seconde flip-flop !

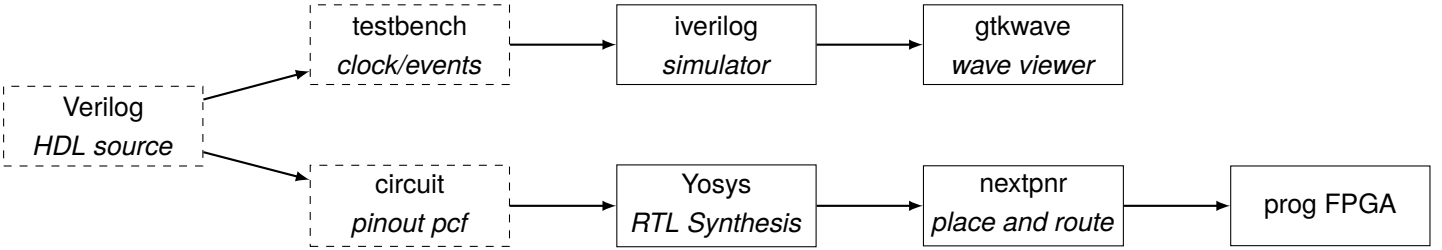
$$min(t_{\text{clock}}) = t_{\text{su}} + t_p$$

⟹ **Période d'horloge minimale** permettant au circuit de fonctionner **sans erreurs de temporisations**.

⟹ **fréquence d'horloge maximale** pour laquelle le **circuit fonctionne correctement**.

Conception
de circuits logiques
avec Verilog

- ▷ langage de **description matériel**, HDL, «*Hadware Definition Language*» ;
- ▷ utilise une **syntaxe similaire au C** : opérateurs logiques, contrôle de flow (if/else, case), sensible à la casse, pré-processeur, *etc.* ;
- ▷ «*décrit*» le **comportement**, «*behavior*», du circuit attendu :
  - ◇ le circuit est **simulé** grâce aux composants disponibles : flip-flops, LUT, RAM, DSP, *etc.* ;
  - ◇ les composants utilisés sont ensuite sélectionnés et reliés entre eux : «*place and route*» ;
  - ◇ un calcul de faisabilité est réalisé sur la proposition : contraintes dues aux temps de propagation des chemins définis lors du placement/routage, limite des fréquences utilisables, *etc* ;
  - ◇ si les contraintes sont :
    - ⋆ **respectées** : le placement/routage est accepté, une solution est atteinte ;
    - ⋆ **non respectées** : on essaye un nouveau placement/routage pour obtenir une nouvelle proposition ;
- ▷ il contient :
  - ◇ des parties «*synthétisables*» : pouvant donner lieu à un circuit physique ;
  - ◇ des parties **non synthétisable** : affichage, delais, par exemple pour faire des simulations, tests ou déboguages ;
- ▷ Seul le **résultat synthétisable** peut être traduit en :
  - ◇ FPGA, «*Field Programmable Gate Array*» ;
  - ◇ ASIC, «*Application-Specific Integrated Circuit*» ;
- ▷ un **concurrent** : VHDL, «*VHSIC Hardware Description Language*» avec VHSIC signifiant, «*Very High Speed Integrated Circuit*» ;
- ▷ des **évolutions** : **System Verilog**, qui ajoute des capacités de **vérification** au design (des types de données par exemple).



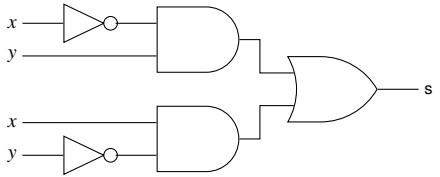
- **yosys** :
  - framework for RTL, «*Register-Transfer Level*» synthesis tools ; It currently has
  - extensive Verilog-2005 support ;
  - basic set of synthesis algorithms for various application domains.
- **nextpnr** :
  - vendor neutral, timing driven, FOSS FPGA place and route tool.
  - Lattice iCE40 devices supported by Project IceStorm
  - Lattice ECP5 devices supported by Project Trellis
  - Lattice Nexus devices supported by Project Oxide
  - Gowin LittleBee devices supported by Project Apicula
  - (experimental) Cyclone V devices supported by Mistral
  - (experimental) Lattice MachXO2 devices supported by Project Trellis

celui qu'on va utiliser

- **icarus verilog**, «*iverilog*» :
  - simulator.
- **gtkwave** :
  - wave viewer, scriptable with TCL.

Verilog : notion de Modules et ports

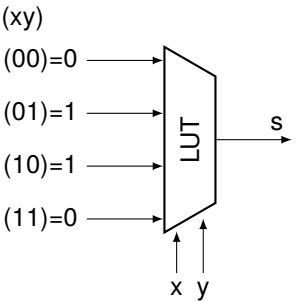
Le circuit logique du «*xor*» :



Son comportement exprimé en Verilog :

```
assign s = x ^ y;
```

Sa simulation en FPGA :



La notion de module

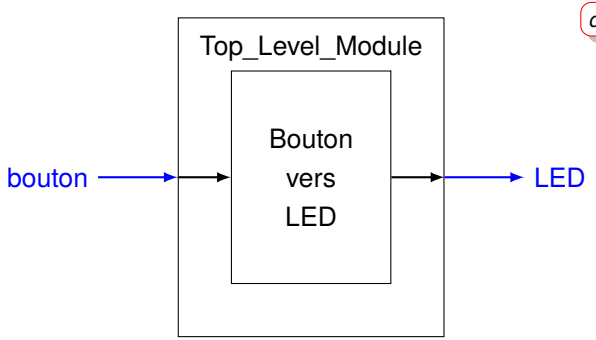
- le **module** est l'élément de base du circuit : il fournit **abstraction** et **encapsulation** ;
- un module doit être écrit dans un **fichier verilog** (extension «*.v*») dont le nom correspond à celui du module ;
- un module est constitué de :
  - une **liste de «*ports*»** ;
  - le code verilog implémentant les fonctionnalités désirées ;
- les «*ports*» permettent les **connexions** entre un module et son environnement :
  - chaque «*port*» possède un nom et un type : `input`, `output` ou `inout` ;
  - les «*ports*» sont déclarés dans la partie «*port declaration*» du module :

```
module example_design (input entree, output sortie);

   assign sortie = entree ;
endmodule
```

les ports du module

Le module de «*Top Level*»

- tous les circuits complets ont un module «*top-level*» qui est au sommet de la hiérarchie des différents modules ;
- le module «*top-level*» définit les E/S pour l'intégralité du système digital ;
- **tous les modules** du circuit complet sont **instanciés** à l'intérieur de ce module «*top-level*» :




```
// Définition d'un module
module bouton_vers_LED(input e,
                       output s);
   assign s = e;
end

// Définition du module top-level
module Top_Level_Module
   (input bouton, output LED);

   //instanciation du module
   bouton_vers_LED bvl(
      .e(bouton),
      .s(LED)
      );
endmodule
```

définition

connexions

instanciation

- **L'instanciation** du module «`bouton_vers_led`» définit le module appelé «`bvl`» ;
- Les **connexions** «`.e(bouton)`» et «`.s(LED)`» connectent les E/S du module «`bvl`» à celles du module top-level.

Wires : fil simple ou fil multi bits

«*Wires*» ou «*fils*»

- ▷ correspondent à des **fils de connexion** : transmettre des valeurs entre entrées et sorties ;
- ▷ doivent être **déclarés** avant d'être utilisés :

```
wire a;
wire b;
```

- ▷ peuvent être **scalaire** (1 seul bit) ou **vecteurs** (plusieurs bits) :

```
wire [7:0] d; // 8bits
wire [31:0] e; // 32 bits
```

- ▷ sont définis en **vecteur multi-bits** avec la syntaxe `[MSB bit index : LSB bit index]` avant le nom du signal pour définir sa taille en bits :

```
module traitement_sur_deux_bits ( input [1:0] x, output [1:0] y);
   wire [1:0] valeur_temporaire;
   ...
endmodule
```

- ▷ lorsque l'on connecte des **entrées multi-bits** vers des **sorties multi-bits**, **le nombre de bits doit correspondre** !
- ▷ un fil, «*wire*», peut être affecté à des équations logiques, d'autres fils ou des opérations réalisées sur ses fils :
  - ◇ on utilise l'instruction «*assign*» :
    - ⋆ **l'argument de gauche** de «`assign`» doit être un «*wire*» mais ne peut pas être un «`input wire`» ;
    - ⋆ **l'argument de droite** de «`assign`» peut être n'importe quelle expression consituées d'opérateurs de Verilog et de «*wires*» :

```
module connecter(input a, output b);
   assign a = !b;
endmodule
```

Et en terme de placement/routage
ca donne quoi ?

Réaliser un circuit dans un FPGA

C’est résoudre un problème de contraintes

Retournons sur Verilog

Opérateur	Symbol	Opération
arithmétique	+	addition
	-	soustraction
	*	multiplication
	/	division
	%	module
logique	!	non logique
	&&	et logique
	||	ou logique
relationnel	>	plus grand que
	<	moins grand que
	>=	plus grand que ou égal
	<=	moins grand que ou égal
égalité	==	égal
	!=	différent

Opérateur	Symbol	Opération
bit à bit	~	négation
	&	et
	|	ou
	^	xor
décalage	<<	décalage à gauche logique
	>>	décalage à droite logique
	<<<	décalage à gauche arithmétique
	>>>	décalage à gauche arithmétique
concaténation	{}	grouper des bits
duplication	{{}}	dupliquer des bits
indexer/découper		sélectionner des bits

Exemples

```
wire [7:0] a;
wire [31:0] b;
wire [31:0] c;
assign c = {a, b[23:0]};// concaténation et découpage
assign c = { 32{a[5]} };// duplication et indexage
```

```
wire operande1 [31:0];
wire operande2 [31:0];
wire resultat [31:0];
assign resultat = operande1 + operande2;
```

Opérateur ternaire : «*if-else*» avec une instruction «*assign*»

```
assign sortie = a > 10 ? 10 : a ; // affecte 10 si a > 10, sinon affecte a
```

Toutes les valeurs sont binaires

Entrée des valeurs

`[nbre de bit]'[base][valeur]`

où la base peut être :

- `d` : décimal ;
- `h` : hexadécimal ;
- `o` : octal ;
- `b` : binaire.

Attention

Il est important de faire correspondre le nombre de bits des opérateurs et des connexions entre modules !

Exemples

`2'd 2`	la valeur sur 1 sur 2 bits : `10`
`16'h abcd`	la valeur `0xabcd` en hexadécimal
`8'b 10001100`	la valeur `0b10001100`

Registre

- permettent de mémoriser un état qui peut changer :

syntaxe de 2001 vs syntaxe de 1995

```
reg x;
reg [31:0] y; // ou reg y [31:0]
```

- différent du «*wire*» qui sert à connecter ou à fixer une valeur :

```
wire [1:0] c = 2'b 01;
```

Bloc de logique combinatoire, «*Combinational Logic Blocks*» : le «*always @(*)*»

```
reg x;
reg y;

always @(*) begin
   x = ~y;
end
```

la liste de sensibilité

La liste de «*sensibilité*» ou «*sensitivity list*» est la liste des signaux à surveiller et dont la modification entraîne la ré-évaluation du bloc.

Ici, on mets `(*)`, ce qui veut dire que le bloc est recalculé pour toute modification des signaux qu'il utilise en entrée, c-à-d ici y.

L'instruction conditionnelle «*if-else*»

```
wire w;
wire x;
wire y;
reg [1:0] z;

always @(*) begin
   if (x) begin
      z[0] = x;
      z[1] = y;
   end
   else begin
      z[0] = y;
      z[1] = x;
   end
end
```

- ▷ l'utilisation de «*if-else*» entraîne la génération de multiplexeurs en hardware ;
- ▷ si plusieurs opérations sont nécessaires, il faut les encapsuler dans un «*begin-end*» ;

Attention

Il est important de traiter toutes les valeurs de la condition.

Le «*case*» : énumérer les valeurs possibles

```
reg [1:0] x;
reg [1:0] y;
always @(*) begin
   case(x)
      0: y = 2'd 2;
      1: y = 2'd 3;
      2: y = 2'd 1;
      default: y = 2'd 0;
   endcase
end
```

Ce code va générer plusieurs multiplexeurs pour le mettre en œuvre en hardware.

Attention

Il est important de mettre un «*default*».

Attention à ne pas générer des «*latches*» au lieu de «*flip-flops*»

Un latch ne dépend pas de l'horloge pour enregistrer sa valeur⟹il peut être ***imprévisible*** *et doit être éviter*.

Code pouvant générer un «*latch*» :

```
wire [1:0] x;
reg  [1:0] y;
always @(*) begin
   if(x == 2'b10) begin
         y = 2'd3;
     end else if(x == 2'b11) begin
         y = 2'd2;
      end
 end
```

Ici, on affecte pas de valeurs au registre y quelles que soient les valeurs de x.

Code ne générant pas de «*latch*» :

```
wire [1:0] x;
reg  [1:0] y;
always @(*) begin
    y = 2'b00;
    if(x == 2'b10) begin
        y = 2'd3;
    end else if(x == 2'b11) begin
        y = 2'd2;
    end
end
```

valeur utilisée par défaut

Ici, il y a une valeur par défaut.

Éviter les «*latches*»

Génération involontaire de «*latches*»

entree A	entree B	sortie Q
0	0	0
0	1	1
1	0	1
1	1	undefined

Ici,

- ▷ *la sortie vaut zéro si les deux entrées valent zéro ;*
- ▷ *la sortie vaut un si les deux entrées sont différentes ;*

***Mais** que se passe-t-il si les deux entrées valent un ?*

L'outil de programmation FPGA va conclure que dans ce cas là, la sortie doit conserver son état précédent !

⟹ *une «*latch*» va être générer pour mémoriser cet état sans utiliser l'horloge.*

Le comportement sera alors le suivant :

- ▷ *si la valeur de la sortie est 0 et que les deux entrées sont à 1 alors la sortie reste à 1 ;*
- ▷ *si la valeur de la sortie est 1 et que les deux entrées sont à 1 alors la sortie reste à 1 ;*

Attention

Dans un «*case*» ou une succession de «*if-else*» il est nécessaire de traiter **toutes les combinaisons** possibles des conditions et/ou d'utiliser une **valeur par défaut**.

Les blocs de logiques séquentielles

- ne sont mis à jour que lors d'un **cycle d'horloge** ;
- utilisent un **identifiant spécial** dans leur liste de sensibilitée : «*posedge*» qui signifie «*front montant*»

```
reg [1:0] x;

always @(posedge clk) begin
   x <= x + 1;
end
```

Ici, on incrémente x tous les cycles d'horloge.

Affectation bloquante et non bloquante

Verilog dispose de deux opérateurs d'affectation :

- `<=` : **affectation non bloquante** $\Longrightarrow$ réservé pour les blocs de logique séquentielle ;
  «Non bloquant» *signifie qu'il n'empêche pas* ***les autres affectations*** *d'avoir* ***lieu en même temps***.
  $\Longrightarrow$ *toutes les affectations ont lieu* ***en même temps*** *et seront* ***valides au prochain cycle d'horloge***.
- `=` : **affectation bloquante** $\Longrightarrow$ réservé pour les blocs de logique combinatoire.
  «Bloquant» *signifie que les* ***affectations suivantes*** *auront lieu* ***uniquement après celle-ci***.
  $\Longrightarrow$*les affectations sont réalisées* ***dans l'ordre d'écriture*** *dans le source Verilog.*

Exemple : un compteur qui s'incrémente à chaque cycle d'horloge

```
reg [1:0] x;
reg [1:0] next_x;

always @(*) begin
   next_x = x + 1;
end

always @(posedge clock) begin
   x <= next_x;
end
```

bloc combinatoire

bloc séquentiel

ne sera valide que lors du prochain cycle d'horloge

Ici :

- *le* ***bloc combinatoire*** *réagit à chaque modification de ces entrées : dès que x est modifié alors `next_x` est lui aussi modifié (après que l'addition soit finie) ;*
- *le* ***bloc séquentiel*** *ne réagit que lors d'un nouveau cycle d'horloge : à ce moment là il enregistre la valeur courante de `next_x` dans x*

Verilog : simulation du compteur en ligne

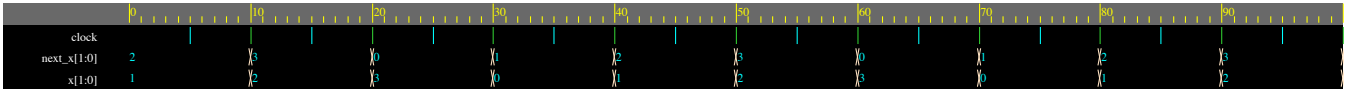
Utilisation de https://www.edaplayground.com/

testbench.sv

```
module testbench();
  reg clock;
  initial begin
    clock = 1'b1;
    forever #5 clock = ~clock;
  end
  compteur dut(
    .clock(clock)
  );
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    #100 $finish;
  end
endmodule
```

design.sv

```
module compteur(input clock);
  reg [1:0] x = 0;
  reg [1:0] next_x = 0;

   always @(*) begin
      next_x = x + 1;
   end

   always @(posedge clock) begin
      x <= next_x;
   end
endmodule
```



Verilog : simulation du compteur en ligne – version 2

Utilisation de https://www.edaplayground.com/

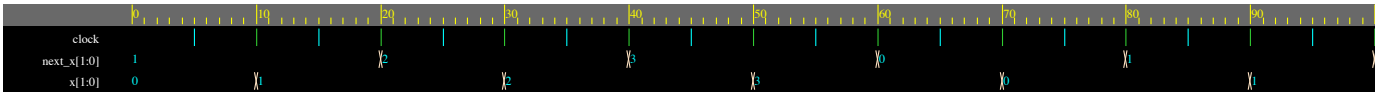
testbench.sv

```
module testbench();
  reg clock;
  initial begin
    clock = 1'b1;
    forever #5 clock = ~clock;
  end
  compteur dut(
    .clock(clock)
  );
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    #100 $finish;
  end
endmodule
```

design.sv

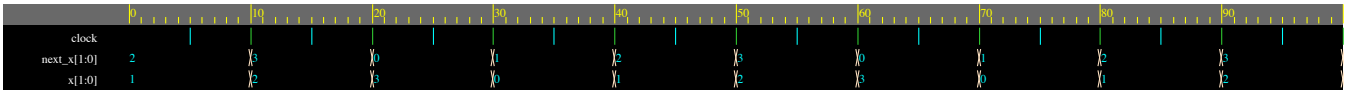
```
module yop(input clock);
  reg [1:0] x = 0;
  reg [1:0] next_x = 0;

   always @(posedge clock) begin
        x <= next_x;
        next_x <= x + 1;
   end
endmodule
```

on supprime la partie combinatoire.



Brought to you by DOULOS (http://www.doulos.com)

et la version précédente :



Brought to you by DOULOS (http://www.doulos.com)

Verilog : simulation du compteur en ligne – version 3

Utilisation de https://www.edaplayground.com/

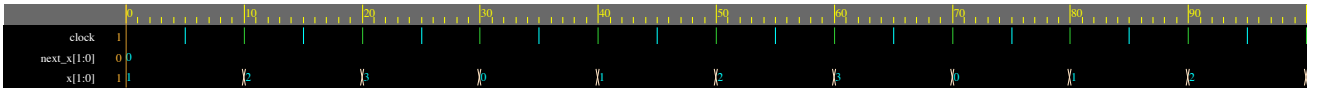
testbench.sv

```
module testbench();
  reg clock;
  initial begin
    clock = 1'b1;
    forever #5 clock = ~clock;
  end
  compteur dut(
    .clock(clock)
  );
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
    #100 $finish;
  end
endmodule
```

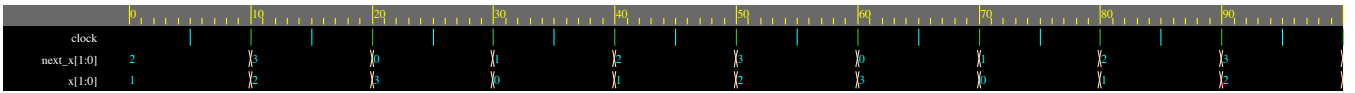
design.sv

```
module yop(input clock);
  reg [1:0] x = 0;

   always @(posedge clock) begin
        x <= x + 1;
   end
endmodule
```

on supprime l'utilisation de `next_x`.



et la version 1 :

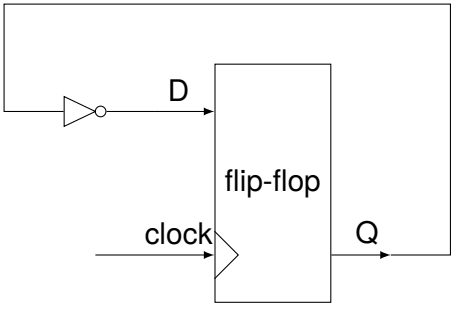


Un circuit oscillant

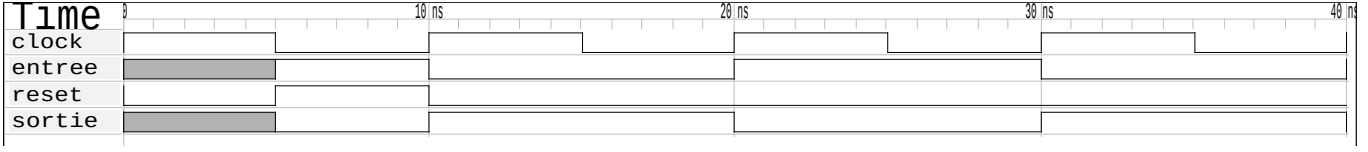
```
module oscillator
(input clock, input reset, output reg sortie);

wire entree;

assign entree = ~sortie;

always @(posedge clock or posedge reset) begin
  if (reset)
    sortie <= 0;
  else
    sortie <= entree;
end
endmodule
```

combinatoire

séquentielle



La flip-flop change de valeur à chaque cycle d'horloge :



Ceci est la sortie de `gtkwave`.

Un circuit oscillant : simulation avec «*icarus verilog*»

```
`include "oscillator.v"

`timescale 1ns / 1ps

module tb ();
wire sortie;
reg clock;
reg reset;

oscillator dut(.clock(clock),
   .reset(reset), .sortie(sortie));

initial begin
  clock = 1;
  forever #5 clock <= ~clock;
end

initial begin
  reset = 0;
  #5 reset = 1;
  #5 reset = 0;
end

initial begin
  $monitor("time=%3d, sortie=%b",$time, sortie);
  $dumpfile("oscillator.vcd");
  $dumpvars(0, tb);
  #40 $finish ;
end
endmodule
```

À chaque cycle d'horloge, la flip-flop change de valeur :

xterm

```
$ iverilog -o tb tb.v
$ ./tb
VCD info: dumpfile
oscillator.vcd opened for
output.
time=  0, sortie=x
time=  5, sortie=0
time= 10, sortie=1
time= 20, sortie=0
time= 30, sortie=1
time= 40, sortie=0
```

Attention

- le bloc `initial begin ... end` n'est **pas synthétisable** !
  ⟹il doit être uniquement utilisé dans un «*banc test*».
- les temps d'attentes `#5` indique d'attendre $5ns$: ils ne sont **pas synthétisables** !
  ⟹il doit être uniquement utilisé dans un «*banc test*».
- le bloc `forever #5` n'est **pas synthétisable** !
  ⟹il doit être uniquement utilisé dans un «*banc test*».

Définition de constantes

```
   localparam coeff = 5;
   reg [31:0] x;
   reg [31:0] y;

   always @(*) begin
      x = coeff * y;
   end
```

- `parameter` : défini un paramètre global ;
- `localparam` : défini un paramètre local au module.

Module paramètrable

On peut définir un module paramétrable :

```
module circuit #(parameter taille=32)
   (input [taille-1:0] entree,
   output [taille-1:0] sortie);
   sortie = ~a;
endmodule

module top_level();
   localparam circuit1_taille = 64;
   localparam circuit2_taille = 32;
   reg [circuit1_taille-1:0] e1;
   reg [circuit2_taille-1:0] e2;
   wire [circuit1_taille-1:0] s1;
   wire [circuit2_taille-1:0] s2;
   circuit #(.taille(circuit1_taille)) circuit1 (.entree(e1), .sortie(s1));
   circuit #(.taille(circuit2_taille)) circuit2 (.entree(e2), .sortie(s2));
endmodule
```

Choix d'un «*reg*» ou d'un «*wire*» ?

Comment choisir entre «*reg*» et «*wire*» ?

- ▷ si un signal doit être affecté **à l'intérieur** d'un bloc «*always*»
  $\Longrightarrow$ il doit être déclaré comme un «*registre*» ;

- ▷ si un signal est **affecté** comme un «*continuous assignment*», c-à-d en dehors d'un bloc «*always*»
  $\Longrightarrow$ il doit être déclaré comme un «*wire*» ;

  *Un **continuous assignment** est une affectation* «combinatoire» *où dès que la partie droite de l'affectation change, elle est prise en compte par la partie gauche de l'affectation :*

```
wire a, b, c;

assign a = b & c;
```

  Dès que `b` *ou* `c` *change de valeur, l'expression* `b & c` *est évaluée et* `a` *est mis à jour avec le résultat. Dans le FPGA, il est probable qu'il sera simulé par une ou plusieurs LUTs*.

- ▷ par défaut les entrées et sortie d'un module sont des «*wires*»

  mais si un port de sortie doit être affecté dans un bloc «*always*», alors il doit être déclaré explicitement comme un registre :

```
module circuit (output reg sortie);
```

Et si on voulez faciliter l’accès à des données ?

On peut définir des **registres indexables** :

```
reg [M:0] mémoire [N:0]
```

Ce qui définit :

▷ un tableau de `N+1` éléments ;

▷ où chaque élément est constitué de `M+1` bits.

```
reg [31:0] memoire [7:0]; // définit un tableau de 8 valeurs sur 32bits

memoire[2] // le troisième élément du tableau
memoire[5][7:0] // l'octet de poids faible du sixième élément du tableau
```

Ce code donnera lieu à l'utilisation de la **mémoire** présente dans le FPGA, appelée BRAM.

La quantité de mémoire disponible dans un FPGA est indiquée en bits et cette mémoire peut être regroupée suivant des mots du nombre de bits voulu.

Par exemple, le ice40 hx8k propose 80kb de BRAM, soit 10ko de mémoire maximum.

Pour «*inférer*» par le synthétiseur de la mémoire BRAM, c-à-d de la mémoire interne au FPGA, il faut définir un module qui possède une «*interface*» propre à l'accès à la mémoire en lecture comme en écriture :

Le module de mémoire :

```
module memoire #(
  parameter WORD = 8,
  parameter TAILLE = 10
)
(input clock,
    input w_en,
    input r_en,
    input [TAILLE_ADRESSE - 1:0] w_addr,
    input [TAILLE_ADRESSE - 1:0] r_addr,
    input [WORD - 1:0] w_data,
    output reg [WORD - 1:0] r_data
  );
  localparam TAILLE_ADRESSE = $clog2(TAILLE);
  reg [WORD - 1 :0] mem [0:TAILLE - 1];
  always @(posedge clock) begin
    if (w_en == 1'b1) begin
      mem[w_addr] <= w_data;
    end
    if (r_en == 1'b1) begin
      r_data <= mem[r_addr];
    end
  end
endmodule
```

Pour utiliser le module mémoire :

```
module led_sequencer(
  input clock,
  input slow_clock,
  input reset,
  output reg [3:0] LEDS);

  reg w_en = 0;
  reg r_en = 1;
  reg [3:0] r_addr;
  reg [3:0] w_addr = 0;
  reg [7:0] w_data;
  wire [7:0] r_data;

  memoire #(.WORD(8), .TAILLE(10))
  config_leds
  (
    .clock(clock),
    .w_en(w_en),
    .r_en(r_en),
    .w_addr(w_addr),
    .r_addr(r_addr),
    .w_data(w_data),
    .r_data(r_data)
  );

  reg [3:0] adresse_lecture = 0;

...
```

Il est possible que l'outil de synthèse simule de la mémoire avec des registres (read/write) ou des LUTs (read only).

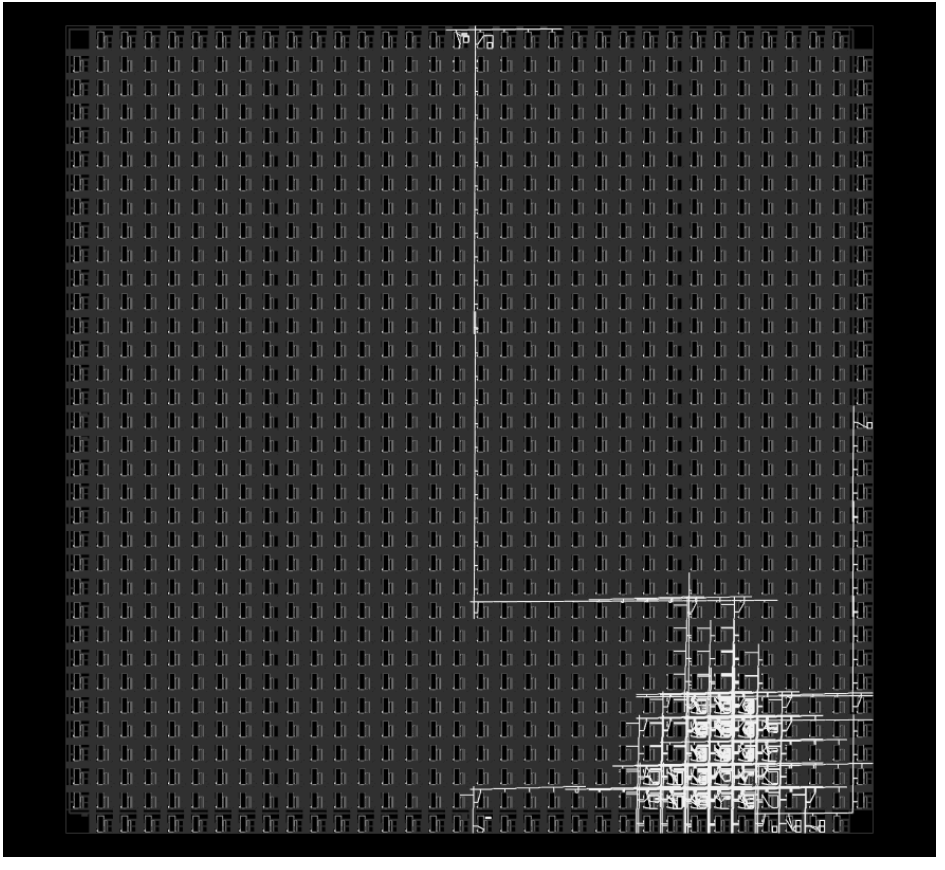
Inférence par l'outil de synthèse de l'utilisation de mémoire

xterm

```
yosys -ql synthese.log -p 'synth_ice40 -top top_level -json synthese.json' synthese.v
nextpnr-ice40 --freq 90 --hx8k --package tq144:4k --asc synthese.asc --pcf blackice-ii.pcf --json
synthese.json

Info: Packing constants..
Info: Packing IOs..
Info: Packing LUT-FFs..
Info:       26 LCs used as LUT4 only
Info:       41 LCs used as LUT4 and DFF
Info: Packing non-LUT FFs..
Info:       22 LCs used as DFF only
Info: Packing carries..
Info:        0 LCs used as CARRY only
Info: Packing indirect carry+LUT pairs...
Info:        0 LUTs merged into carry LCs
Info: Packing RAMs..
Info: Placing PLLs..
Info: Packing special functions..
Info: Packing PLLs..
Info: Promoting globals..
Info: promoting clock$SB_IO_IN (fanout 65)
Info: promoting reset_SB_LUT4_I3_3_O [reset] (fanout 24)
Info: Constraining chains...
Info:        2 LCs used to legalise carry chains.
Info: Checksum: 0x7d34fe38

Info: Device utilisation:
Info:            ICESTORM_LC:    93/ 7680     1%
Info:           ICESTORM_RAM:     1/   32     3%
Info:                  SB_IO:     8/  256     3%
Info:                  SB_GB:     2/    8    25%
Info:           ICESTORM_PLL:     0/    2     0%
Info:            SB_WARMBOOT:     0/    1     0%
```

Yosys a utilisé de la mémoire !

Le placement réalisé par nextpnr



```
`include "fichier.v"

`ifndef CONSTANTES
`define CONSTANTES

`define NOMBRE_BITS 32
`endif
```

Fonctions utiles

$clog2(x)	retourne la valeur arrondie à la valeur supérieure de $log_2(x)$
$floor(x)	arrondi à la valeur inférieure
$ceil(x)	arrondi à la valeur supérieure

Exemple :

un compteur allant jusqu'à la valeur 500 000

```
reg [$clog2(500000):0] compteur;
```

Si on veut tester dans le simulateur «*iverilog*» :

```
module sortie_verilog;
initial
begin
  $display("log2(500000)");
  $display($clog2(500000));
  $finish ;
end
endmodule
```

xterm

```
$ iverilog -o sortie_verilog sortie_verilog.v
$ ./sortie_verilog
log2(500000)
        19
```

Un diviseur d'horloge paramétrable

*On peut l'utiliser comme «*clock*» d'un circuit séquentiel, c-à-d que l'horloge de ce circuit est différente des autres.*

```
`timescale 1ns / 1ps
// HORLOGE_ENTREE 100_000_000
// HORLOGE_SORTIE 4
// COMPTEUR =HORLOGE_ENTREE/HORLOGE_SORTIE = 25000000
//`define COMPTEUR 25000000
`define COMPTEUR 25

module clock_divider(input clock, output reg slow_clock, input reset);
  //parameter COMPTEUR = 25000000;
  parameter COMPTEUR = 25;
  reg [$clog2(COMPTEUR):0] compteur;

  always @(posedge clock) begin
    if (reset) begin
      compteur <= 0;
      slow_clock <= 0;
    end
    else begin
      compteur <= compteur + 1;
      if (compteur == COMPTEUR) begin
      compteur <= 0;
        slow_clock <= ~ slow_clock;
      end
    end
  end
endmodule
```

Mais attention : on passe d'un domaine d'horloge à un autre, ce qui peut être dangereux...

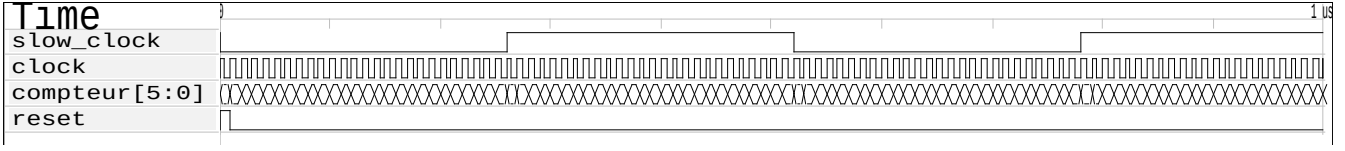
Cette solution est préférable car elle n'est pas utilisée pour piloter un circuit mais simplement comme référence :

```
`ifndef VALEURCOMPTEUR
  `define HORLOGE_ENTREE 100000000
  `define HORLOGE_SORTIE 16
  `define VALEURCOMPTEUR (`HORLOGE_ENTREE/`HORLOGE_SORTIE)
`endif

module clock_divider(input clock, output reg slow_clock, input reset);
localparam COMPTEUR = `VALEURCOMPTEUR;
reg [$clog2(COMPTEUR+1):0] compteur = 0;

always @(posedge clock) begin
  if (reset) begin
    compteur <= 0;
    slow_clock <= 0;
  end
  else begin
    if (compteur == COMPTEUR) begin
      compteur <= 0;
      slow_clock <=  1;
    end
    else begin
      compteur <= compteur + 1;
      slow_clock <= 0;
    end
  end
end
endmodule
```

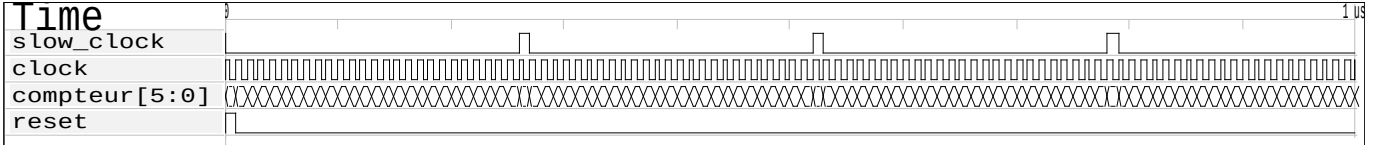
On appelle cette version «clock enabled» *car dans le circuit séquentiel l'utilisant :*

```
always @(posedge clock or posedge reset) begin
...
     if (slow_clock) begin
         // traitement lié à l'horloge dérivée
   end
 end
```

Diviseur d'horloge



Clock enabled ⟹ impulsion régulière

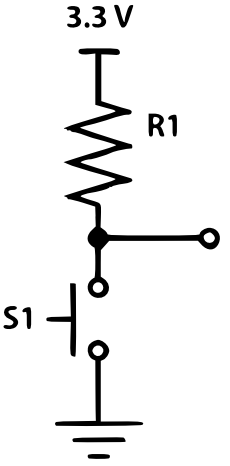


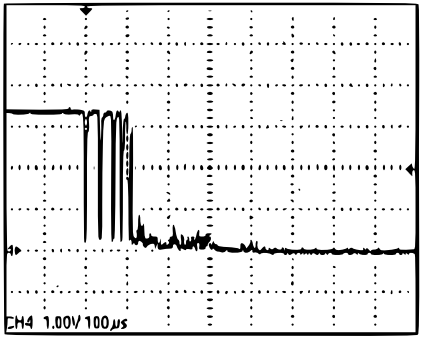
Comment faire des circuits plus complexes ?

$\Longrightarrow$les FSMs !

Debouncing

*Lorsque l'on appui sur un **bouton mécanique**, la partie réalisant le contact électrique peut rebondir,* «bounce»*, ce qui ouvre et ferme le circuit très rapidement : oscillation observée sur la trace d'oscilloscope*





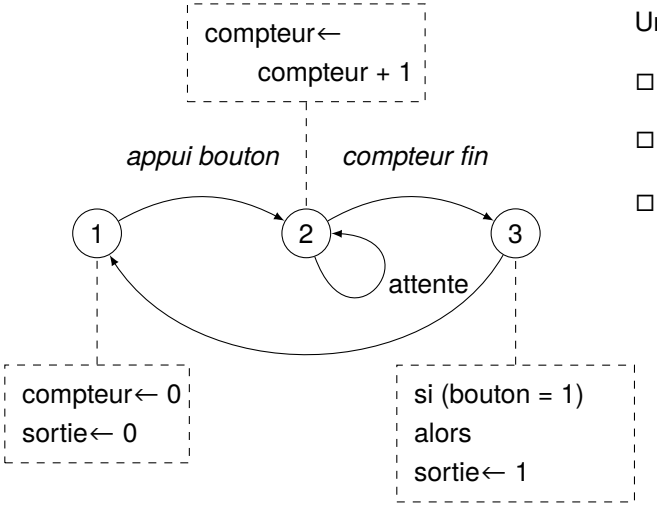
Ici, le bouton est «active low»*:*

- ☐ *lorsqu'il est appuyé il met le signal à zéro ;*
- ☐ *lorsqu'il est relâché, la résistance* «pull-up» *mets le signal à un.*

Circuit réalisant le debouncing

- ▷ détecter l'appui du bouton ;
- ▷ attendre au-dessus de $20 - 40ms$;
- ▷ si le bouton est encore appuyé : valider son appui.

*L'appui du bouton est également **asynchrone** et il peut être nécessaire de le **synchroniser** au reste du circuit.*



Un **automate à nombre fini d'états** est composé de :

- **états** : *ici, 3 états* ;
- **transitions** : *un événement*;
- **actions** :
  - *incrémenter le compteur*
  - *envoyer une pulsation sur la sortie*

En FPGA :

- **logique séquentielle**, pour «*parcourir*» l'automate en fonction de l'horloge ;
- **registre** pour mémoriser un état : la taille du registre est proportionnelle au nombre d'états (log_2) ;
- les **actions** :
  - modifier les valeurs de registres externes à l'automate ;
- les **transitions** :
  - évaluer des conditions ;
  - faire évoluer le registre d'état ;

  ⟹un «*case*» en verilog

  ⟹ne pas oublier le cas «*default*» si on énumére pas toutes les valeurs possibles des états compte tenu de la taille du registre d'état

  (exemple 10 transitions parmis les 16 possibles d'un registre sur 4bits).

FSM : le debouncing

pour les 100MHz du FPGA

```
`ifndef HORLOGE
`define HORLOGE 100000000
`endif
`define NOMBRE_ETATS 3
`define MAXVALUE (`HORLOGE*40/1000)

module debouncer (
  input clock,
  input reset,
  input bouton,
  output reg sortie);

reg [$clog2(`NOMBRE_ETATS)-1:0] etat;
reg [$clog2(`MAXVALUE+1):0] compteur;

localparam DEBUT = 0;
localparam ATTENTE = 1;
localparam FIN = 2;
```

les états

- ▷ lors du reset, on initialise l'automate à l'état de début ;
- ▷ les différentes actions sont exprimées dans le `case` ;
- ▷ les transitions sont effectuées lors d'événements :
  - ◇ passer de l'état `DEBUT` vers `ATTENTE` lors de l'appui du bouton ;
  - ◇ de l'état `ATTENTE` vers `FIN` lorsque le compteur est terminé ;

```
always @(posedge clock or posedge reset) begin
  if (reset) begin
    etat <= DEBUT;
    compteur <= 0;
  end
  else begin
    case (etat)
      DEBUT: begin
        compteur <= 0;
        sortie <= 0;
        if (bouton) begin
          etat<= ATTENTE;
        end
      end
      ATTENTE: begin
        compteur <= compteur + 1;
        if (compteur >= `MAXVALUE)
          etat <= FIN;
      end
      FIN:
        if (bouton) begin
          sortie <= 1;
          etat <= DEBUT;
        end
        else
          etat <= DEBUT;
      default:
        etat <= DEBUT;
    endcase
  end
end
endmodule
```

Dans l'état `FIN`, on mets la `sortie` à 1, puis on transite vers l'état `DEBUT` où la sortie repasse à zéro

⟹ Une pulsation est transmise sur la sortie.

```
`define HORLOGE 1000
`include "debouncer.v"

`timescale 1ns / 1ps

module tb ();
reg clock;
reg reset;
reg bouton;
wire sortie;

debouncer dut (
  .clock(clock),
  .reset(reset),
  .bouton(bouton),
  .sortie(sortie)
);
initial begin
  bouton = 0;
end

initial begin
  clock = 1;
  forever #5 clock = ~clock;
end
```

le circuit de «test_bench» *ou* «banc test»

instanciation du module et connexion

condition de départ

génération de l'horloge

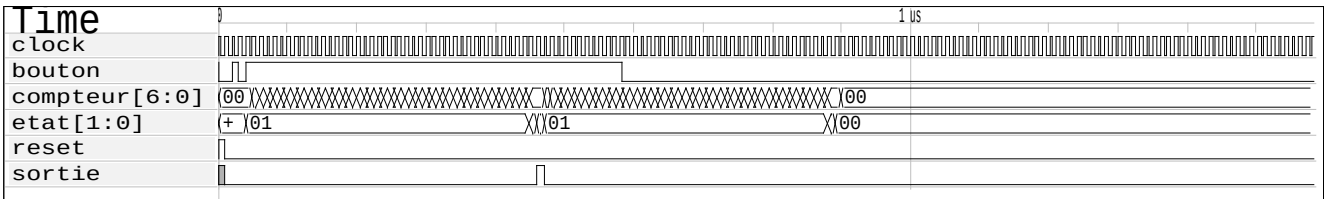
```
initial begin
  reset = 1;
  #10 reset = 0;
end

initial begin
  $monitor("time %3d bouton %b sortie %b\n",
$time, bouton, sortie);
  $dumpfile("debouncer.vcd");
  $dumpvars(0, tb);

  #22 bouton = 1;
  #8 bouton = 0;
  #10 bouton = 1;
  #543 bouton = 0;

  #1000 $finish;
end

endmodule
```

envoi du reset

création des événements

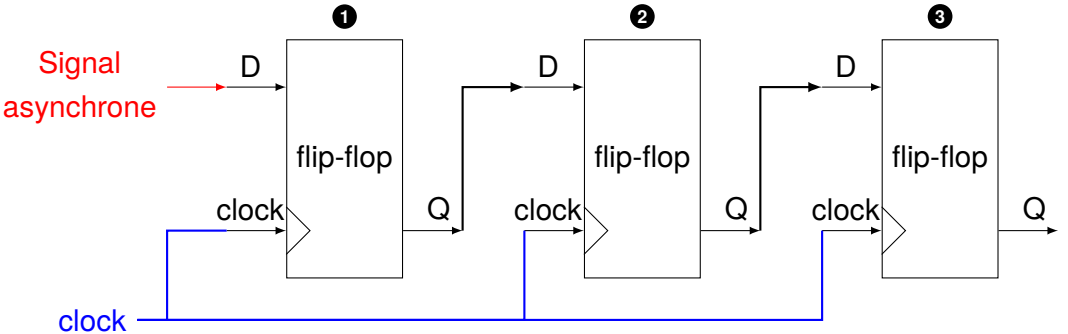
terminaison

- ▷ `$monitor` : affiche les contenus des signaux à chaque modification ;
- ▷ `$dumpfile` et `$dumpvar` : enregistre un fichier au format **gtkwave**



Mais un bouton, c’est pas de l’asynchrone ?

Comment détecter un événement asynchrone dans un circuit FPGA synchrone ?



Le «signal asynchrone» *est en entrée de la flip-flop* ❶ *:*

▷ *le signal va* ***probablement*** *se modifier pendant le* «time to setup» *ou* «time to hold» *de la flip-flop ;*

⟹ *le signal va la mettre en* «metastability»

⟹ *il va falloir un délai inconnu avant que la flip-flop atteigne un état stable.*

⟹ *la sortie de* ❶ *va finalement être synchronisée sur l'horloge ;*

Pour détecter la transition, «edge»*, montante ou descendante, on compare les sorties de* ❷ *et* ❸ *:*

▷ *si elles sont différentes : une transition est détectée ;*

▷ *suivant les valeurs de* ❷ *et* ❸ *on sait si elle est montante ou descendante.*

Simulation du «*edge detector*»

```
module edge_detect (input async_sig,
                    input clk,
                    output reg rise,
                    output reg fall);

  reg [1:3] resync;

  always @(posedge clk)
  begin
    // detect rising and falling edges.
    rise <= resync[2] & !resync[3];
    fall <= resync[3] & !resync[2];
    // update history shifter.
    resync <= {async_sig , resync[1:2]};
  end

endmodule
```

xterm

```
$ ./edge_tb
time   0 async 0, fall x, rise x
time  20 async 0, fall 0, rise x
time  30 async 0, fall 0, rise 0
time  79 async 1, fall 0, rise 0
time 100 async 1, fall 1, rise 0
time 110 async 1, fall 0, rise 0
time 141 async 0, fall 0, rise 0
time 170 async 0, fall 0, rise 1
time 180 async 0, fall 0, rise 0
time 191 async 1, fall 0, rise 0
time 220 async 1, fall 1, rise 0
time 230 async 1, fall 0, rise 0
time 250 async 0, fall 0, rise 0
time 270 async 0, fall 0, rise 1
time 280 async 0, fall 0, rise 0
```

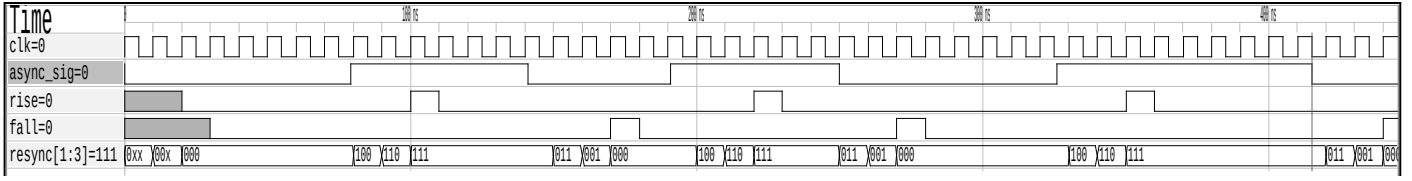
```
`timescale 1ns / 1ps
//---------------TB-----------------------------
module edge_tb;

  reg clk, async = 0;
  wire rise, fall;

  edge_detect dut
       (.async_sig(async),
        .clk(clk),
        .rise(rise),
        .fall(fall));

initial begin
    clk = 1'b1;
    forever #5 clk = ~clk;
end

// Produce a randomly-changing async signal.
  time delay;

initial
begin
    $dumpfile("edge_tb.vcd");
    $dumpvars(0, edge_tb);
    $monitor("time %3d async %b, fall %b, rise
%b",
          $time, async, rise, fall);
  while ($time < 1000) begin
    // wait for a random number of ns
    delay = $urandom_range(50,100);
    #delay;
    async = ~ async;
  end
  $finish;
end

endmodule
```

génération de signal aléatoire

Le signal en entrée change de m**anière aléatoire** dans le simulateur :

▷ il est **asynchrone** par nature ;

⟹ ses changements de valeur ne sont **pas synchronisés** sur l'horloge du circuit :



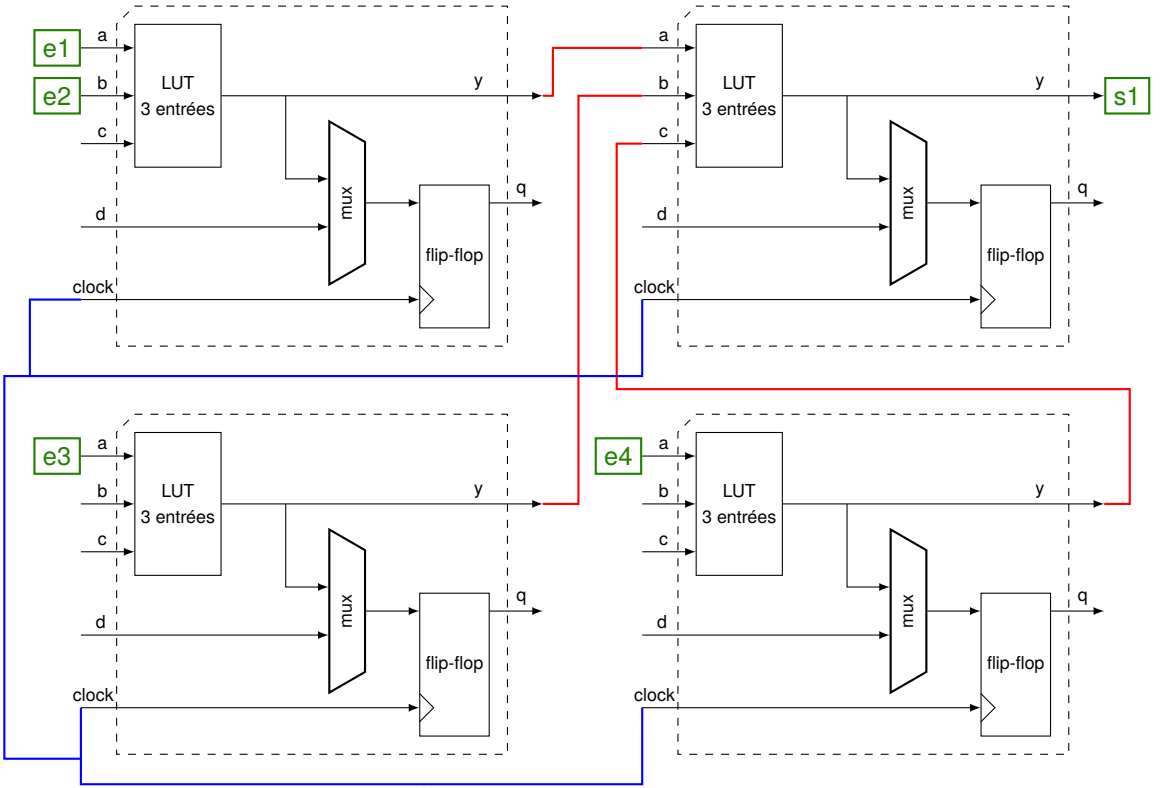
Pour chaque **évévement au bord,** «*edge*», c-à-d lorsque le signal asynchrone change :

▷ une **détection** est réalisée par le circuit ;

⟹ une **impulsion** est créée par le circuit.

Et comment on arrive à un circuit physique ?

- ▷ on réalise les fonctions logiques avec des (LUT) et les mémorisations de bits avec des Flip-Flops :
  - ◇ sélectionner les blocks logiques qui vont les «*héberger*» ⟹ **placement**
- ▷ on route les E/S du circuit ($e1$, $e2$, ..., $s1$, $s2$,...) vers d'autres blocks ou des pins d'E/S ⟹ **routage**

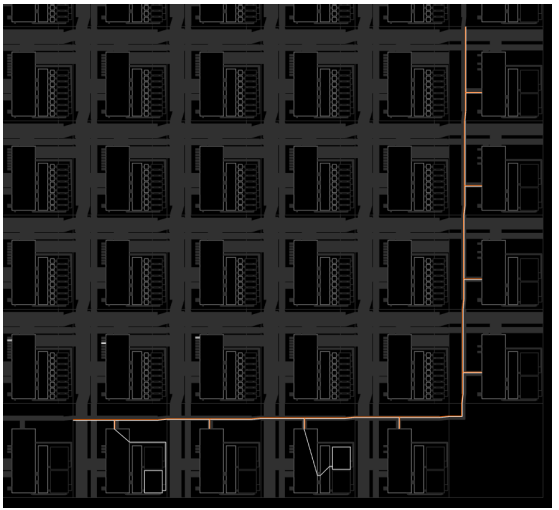


On a sélectionner quatre blocks, router les E/S externes et internes, l'horloge...

On réalise un simple circuit :

1. un bouton est relié à une LED ;
2. la négation de l'état du bouton est reliée à la LED.

1
```
assign led1 = B1;
```



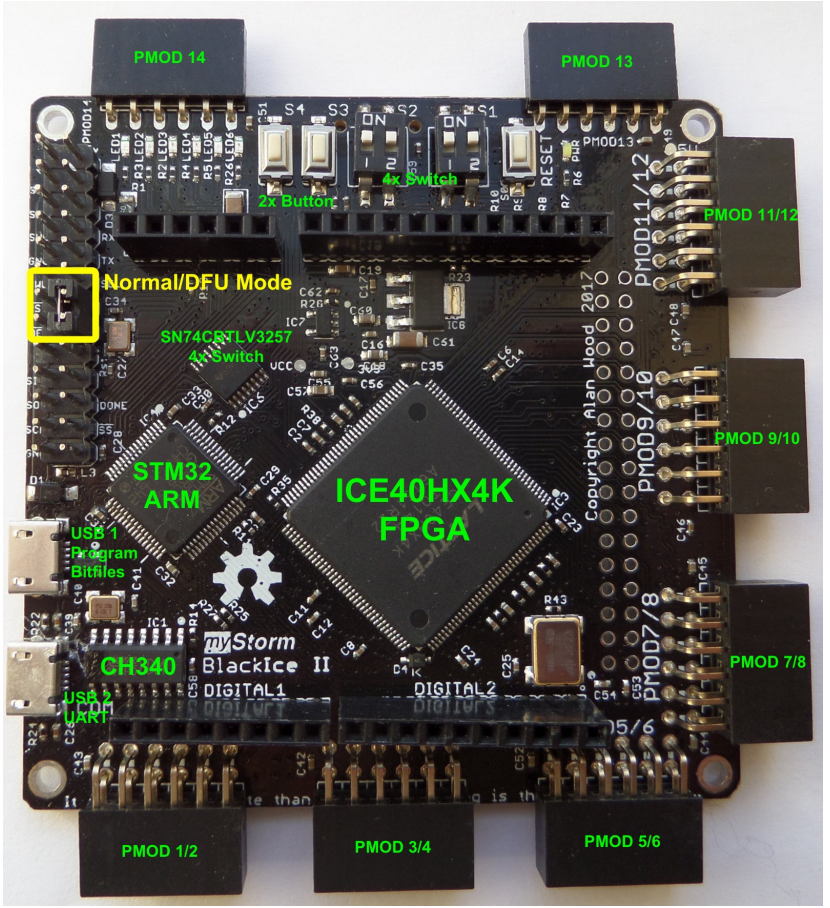
Le routage connecte le «pad» *d'entrée relié au bouton, au pad de sortie relié à la LED.*

2
```
assign led1 = !B1;
```

Le routage connecte le «pad» *d'entrée relié au bouton à un block logique pour faire la négation, puis le résultat est relié au pad de sortie relié à la LED.*

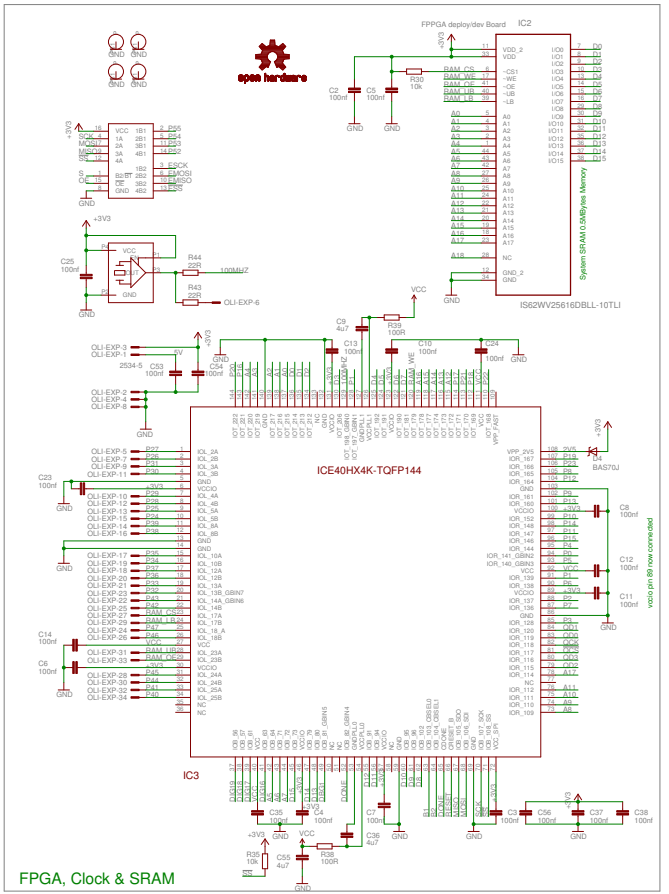
Les représentations données ont été obtenues grâce à l'outil «open source» `nextpnr-ice40`.

La «*board*» utilisée : le «*Black Ice II*»



- ☐ contient un FGPA **Lattice iCE 40HX8k** ;
- ☐ contient un micro contrôleur **ARM STM32**.
- ☐ le composant CH340 assure le lien USB-Serial ;
- ☐ le microcontrôleur ARM STM32 assure la gestion du «*bitstream*» pour le FPGA :
  - ◇ sauvegarde en mémoire flash pour programmation à l'allumage ;
  - ◇ transfert par le port série du bitstream pour programmation à la volée.
- ☐ des **connecteurs** standardisés au format PMODs.

- Lattice Ice40 HX4K TQFP144 FPGA with 56 PIO and 80Kb BRAM & 4Mb (256kx16) SRAM
- STM32L433 ARM Cortex M4 Microcontroller 26 GPIO 256KB Flash and 64KB RAM
- 100Mhz Oscillator (Ice40), 12Mhz crystal (STM32)
- SPI Mux control between Microcontroller, LEDs and RPi header
- SDCard SDIO connections to both Ice40 and STM32L433
- USB 1 - IceBoot for programming Ice40 with synthesised bitfiles
- USB 2 - Serial for monitoring and debugging Ice40 FPGA development
- Dip switches for input codes and/or configurations
- Push Buttons for reading inputs and resetting the board
- 4 x coloured LEDs for FPGA output indicators and fun!
- 1 x Status LED, Programmed LED & a Power LED
- 6 x Double PMODS (8 PIOs each) expansion connectors
- 2 x Single PMODS (4 PIOs each) extension connectors
- Arduino Shield Compatible Headers (plus 4 pin extension)
- RPi Header (26Pin) allows direct integration with all Raspberry Pi variants
- All hardware is Completely OpenSource and fully reusable
- OpenSource Verilog Toolchain - Clifford Wolf's IceStorm
- IceStudio and APIO support for getting started quickly



- Lattice Semiconductor
- FPGA iCE40 HX Family 3520 Cells 40nm Technology 1.2V
- 144-Pin TQFP Tray

Max Frequency	533 MHz
Max Operating Temperature	85 °C
Max Supply Voltage	1.26 V
Memory Size	10 kB
Min Operating Temperature	-40 °C
Min Supply Voltage	1.14 V
Number of Gates	3520
Number of I/Os	107
Number of Logic Blocks (LABs)	440
Number of Logic Elements/Cells	3520
Number of Macrocells	3520
Number of Registers	3520
Operating Supply Voltage	1.2 V
RAM Size	10 kB

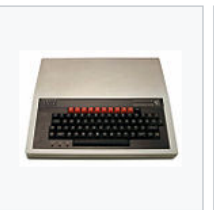
Mais ça marche comment un processeur ?

exemple le processeur 6502

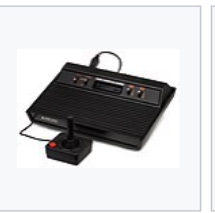
- processeur développé par Chuck Peddle pour la société MOS Technology ;
- introduit en 1975 ;
- très populaire :

Apple IIe

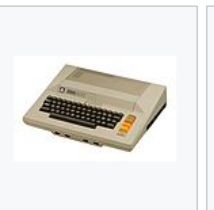
Commodore PET



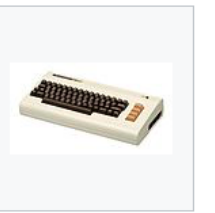
BBC Micro



Atari 2600



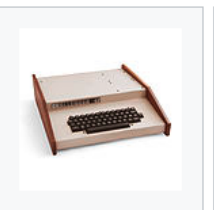
Atari 800



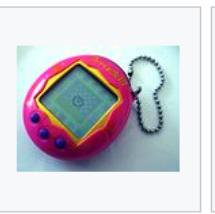
Commodore VIC-20

Commodore 64

Family Computer



Ohio Scientific Challenger 4P

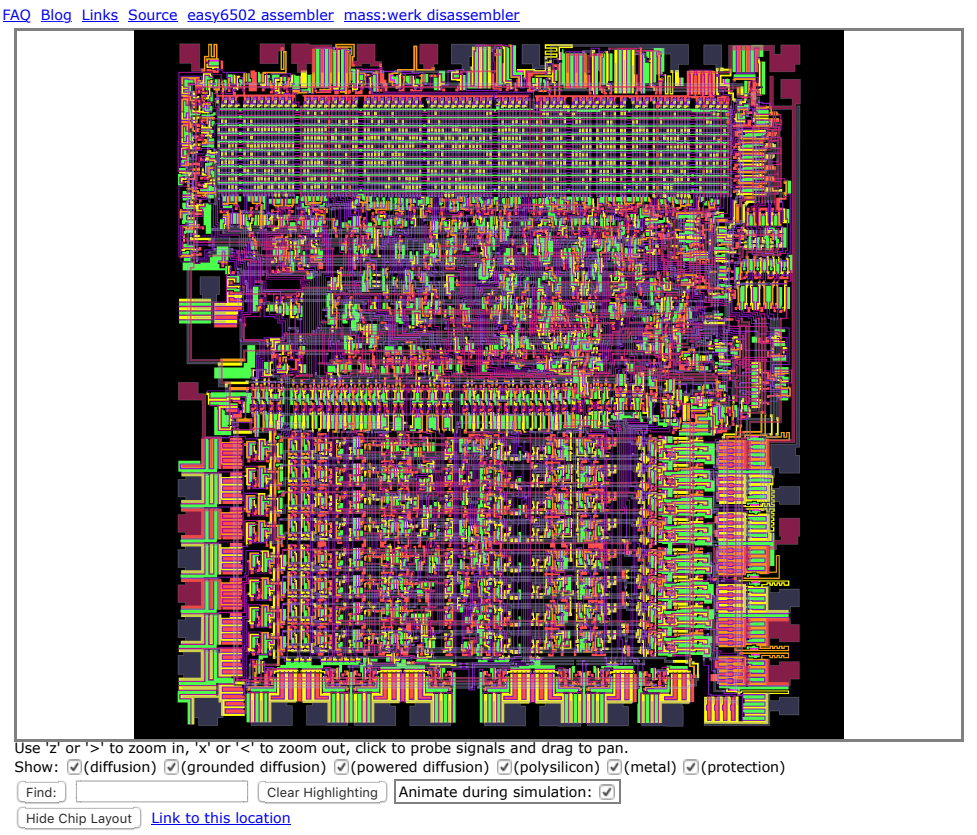


Tamagotchi digital pet[53]

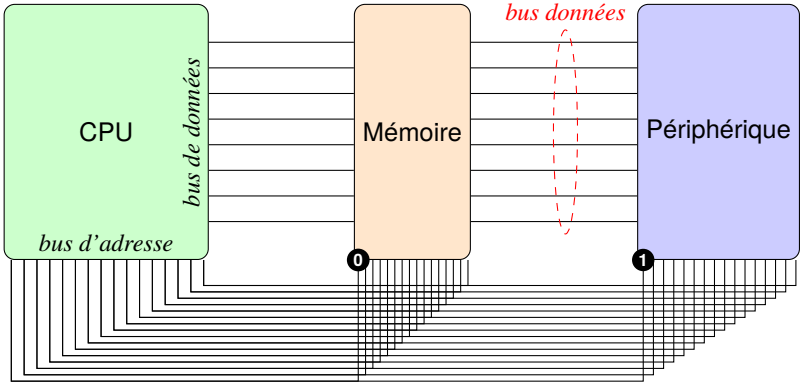
Atari Lynx

- toujours en vente et utilisé dans les **systèmes embarqués** ;
- processeur 8bits, avec un bus d'adresse sur 16bits et «*little-endian*», cadencé de 1 à 2 MHz

Le processeur 6502 : représentation visuelle

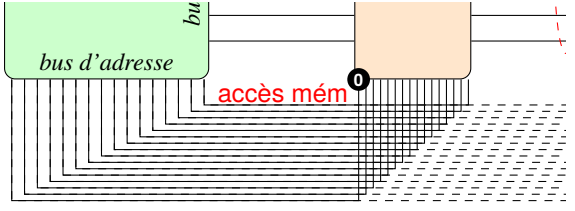


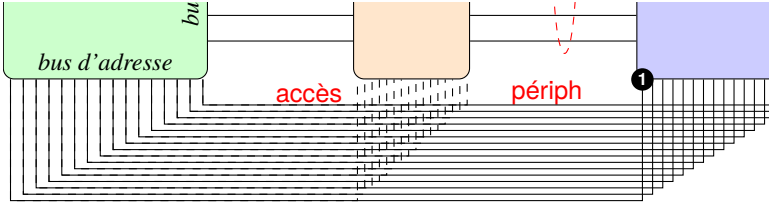
http://www.visual6502.org/JSSim/expert.html



Pour accéder à :

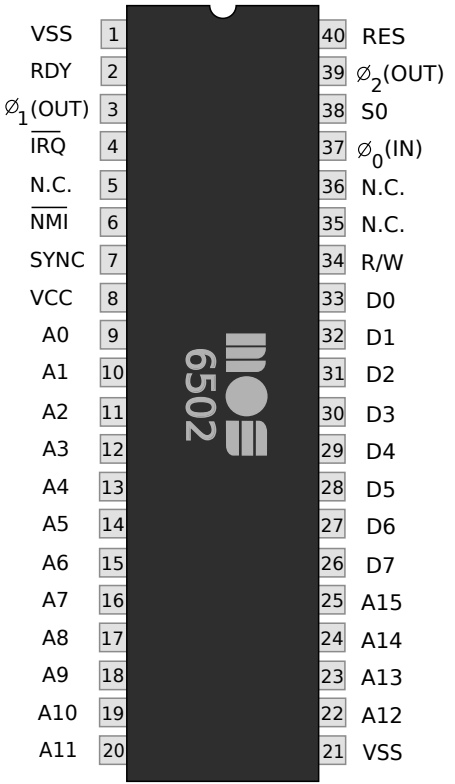
- la **mémoire** :
  - on va de l'adresse (**0**000 0000 0000 0000)$_2$ à l'adresse (**0**111 1111 1111 1111)$_2$;
  - ce qui donne de l'adresse (00 00)$_{16}$ à l'adresse (7F FF)$_{16}$;





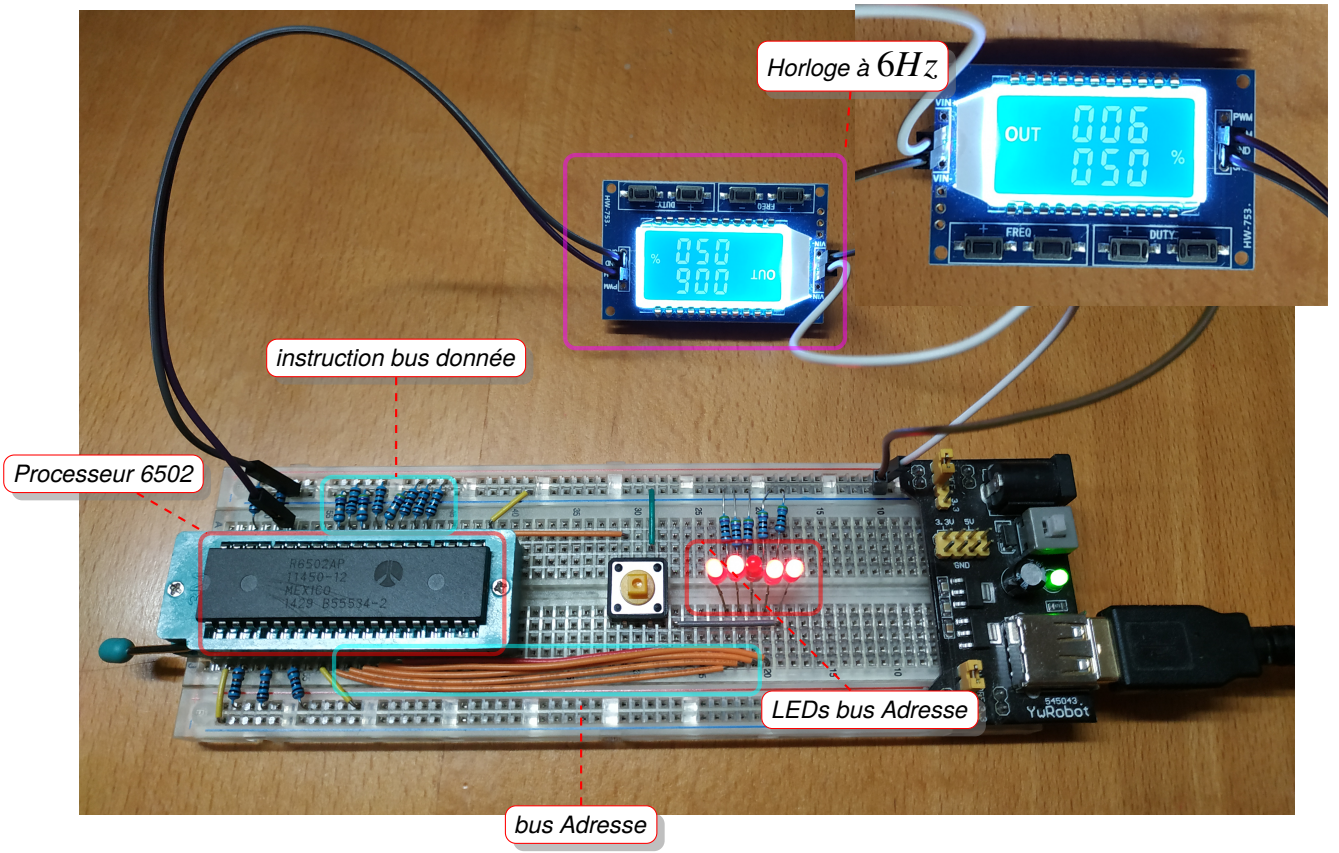
- aux **périphériques** :
  - on va de l'adresse (**1**000 0000 0000 0000)$_2$ à l'adresse (**1**111 1111 1111 1111)$_2$;
  - ce qui donne de l'adresse (80 00)$_{16}$ à l'adresse (FF FF)$_{16}$;

Le 6502 : connexion avec l'extérieur

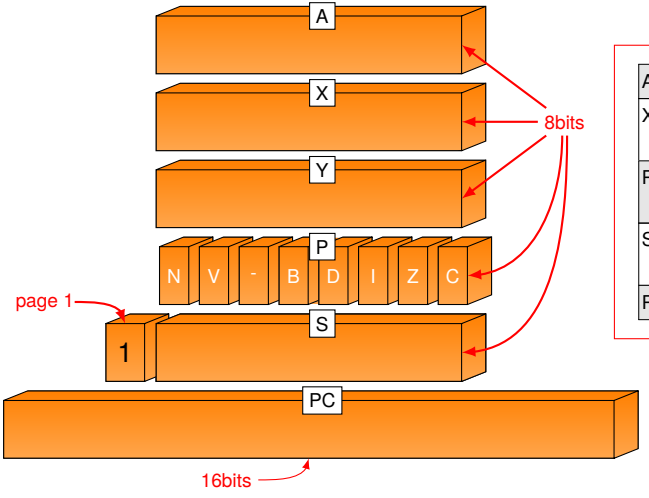


- ▷ Accès à la mémoire :
  - ☐ A_0, ..., A_{15} : 16 bits d'adresse ;
  - ☐ D_0, ..., D_7 : 8 bits de données ;
  - ☐ R/W : indique si c'est une opération de lecture ou d'écriture ;
- ▷ Interactions avec l'extérieur :
  - ☐ $Sync$: signal d'horloge : ryhtme le travail du processeur ;
  - ☐ NMI : «*Non Maskable Interruption*» : signal d'interruption ;
  - ☐ RES : «*reset*», réinitialise l'état du processeur et, si maintenue, le bloque ;

6502 sur «*breadboard*» : un processeur, une horloge et deux bus



Et la programmation d'un processeur ?

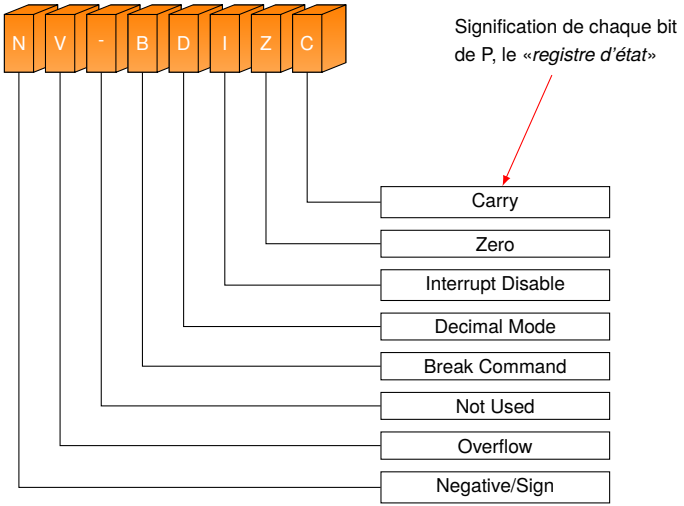


Registres

A	«*Accumulator*»	stockage depuis ou vers l'ALU
X & Y	«*Index register*»	utilisés dans certaines instructions pour calculer une adresse par décalage : `adresse+X`
P	«*Processor status register*»	chaque bit indique 1 état suite à l'exécution de l'instruction : nombre positif, nul, etc.
S	«*Stack pointer*»	contient l'adresse du dernier octet dans la pile le bit de préfixe à 1 place la pile sur la seconde page
PC	«*Program counter*»	indique l'adresse de la prochaine instruction à exécuter

Explications du registre d'état

Carry	indique un bit de retenu après opération de l'ALU (9ème bit...)
Zero	la valeur de X, Y ou A est devenue zéro
oVerflow	dépassement de capacité lors d'opération sur des nombres signés
Negative	vrai si le bit de rang 7 est à 1



mode	opérande
immédiat	la donnée
absolu	n'importe quelle adresse
page zéro	un octet correspondant au second octet d'adresse, le premier est fixé à zéro
indexé X	adresse+registre X
indexé Y	adresse+registre Y
implicite	pas d'opérande
relatif	un octet relatif en complément à deux, de -128 à 127

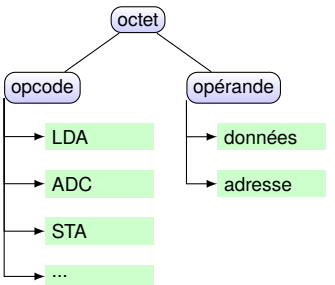
Chaque instruction est **codée sur un octet** en fonction du mode choisi.

Exemple : l'instruction ADC donne l'octet `69` *si la valeur à additionner est donnée en paramètre (mode immédiat).*

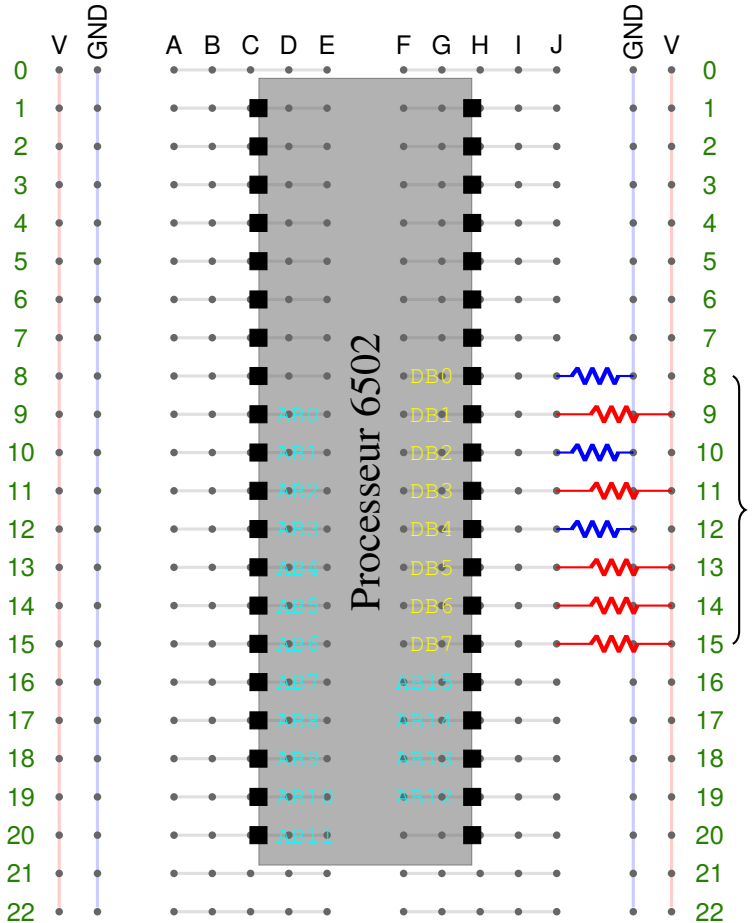
`69 01` *signifie additionner la valeur 1 dans l'accumulateur.*

Certaines instructions ***modifient le registre d'état P*** *: Exemple pour faire un saut sur la condition que X soit égal à zéro :*
`CPX $0` *; compare la valeur du registre X avec 0 ⟹ positionne le bit Z à nul si les deux valeurs sont identiques (on fait une soustraction)*
`BEQ 0A` *; test la valeur du bit Z : 1 donne vrai et 0 donne faux*

Un octet en mémoire peut être :



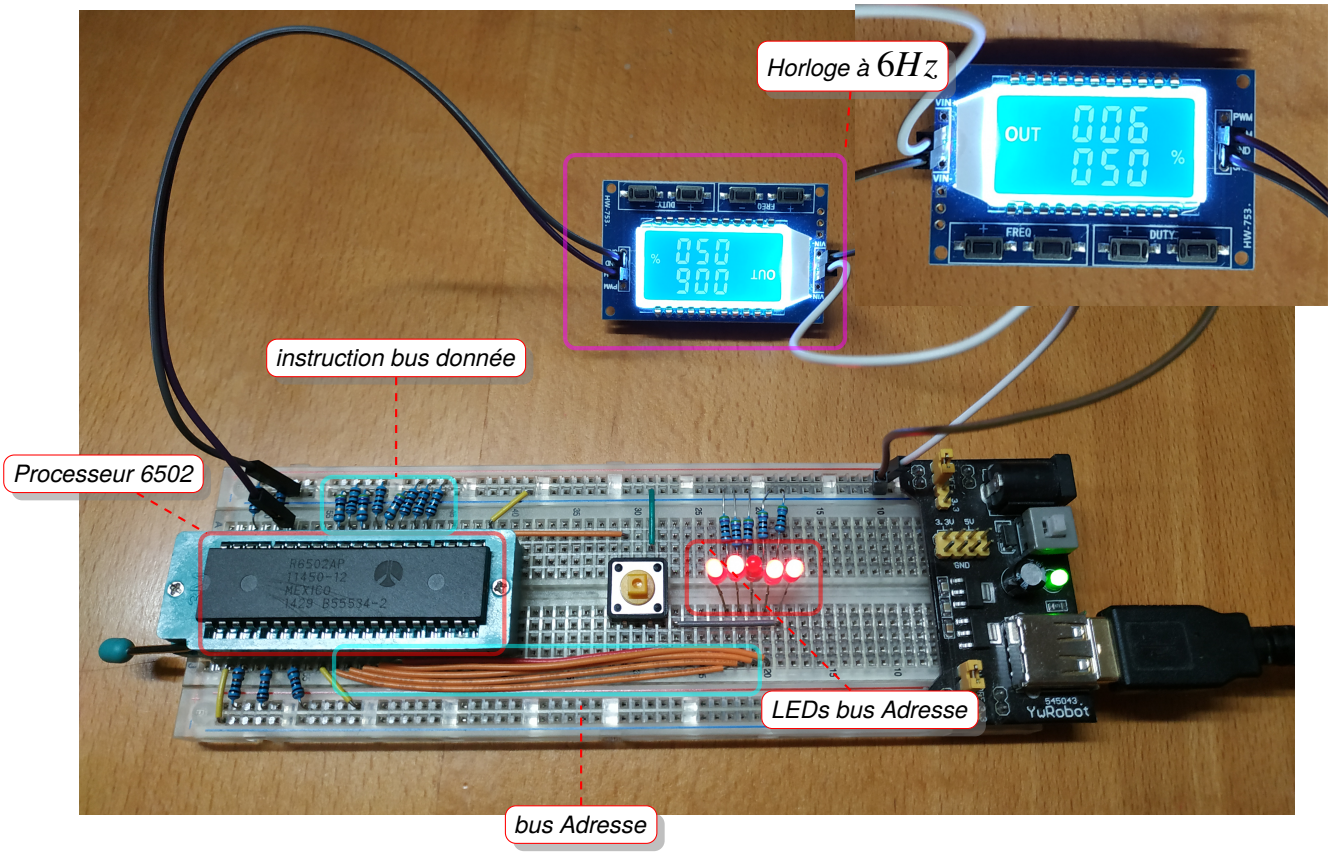
Ins	description	mode adressage						
		immédiat	absolu	page zéro	indexé X	indexé Y	implicite	relatif
ADC	ajoute un octet avec le bit de retenu dans l'accumulateur	69	6D	65	7D	79		
BEQ	«*Branch if EQual*», saut vers une adresse si vrai							F0
BNE	«*Branch if Not Equal*», saut vers une adresse si faux							D0
CPX	compare avec le registre X	E0	EC	E4				
INX	Incrémente la valeur dans le registre X						E8	
INY	Incrémente la valeur dans le registre Y						C8	
JMP	«*JuMP*», saut		4C					
JSR	«*Jump to SubRoutine*», saut vers un sous-programme		20					
LDA	charge un octet dans le registre A	A9	AD	A5	BD	B9		
LDX	charge un octet dans le registre X	A2	AE	A6		BE		
LDY	charge un octet dans le registre Y	A0	AC	A4	BC			
RTS	«*ReTurn from Subroutine*», retour d'un sous-programme						60	
STA	stocke l'accumulateur à une adresse donnée		8D	85	9D	99		
NOP	ne fait rien						EA	



⟹11101010 de DB7 à DB0
ce qui donne en hexa $\{1110\}_2\{1010\}_2 = \{E\}_{16}\{A\}_{16}$

EA en instruction 6052 ⟹ NOP pour «*No OPeration*»
Ce qui veut dire que le processeur ne fait rien
Il passe seulement à l'instruction suivante
⟹il incrémente le CO, «*Compteur ordinal*», ou «*instruction counter*»
⟹l'adresse est incrémentée sur le bus d'adresse AB0 à AB15

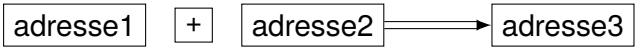
6502 sur «*breadboard*» : un processeur, une horloge et deux bus



Et pour des programmes plus gros ?

Additionner deux nombres en assembleur 6502

Sur 8bits



Le programme assembleur :

```
LDA adresse1    ; charge le nombre stocké à l'adresse 1 dans l'accumulateur
ADC adresse2    ; additionne le nombre stocké à l'adresse 2 à l'accumulateur
STA adresse3    ; stocke le contenu de l'accumulateur à l'adresse 3
RTS             ; retourne
```

Les mnémoniques :

```
AD adresse1
6D adresse2
8D adresse3
60
```

Sur 16bits

Le premier nombre sur deux octets | W | W1 |

Le second nombre sur deux octets | X | X1 |

Attention : on est en «*Little Endian*»,
c-à-d avec inversion des octets de la valeur sur 16bits.

		octets		
		1er	2nd	
Premier nombre	307	51	1	car 307=1*256+51
Second nombre	764	252	2	car 764=2*256+252

Le programme assembleur :

```
CLC
LDA adresse W
ADC adresse X
STA adresse Y
LDA adresse W1
ADC adresse X1
STA adresse Y1
LDA #&0
ADC #&0
STA adresse Z
RTS
```

Les mnémoniques :

```
18
AD adresse W
6D adresse X
8D adresse Y
AD adresse W1
6D adresse X1
8D adresse Y1
A9 00
69 00
8D adresse Z
60
```

$\Longrightarrow$ on utilise le bit de retenu...

Application d'un xor d'un texte avec un mot de passe

Le programme calcule $saisie_i \oplus mdp_i$ pour chaque caractère i de $saisie$ et de mdp.

```
 1 define sortie $200 ; on définie l'adresse de sortie à 0200
 2
 3 LDA saisie   ; on lit la taille de la chaîne saisie
 4 STA sortie   ; on la reporte dans la chaîne de sortie
 5 ADC #$1      ; on incrémente la valeur pour la comparaison utilisée pour arrêter la boucle
 6 STA $0      ; on la stocke dans la page zéro
 7 LDA mdp      ; on lit la taille de la chaîne mdp
 8 ADC #$1      ; on incrémente la valeur utilisée pour réinitialiser l'utilisation du mdp
 9 STA $1      ; on la stocke dans la page zéro
10
11 LDX #$1      ; on charge la valeur 1 dans le registre X
12 LDY #$1      ; on charge la valeur 1 dans le registre Y
13
14 boucle:      ; on définit une étiquette
15    LDA saisie,X   ; on charge dans l'accumulateur la valeur à l'adresse saisie+X
16    EOR mdp,Y      ; on réalise un xor entre le registre A et la valeur à l'adresse mdp+Y
17    STA sortie,X   ; on stocke le résultat à l'adresse sortie+X
18    INX            ; on incrémente la valeur contenu dans le registre X
19    CPX $0         ; on compare la valeur de la taille de la chaîne saisie
20    BEQ fin         ; si elle est identique, on a fini et on mets l'adresse fin dans le registre PC
21    INY            ; on incrémente la valeur contenue dans le registre Y
22    CPY $1         ; on compare avec la valeur de la taille de la chaîne mdp
23    BNE boucle      ; si elle n'est pas égale on recommence la boucle en sautant à l'adresse boucle
24    LDY #$1         ; sinon on réinitialise le registre Y à 1
25    JMP boucle      ; et on effectue un saut à l'adresse boucle
26 fin:         ; étiquette
27    BRK            ; instruction d'arrêt
28
29 saisie:
30    dcb 5,$68,$65,$6c,$6c,$6f ;hello
31 mdp:
32    dcb $9,$74,$6f,$70,$73,$65,$63,$72,$65,$74;topsecret
```

Utilisation du désassembleur

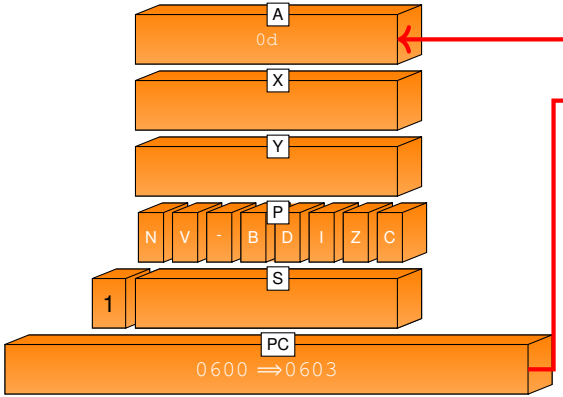
```
Address  Hexdump   Dissassembly
-------------------------------
$0600    ad 2e 06  LDA $062e
$0603    8d 00 02  STA $0200
$0606    69 01     ADC #$01
$0608    85 00     STA $00
$060a    ad 3c 06  LDA $063c
$060d    69 01     ADC #$01
$060f    85 01     STA $01
$0611    a2 01     LDX #$01
$0613    a0 01     LDY #$01
$0615    bd 2e 06  LDA $062e,X
$0618    59 3c 06  EOR $063c,Y
$061b    9d 00 02  STA $0200,X
$061e    e8        INX
$061f    e4 00     CPX $00
$0621    f0 0a     BEQ $062d
$0623    c8        INY
$0624    c4 01     CPY $01
$0626    d0 ed     BNE $0615
$0628    a0 01     LDY #$01
$062a    4c 15 06  JMP $0615
$062d    00        BRK
$062e    0d 68 65  ORA $6568
$0631    6c 6c 6f  JMP ($6f6c)
$0634    20 62 6f  JSR $6f62
$0637    6e 6a 6f  ROR $6f6a
$063a    75 42     ADC $42,X
$063c    09 74     ORA #$74
$063e    6f        ???
$063f    70 73     BVS $06b4
$0641    65 63     ADC $63
$0643    72        ???
$0644    65 74     ADC $74
```

```
0600: ad 2e 06 8d 00 02 69 01 85 00 ad 3c 06 69 01 85
0610: 01 a2 01 a0 01 bd 2e 06 59 3c 06 9d 00 02 e8 e4
0620: 00 f0 0a c8 c4 01 d0 ed a0 01 4c 15 06 00 0d 68
0630: 65 6c 6c 6f 20 62 6f 6e 6a 6f 75 72 09 74 6f 70
0640: 73 65 63 72 65 74
```

On note que :

`$062e`	adresse de la chaîne saisie
`$063c`	adresse de la chaîne mdp
`$062d`	adresse de l'instruction `brk`
`$062e`	le désassembleur trouve des instructions dans le contenu de la chaîne saisie $\Longrightarrow$ Interprétation automatique erronée
`$063e` `$0643`	Interprétation automatique impossible, il n'y a pas d'instruction reconnue

Exécution du code : évolution des contenus des registres



Contenu de la mémoire :

```
Address  Hexdump   Dissassembly
-------------------------------
$0600    ad 2e 06  LDA $062e
$0603    8d 00 02  STA $0200
$0606    69 01     ADC #$01
$0608    85 00     STA $00
$060a    ad 3c 06  LDA $063c
$060d    69 01     ADC #$01
$060f    85 01     STA $01
$0611    a2 01     LDX #$01
$0613    a0 01     LDY #$01
$0615    bd 2e 06  LDA $062e,X
$0618    59 3c 06  EOR $063c,Y
$061b    9d 00 02  STA $0200,X
$061e    e8        INX
$061f    e4 00     CPX $00
$0621    f0 0a     BEQ $062d
$0623    c8        INY
$0624    c4 01     CPY $01
$0626    d0 ed     BNE $0615
$0628    a0 01     LDY #$01
$062a    4c 15 06  JMP $0615
$062d    00        BRK
$062e    0d 68 65  ORA $6568
$0631    6c 6c 6f  JMP ($6f6c)
$0634    20 62 6f  JSR $6f62
$0637    6e 6a 6f  ROR $6f6a
$063a    75 42     ADC $42,X
$063c    09 74     ORA #$74
$063e    6f        ???
$063f    70 73     BVS $06b4
$0641    65 63     ADC $63
$0643    72        ???
$0644    65 74     ADC $74
```

▷ Au démarrage : le registre PC, «*Program Counter*» contient l'adresse `0600` ;

⟹❶ le processeur va chercher l'octet contenu à cette adresse comme prochaine instruction à exécuter : c'est la valeur `ad` qui indique une instruction de chargement du registre A avec le contenu de l'adresse fournie en argument ;

⟹le processeur va chercher les deux octets suivants pour obtenir cette adresse : `2e` et `06` qu'il inverse et obtient au final l'adresse `062e`

⟹l'adresse `062e` contient l'octet `0d` qui va être chargé dans le registre A❷ ;

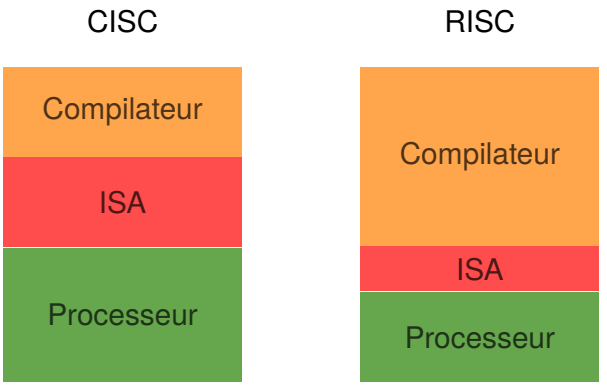
⟹le registre PC passe alors à `0603`❸ pour exécuter la prochaine instruction.

L'exécution d'une instruction et l'accès mémoire prends plusieurs cycles d'horloge.

Et les processeurs plus modernes ?

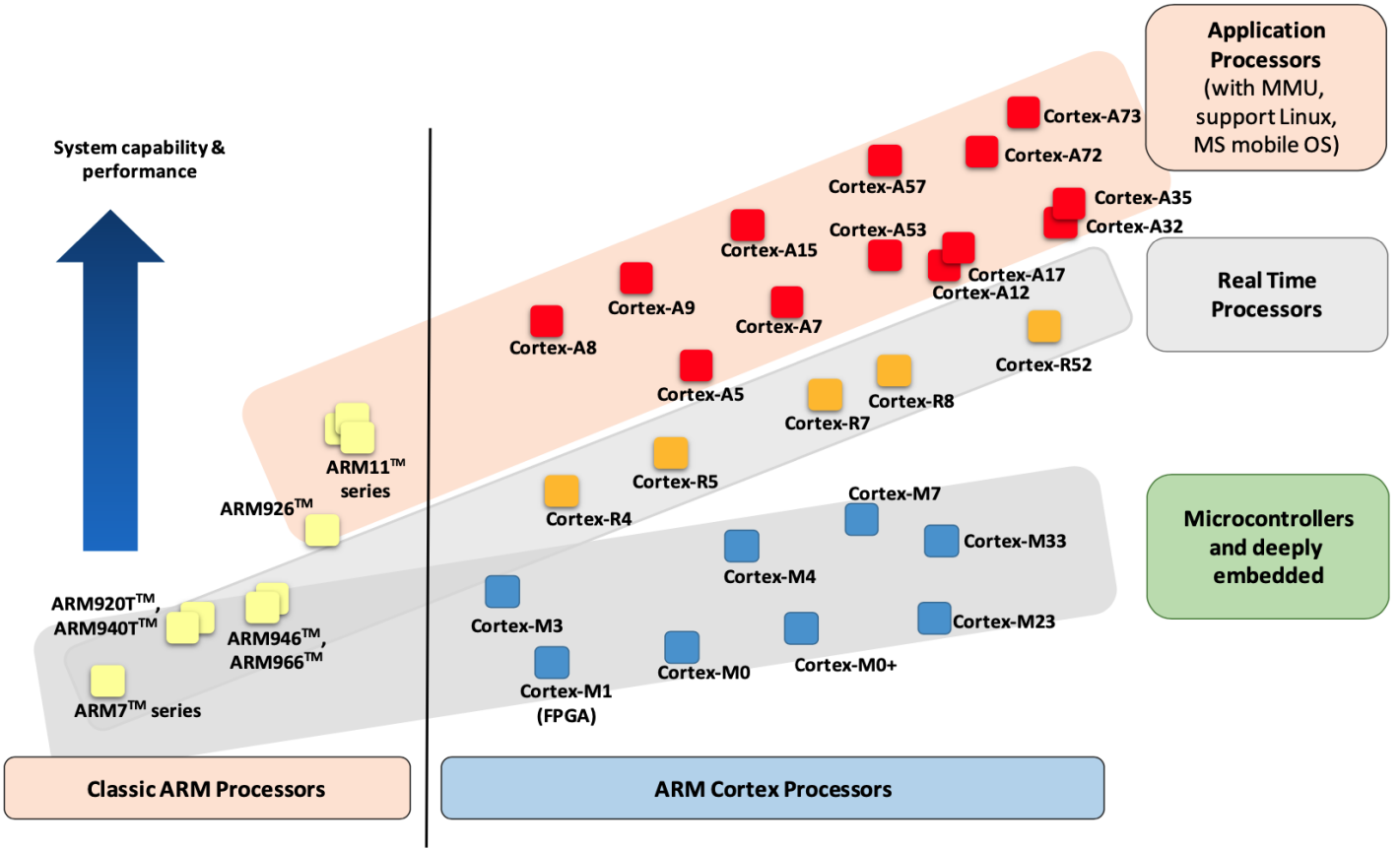
Qu'est-ce que «*RISC-V*» ?

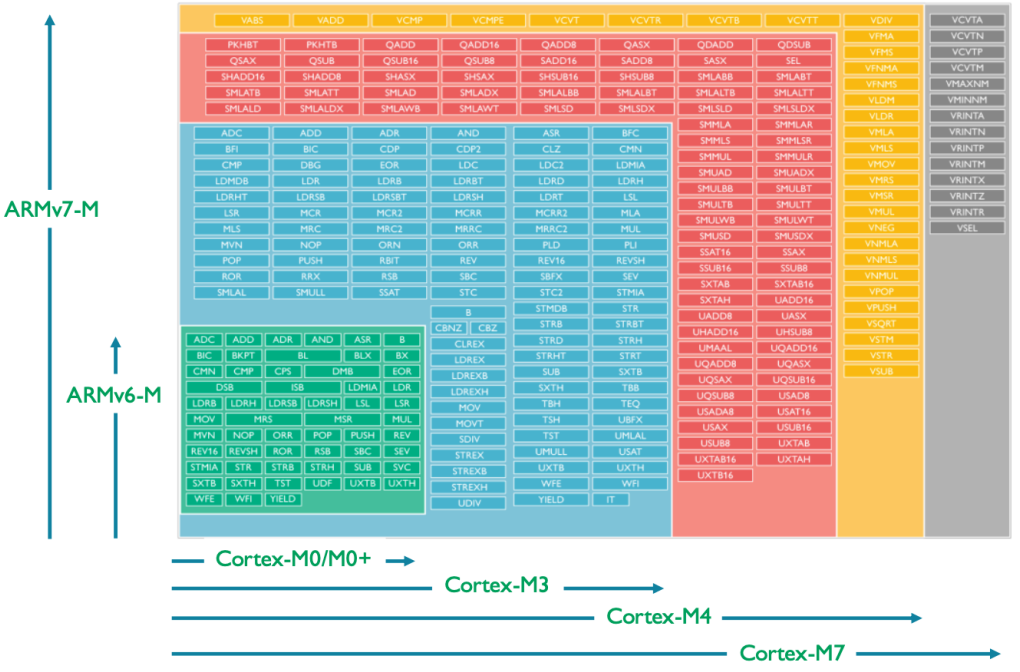
- «*Open Standard Instruction Set*», ISA ;
- basé sur les principes RISC, «*Reduced Instruction Set Computer*» ;



- licence «*Open Source*» sans royalties, mais des extensions payantes... ;
- supporté par :
  - différentes entreprises au niveau de l'offre hardware : sifive ;
  - différents OS : Linux, FreeRTOS ;
  - différents «*toolchains*» : compilateur, linkeur, constructeur de firmware ;
  - sous formes de différentes «*IPs*» pour FPGA, «*Field Programmable Gate Array*».

ARM Cortex et ISA





Floating Point

DSP (SIMD, fast MAC)

Advanced data processing
bit field manipulations

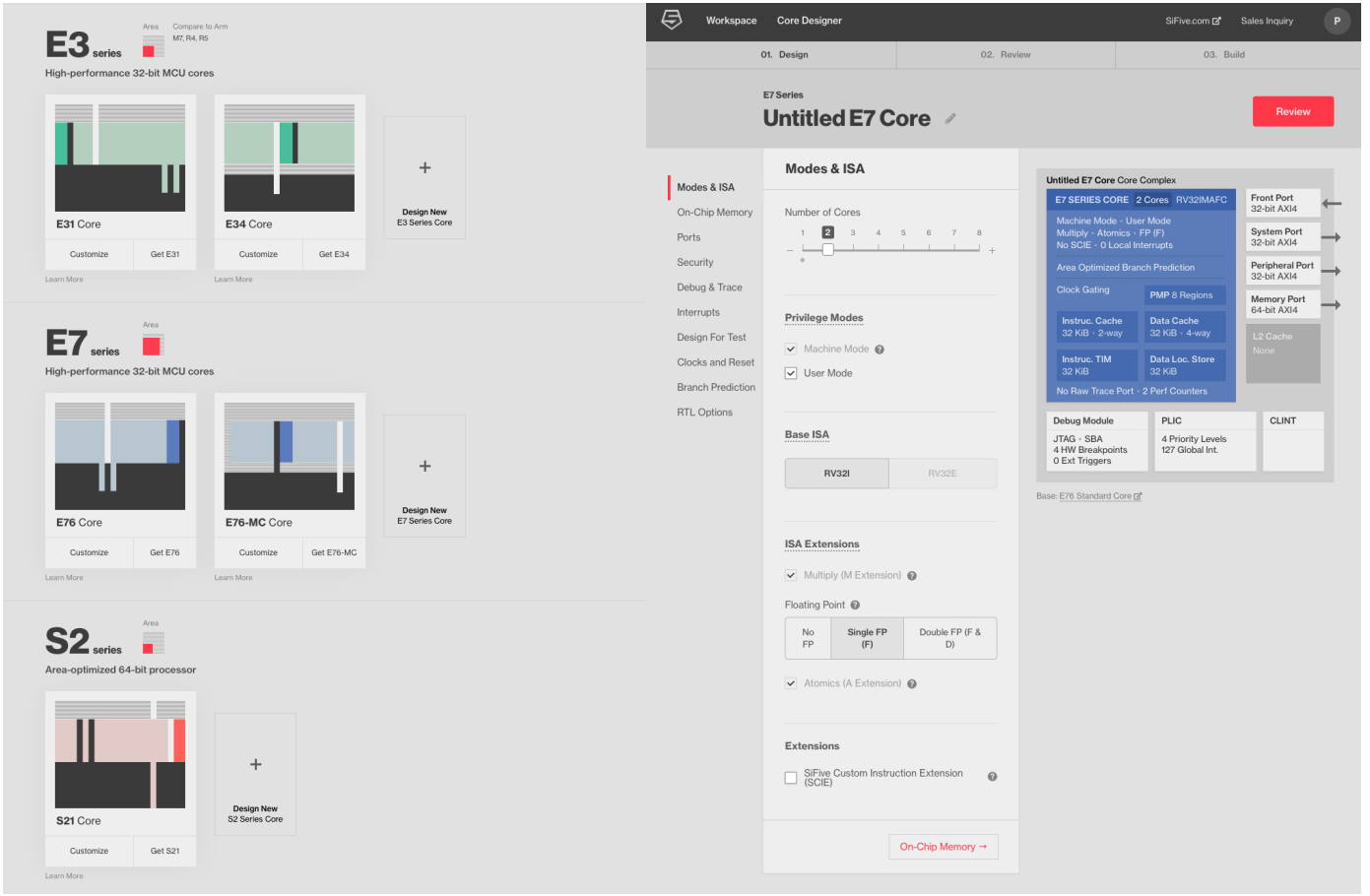
General data processing
I/O control tasks

Name	Description	Version	Status	Instruction count
	Base			
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	"Base Integer Instruction Set 32-bit"	2.1	Ratified	49
RV32E	"Base Integer Instruction Set (embedded) 32-bit 16 registers"	1.9	Open	49
RV64I	"Base Integer Instruction Set 64-bit"	2.1	Ratified	14
RV128I	"Base Integer Instruction Set 128-bit"	1.7	Open	14
	Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified	8
A	Standard Extension for Atomic Instructions	2.1	Ratified	11
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified	25
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified	25
Zicsr	Control and Status Register (CSR)	2.0	Ratified	
Zifencei	Instruction-Fetch Fence	2.0	Ratified	
G	*"Shorthand for the IMAFDZicsr Zifencei base and extensions intended to represent a standard general-purpose ISA"*	N/A	N/A	
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified	27
L	Standard Extension for Decimal Floating-Point	0.0	Open	
C	Standard Extension for Compressed Instructions	2.0	Ratified	36
B	Standard Extension for Bit Manipulation	0.93	Open	42
J	Standard Extension for Dynamically Translated Languages	0.0	Open	
T	Standard Extension for Transactional Memory	0.0	Open	
P	Standard Extension for Packed-SIMD Instructions	0.2	Open	
V	Standard Extension for Vector Operations	0.10	Open	186
N	Standard Extension for User-Level Interrupts	1.1	Open	3
H	Standard Extension for Hypervisor	0.4	Open	2
Zam	Misaligned Atomics	0.1	Open	
Ztso	Total Store Ordering	0.1	Frozen	

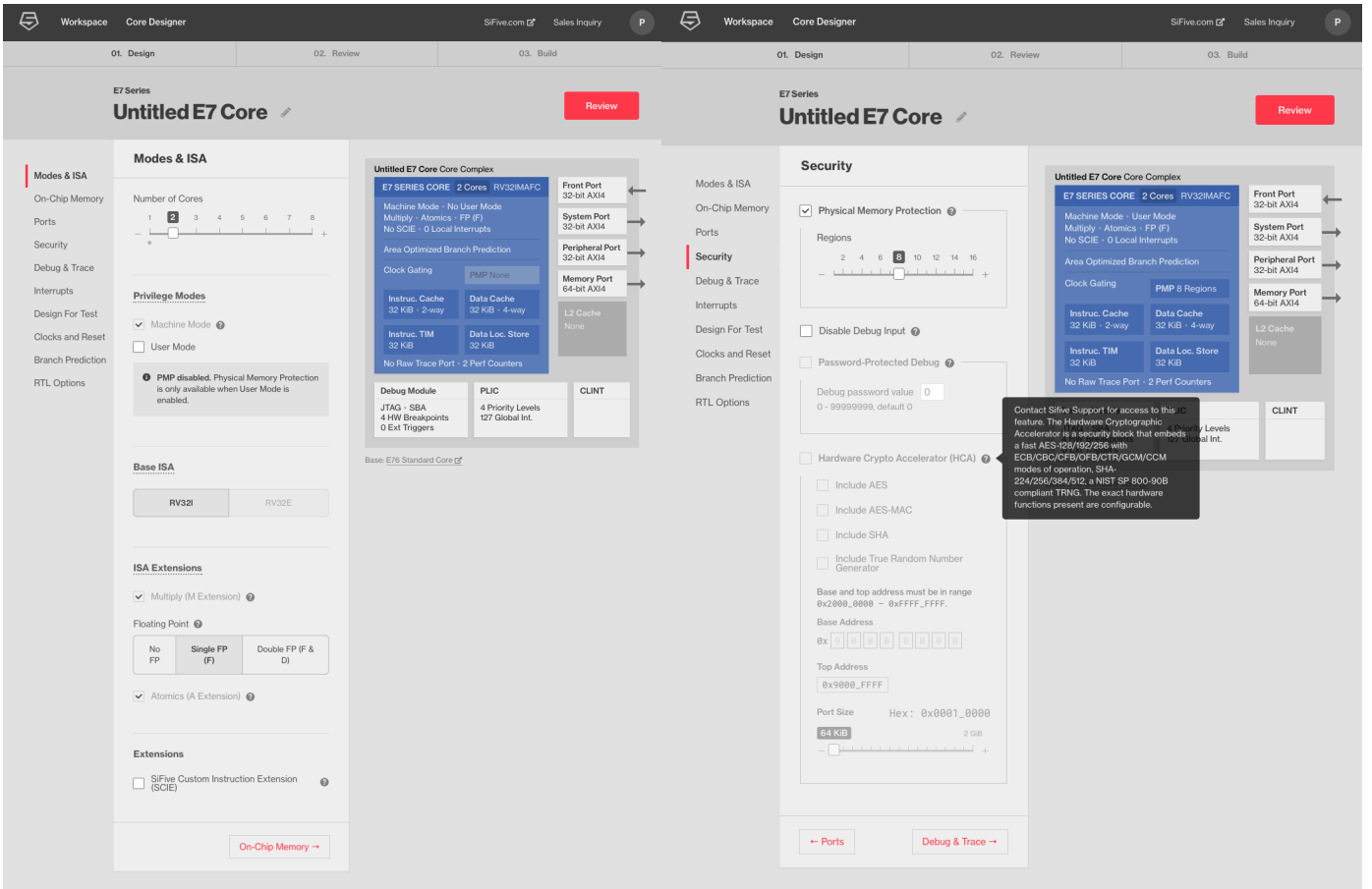
Floating point operation

Le choix des extensions peut amener à des coûts supplémentaires...

La configuration pour la production d'un processeur sur *scs.sifive.com*



La configuration pour la production d'un processeur sur *scs.sifive.com*



Les registres

- ceux disponibles dans l'ISA, «*Instruction Set Architecture*» :
  - le jeu complet de registres disponibles (ici 32) ;
- ceux utilisés pour l'ABI, «*Application Binary Interface*» :
  - **renommés** pour plus de facilité ;
  - liés aux système d'exploitation ;
  - associés aux passages de paramètres de fonction, aux appels systèmes, aux fast IRQs ;

Par exemple :

- *le registre r_1 de l'ISA est aussi appelé ra dans l'ABI ;*
- *les registres r_{10}, r_{11} de l'ISA sont appelés a_0, a_1 dans l'ABI et servent aux deux premiers arguments passés à une fonction.*

L'ISA RiscV en une page

RISC-V Instruction-Set

Erik Engheim <erik.engheim@ma.com>

Arithmetic Operation

Mnemonic	Instruction	Type	Description
`ADD rd, rs1, rs2`	Add	R	`rd ← rs1 + rs2`
`SUB rd, rs1, rs2`	Subtract	R	`rd ← rs1 - rs2`
`ADDI rd, rs1, imm12`	Add immediate	I	`rd ← rs1 + imm12`
`SLT rd, rs1, rs2`	Set less than	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTI rd, rs1, imm12`	Set less than immediate	I	`rd ← rs1 < imm12 ? 1 : 0`
`SLTU rd, rs1, rs2`	Set less than unsigned	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTIU rd, rs1, imm12`	Set less than immediate unsigned	I	`rd ← rs1 < imm12 ? 1 : 0`
`LUI rd, imm20`	Load upper immediate	U	`rd ← imm20 << 12`
`AUIP rd, imm20`	Add upper immediate to PC	U	`rd ← PC + imm20 << 12`

Load / Store Operations

Mnemonic	Instruction	Type	Description
`LD rd, imm12(rs1)`	Load doubleword	I	`rd ← mem[rs1 + imm12]`
`LW rd, imm12(rs1)`	Load word	I	`rd ← mem[rs1 + imm12]`
`LH rd, imm12(rs1)`	Load halfword	I	`rd ← mem[rs1 + imm12]`
`LB rd, imm12(rs1)`	Load byte	I	`rd ← mem[rs1 + imm12]`
`LWU rd, imm12(rs1)`	Load word unsigned	I	`rd ← mem[rs1 + imm12]`
`LHU rd, imm12(rs1)`	Load halfword unsigned	I	`rd ← mem[rs1 + imm12]`
`LBU rd, imm12(rs1)`	Load byte unsigned	I	`rd ← mem[rs1 + imm12]`
`SD rs2, imm12(rs1)`	Store doubleword	S	`rs2 → mem[rs1 + imm12]`
`SW rs2, imm12(rs1)`	Store word	S	`rs2(31:0) → mem[rs1 + imm12]`
`SH rs2, imm12(rs1)`	Store halfword	S	`rs2(15:0) → mem[rs1 + imm12]`
`SB rs2, imm12(rs1)`	Store byte	S	`rs2(7:0) → mem[rs1 + imm12]`

Pseudo Instructions

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

Logical Operations

Mnemonic	Instruction	Type	Description
`AND rd, rs1, rs2`	AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	OR immediate	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1 ^ imm12`
`SLL rd, rs1, rs2`	Shift left logical	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Shift right logical	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1 >> shamt`

Branching

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 = rs2` `pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 ≠ rs2` `pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 ≥ rs2` `pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← PC + 4` `PC ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← PC + 4` `PC ← rs1 + imm12`

Register File

r0	r1	r2	r3
r4	r5	r6	r7
r8	r9	r10	r11
r12	r13	r14	r15
r16	r17	r18	r19
r20	r21	r22	r23
r24	r25	r26	r27
r28	r29	r30	r31

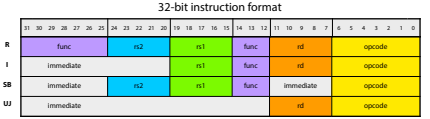
Register Aliases

zero	ra	sp	gp
tp	t0	t1	t2
s0/fp	s1	a0	a1
a2	a3	a4	a5
a6	a7	s2	s3
s4	s5	s6	s7
s8	s9	s10	s11
t3	t4	t5	t6

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

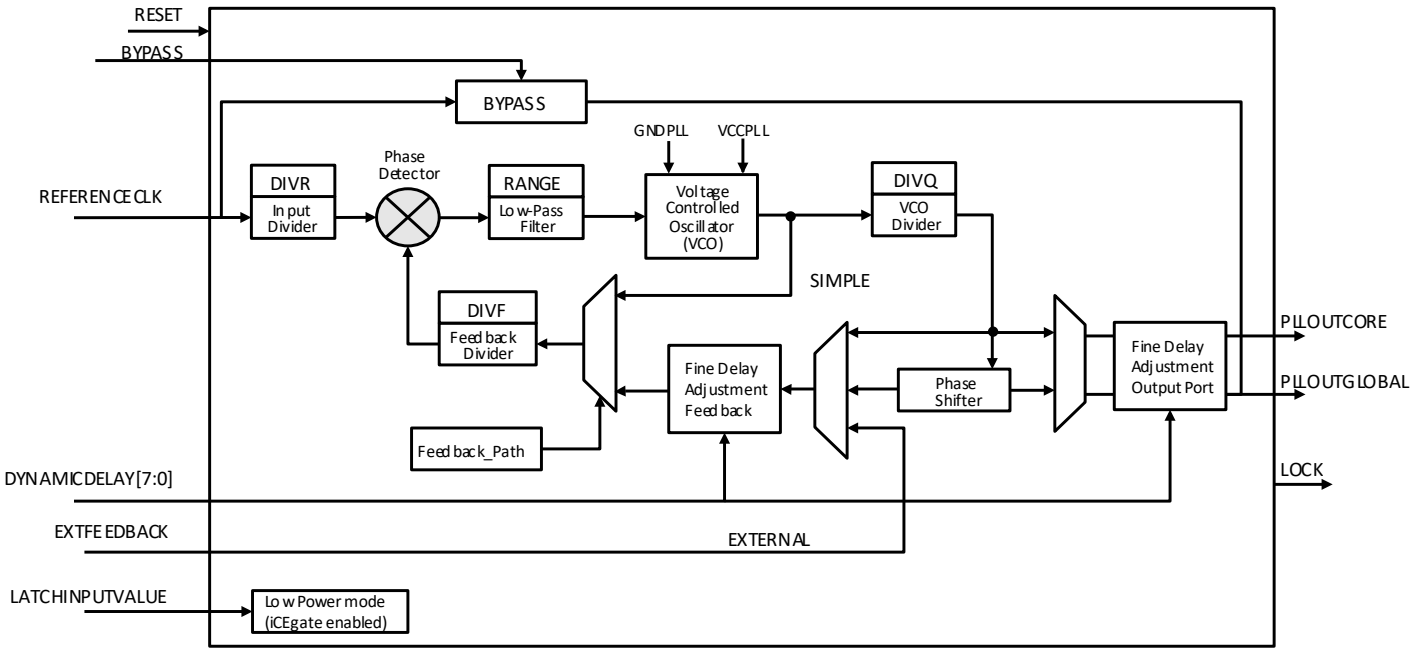
t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - 17 - Function arguments
a0 - a1 - Return value(s)

32-bit instruction format



http://blog.translusion.com/images/posts/RISC-V-cheatsheet-RV32I-4-3.pdf

Et pour notre «*soft core*» ?

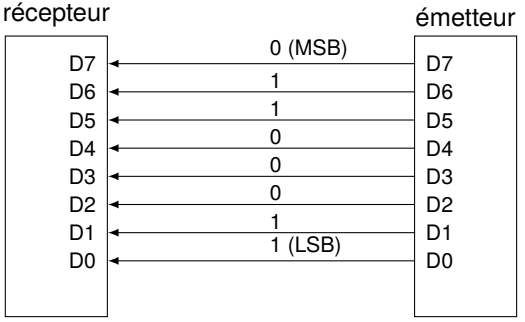

RESET
BYPASS
BYPASS
GNDPLL
VCCPLL
Phase Detector
REFERENCECLK
DIVR
Input Divider
RANGE
Low-Pass Filter
Voltage Controlled Oscillator (VCO)
DIVQ
VCO Divider
SIMPLE
DIVF
Feedback Divider
Fine Delay Adjustment Feedback
Phase Shifter
Fine Delay Adjustment Output Port
PLLOUTCORE
PLLOUTGLOBAL
Feedback_Path
LOCK
DYNAMICDELAY[7:0]
EXTFEEDBACK
EXTERNAL
LATCHINPUTVALUE
Low Power mode (iCEgate enabled)

```
/**
 * PLL configuration
 *
 * This Verilog module was generated automatically
 * using the icepll tool from the IceStorm project.
 * Use at your own risk.
 *
 * Given input frequency:       100.000 MHz
 * Requested output frequency:   16.000 MHz
 * Achieved output frequency:    16.016 MHz
 */

module pll(
    input  clock_in,
    output clock_out,
    output locked
    );

SB_PLL40_CORE #(
        .FEEDBACK_PATH("SIMPLE"),
        .DIVR(4'b0011),     // DIVR =  3
        .DIVF(7'b0101000),  // DIVF = 40
        .DIVQ(3'b110),      // DIVQ =  6
        .FILTER_RANGE(3'b010)   // FILTER_RANGE = 2
    ) uut (
        .LOCK(locked),
        .RESETB(1'b1),
        .BYPASS(1'b0),
        .REFERENCECLK(clock_in),
        .PLLOUTCORE(clock_out)
        );

endmodule
```

Dans le iCE hx8k, il y a deux circuits de PLL.

Et les communications avec l'extérieur ?

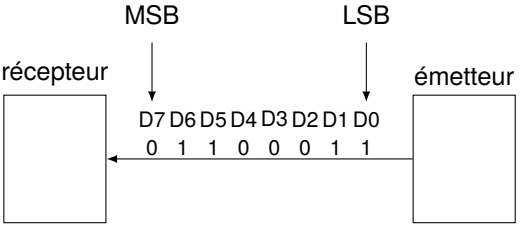
Transmission parallèle



- LSB : «*least significant bit*»
- MSB : «*most significant bit*»

Les bits sont émis **simultanément** sur autant de fils que de nombre de bits utilisé pour le codage.
Ce mode est employé pour les bus internes des ordinateurs (bus 16, 32 ou 64bits) parfois pour la communication vers des périphériques (imprimantes, bus SCSI, bus IDE…).
Exemple : on transmet un octet sur 8 fils, en envoyant en même temps chaque bit sur chaque fil.

Transmission série



Les bits sont transmis **séquentiellement** sur un seul fil.
Dans les réseaux, qu'ils soient locaux ou étendus, c'est la transmission série qui est utilisée.

C'est la **liaison série** qui est la **plus utilisée** (disque dur SATA, USB, ...)

Transmission série sur un seul fil pour une liaison synchrone

- *émetteur*, E, et *récepteur*, R, utilisent une **même base de temps** pour émettre les bits (horloge) ;
- il sont **cadencés** suivant la même horloge ;
- à chaque «top d'horloge», un bit est envoyé et R sait donc «quand» récupérer ce bit.

Le récepteur reçoit de façon continue les informations au rythme auquel l'émetteur les envoie.

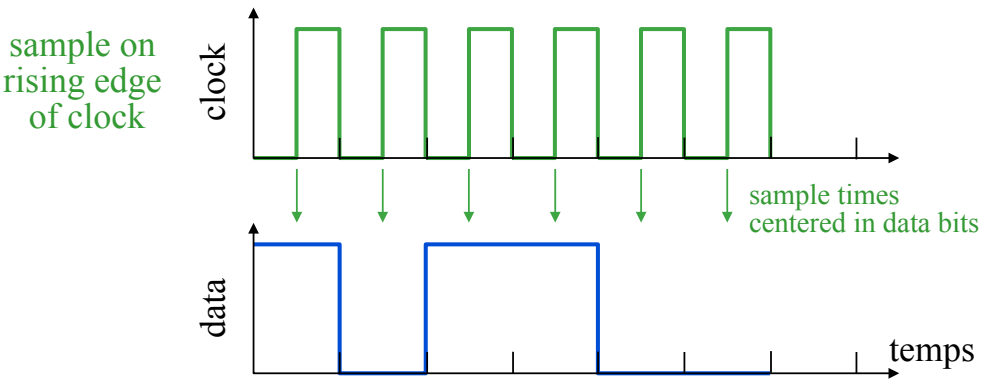
Inconvénient :

▷ la reconnaissance des informations au niveau du récepteur : il peut exister des différences entre les horloges de l'émetteur et du récepteur.

C'est pourquoi chaque envoi de bit doit se faire **sur une durée assez longue** pour que le récepteur la distingue.

Ainsi, la vitesse de transmission **ne peut pas être très élevée** *dans une liaison synchrone sans recourir à du matériel coûteux.*

Transmission série sur deux fils pour une liaison synchrone



Transmission série sur un seul fil pour une liaison asynchrone

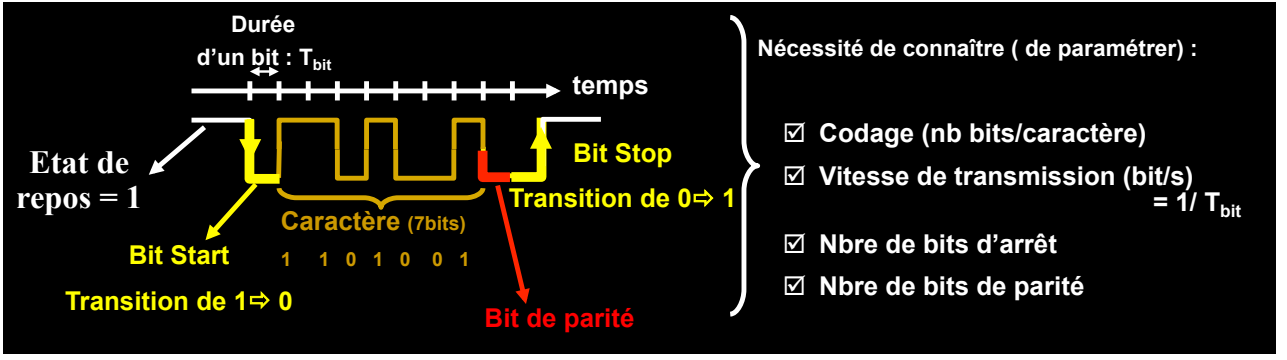
L'émetteur et le récepteur ne sont pas *synchronisés*.
Le récepteur doit détecter des **transitions** au sein des données reçues.

Problème Si un seul bit est transmis pendant une longue période de silence… le récepteur ne pourrait savoir s'il s'agit de `00010000`, ou `10000000` ou encore `00000100`…

Solution **Chaque caractère** est :

- **précédé d'une information** indiquant le début de la transmission du caractère (l'information de début d'émission est appelée bit START) ;
- **terminé par l'envoi** d'une information de fin de transmission (appelée bit STOP, il peut éventuellement y avoir plusieurs bits STOP).

Exemple



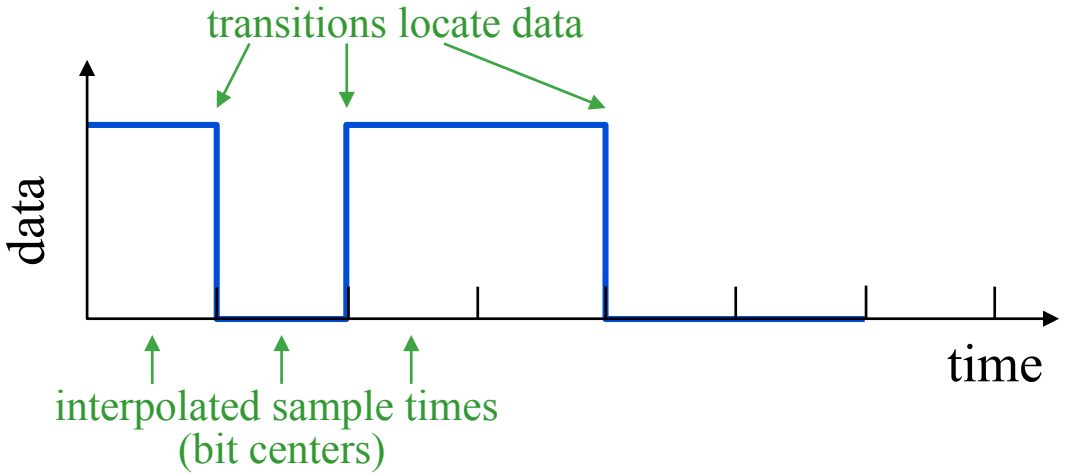
Ici, le codage consiste à passer d'une tension à l'autre seulement si on veut transmettre un bit de valeur différente.

Transmission série sur un seul fil pour une liaison asynchrone

L'émetteur et le récepteur ne sont pas *synchronisés*.

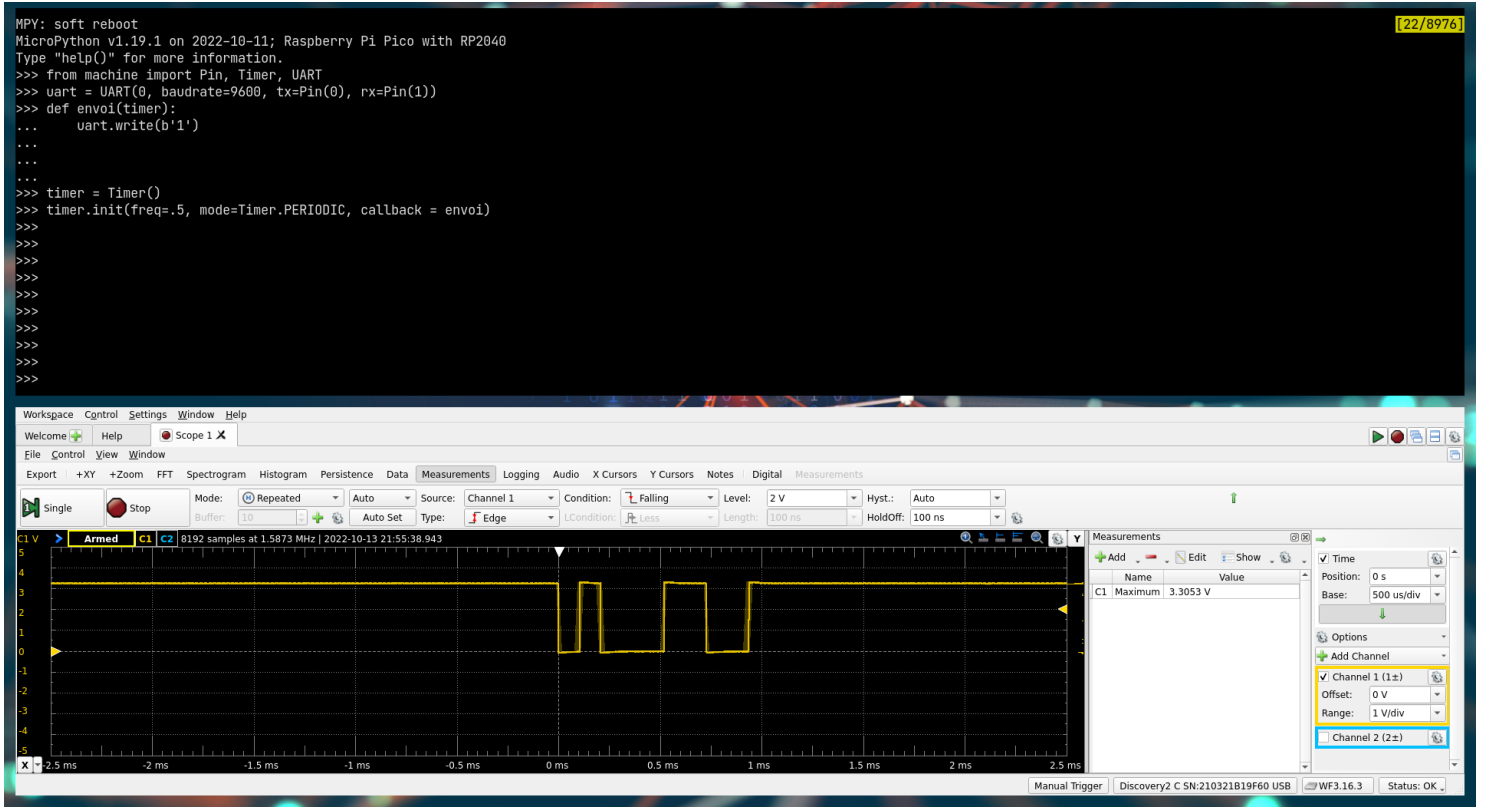
Le récepteur, pour se **synchroniser tout seul** :

- ▷ **connaît le débit** de transmission ;
- ▷ **recherche des transitions** pour se synchroniser et interpoler des mesures d'échantillonnage...
- ▷ **extrait l'horloge** des données :



Et concrètement le port série ça donne quoi ?

À l'aide d'un oscilloscope on peut «*intercepter*» la transmission série sur la broche 0 :



Lorsque le port série ne transmet rien, la broche est au niveau haut, c-à-d à 3,3v.

Grâce à **l'oscilloscope**, on peut déterminer la **vitesse de transmission** :

Ici, on voit que $1/\Delta X = 9,6KHz$ ce qui est le cas : on a configuré le port série pour une transmission à 9600bps :

xterm

```
...
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
...
```

Notre SoC : un processeur, de la mémoire
un port série ?

PicoRV32

- is a CPU core that implements the RISC-V RV32IMC Instruction Set ;
- can be configured as RV32E, RV32I, **RV32IC**, RV32IM, or **RV32IMC** core ;
- optionally contains a built-in **interrupt controller**

Features and Typical Applications

- Small (750-2000 LUTs in 7-Series Xilinx Architecture)
- High fmax (250-450 MHz on 7-Series Xilinx FPGAs)
- Selectable native memory interface or AXI4-Lite master
- Optional IRQ support (using a simple custom ISA)
- Optional Co-Processor Interface

Core Variant	Slice LUTs	LUTs as Memory	Slice Registers
PicoRV32 (small)	761	48	442
PicoRV32 (regular)	917	48	583
PicoRV32 (large)	2019	88	1085

Un «soft core» *est un CPU implémenté dans un FPGA.*

Configuration du «*soft core*»

Le processeur PicoRV32 est 32 bits, mais il est possible de le programmer avec des instructions sur 16bits

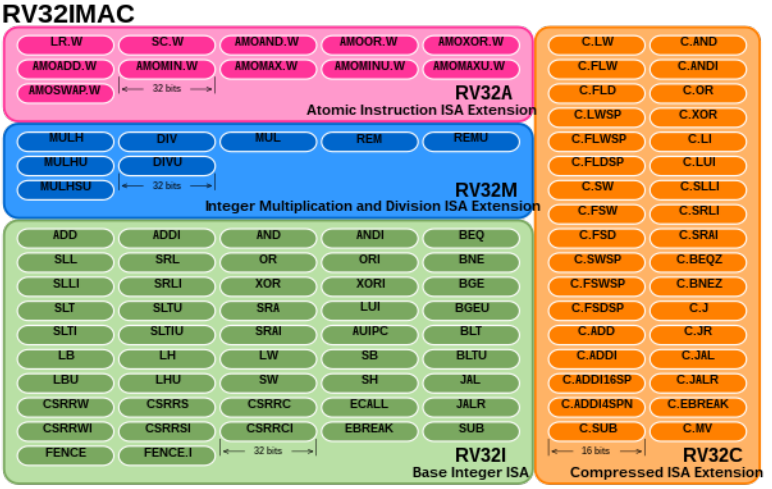
⟹Gain de place sur l'utilisation de la mémoire

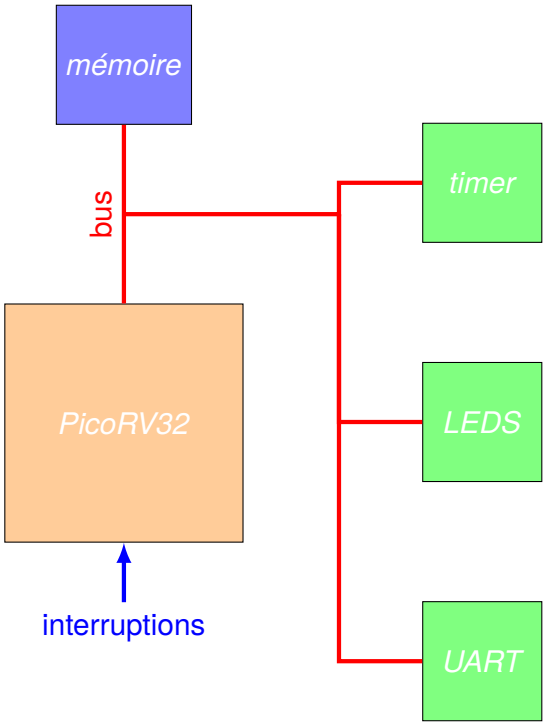
```
parameter [0:0] BARREL_SHIFTER = 1;
parameter [0:0] ENABLE_MUL = 1;
parameter [0:0] ENABLE_DIV = 1;
parameter [0:0] ENABLE_FAST_MUL = 0;
parameter [0:0] ENABLE_COMPRESSED = 1;
parameter [0:0] ENABLE_COUNTERS = 1;
parameter [0:0] ENABLE_IRQ_QREGS = 0;

parameter [31:0] STACKADDR = (32'h 0000_33f0);     // end of memory
parameter [31:0] PROGADDR_RESET = 32'h 0000_0000; // start of memory
parameter [31:0] PROGADDR_IRQ = 32'h 0000_0010;
```

activation des instructions sur 16bits

Gestion de l'IRQ sans registres supplémentaires

0x33f0 → 13296 octets = 3324 mots de 32bits





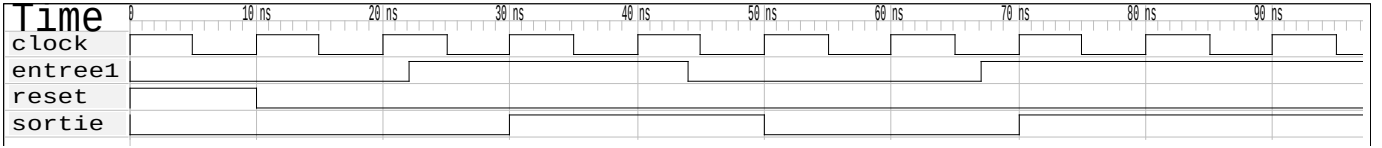
Le SoC, «*System On Chip*» est constitué de :

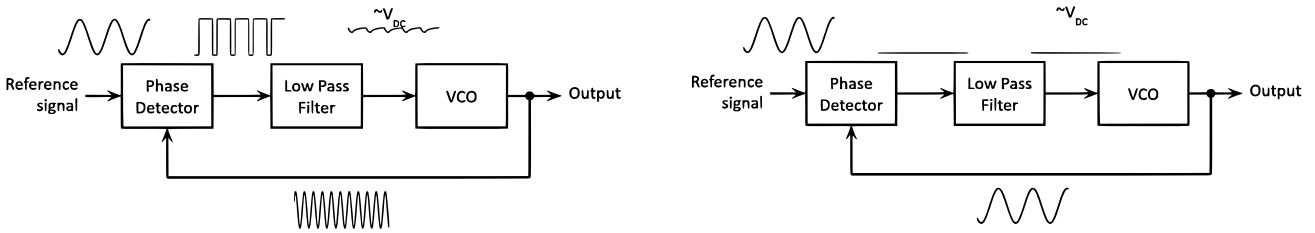
- un registre en entrée et un registre en sortie pour l'UART ;
- un registre en sortie pour les LEDs ;
- un registre en sortie pour un timer qui pourra générer un interruption ;
- de la BRAM pour la mémoire du PicoRV32.

Et si on veut changer
la vitesse de notre circuit ?

On fait varier l'entrée `in_a` du simulateur :

```
in_a = 1'b0;
#22
in_a = 1'b1;
#22
in_a = 1'b0;
#23
in_a = 1'b1;
#30 $finish;
```



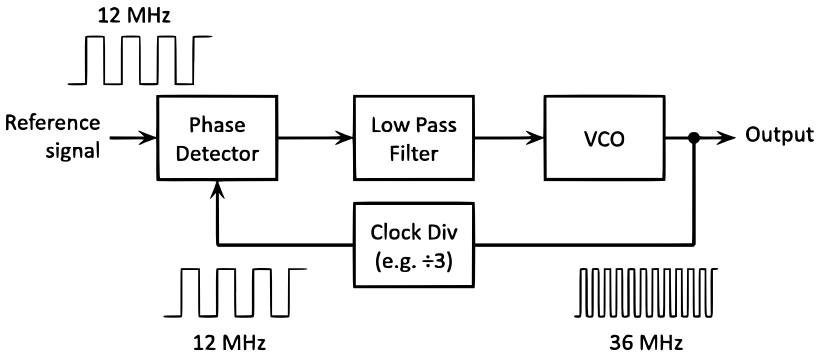


*Un signal est généré par le VCO, «*Voltage Control Oscillator*», et injecté dans le «*Phase detector*» qui génère une impulsion à chaque différence de phase entre ce signal et le signal de référence.*

⟹ce signal crée un signal qui après filtrage donne une ***variation de courant*** *qui est injectée en entrée du VCO.*

⟹après être arrivé à un équilibre : ***le signal de sortie est identique au signal en entrée****.*

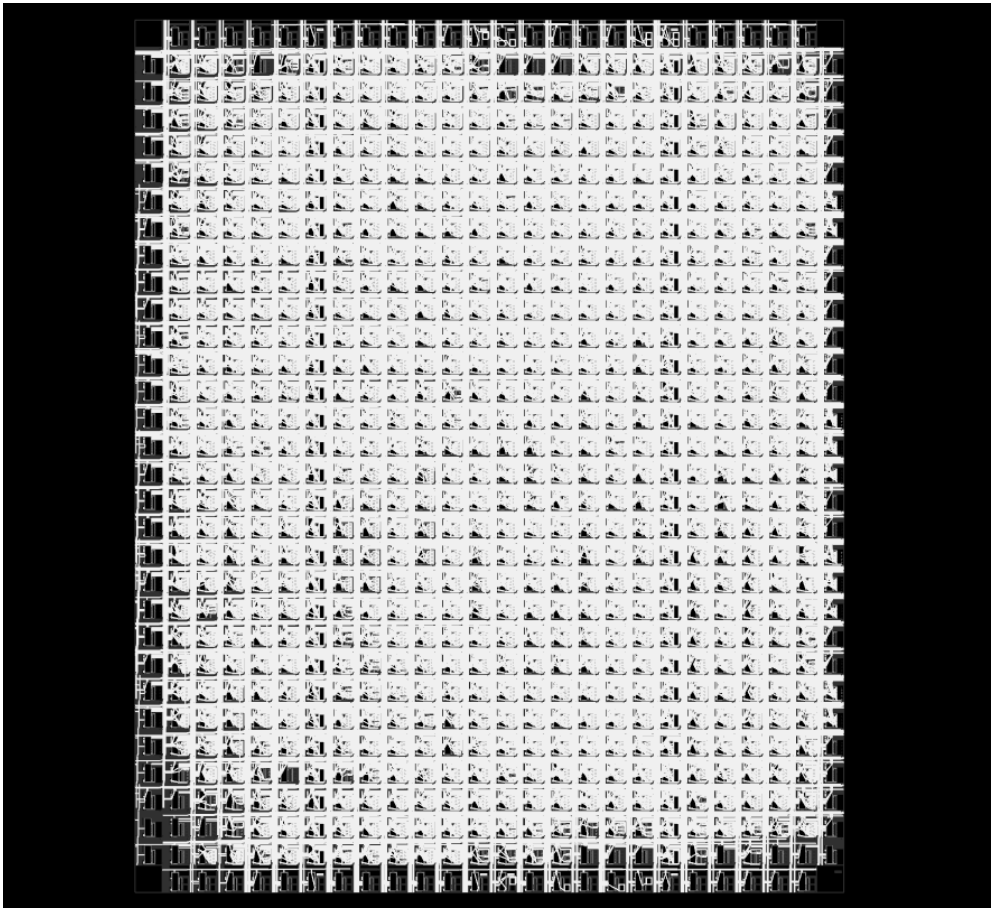
PLL Frequency Multiplier



*Ici, on introduit un «*clock divider*» ce qui permet d'arriver à* ***augmenter la fréquence*** *en du signal en sortie.*

Et pour l'exploitation «*bare metal*»
en C ?

PicoSoC vu par nextpnr



xterm

```
Info: Device utilisation:
Info:            ICESTORM_LC:  4873/ 7680    63%
Info:           ICESTORM_RAM:    30/   32    93%
Info:                  SB_IO:    20/  256     7%
Info:                  SB_GB:     8/    8   100%
Info:           ICESTORM_PLL:     1/    2    50%
Info:            SB_WARMBOOT:     0/    1     0%

Info:  2.0  4.6    Net pmod_hex[2]$SB_IO_OUT budget 20.371000 ns (1,7) -> (0,16)
Info:                Sink pmod_hex[2]$sb_io.D_OUT_0
Info:                Defined in:
Info:                  icebreaker.v:44.18-44.26
Info: 1.4 ns logic, 3.2 ns routing

Info: Critical path report for cross-domain path 'posedge observation$SB_IO_OUT_$glb_clk' ->
'posedge clk$SB_IO_IN':

Info: 1.3 ns logic, 2.8 ns routing

Info: Max frequency for clock 'observation$SB_IO_OUT_$glb_clk': 40.35 MHz (PASS at 16.00 MHz)
Info: Max frequency for clock                  'clk$SB_IO_IN': 227.79 MHz (PASS at 16.00 MHz)
```

Des **instructions spéciales** pour gérer les interruptions sont **ajoutées** au processeur et exprimées en assembleur :

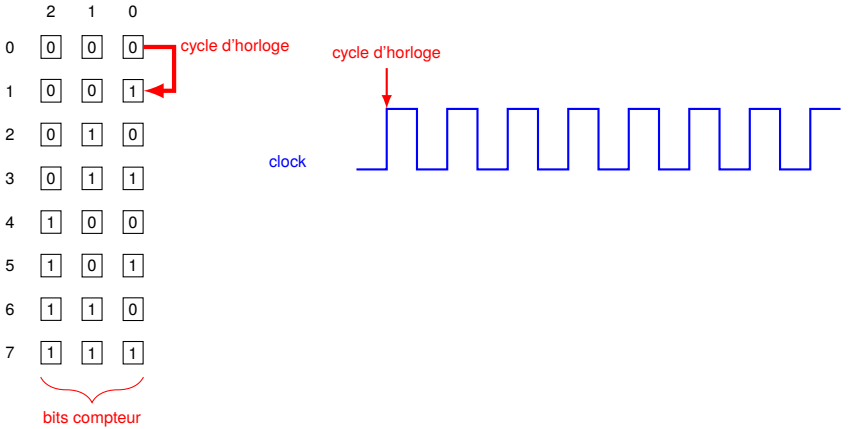
```
#define regnum_fp   8

#define r_type_insn(_f7, _rs2, _rs1, _f3, _rd, _opc) \
.word (((_f7) << 25) | ((_rs2) << 20) | ((_rs1) << 15) | ((_f3) << 12) | ((_rd) << 7) | ((_opc) << 0))

#define picorv32_retirq_insn() \
r_type_insn(0b0000010, 0, 0, 0b000, 0, 0b0001011)

#define picorv32_maskirq_insn(_rd, _rs) \
r_type_insn(0b0000011, 0, regnum_ ## _rs, 0b110, regnum_ ## _rd, 0b0001011)

#define picorv32_waitirq_insn(_rd) \
r_type_insn(0b0000100, 0, 0, 0b100, regnum_ ## _rd, 0b0001011)

#define picorv32_timer_insn(_rd, _rs) \
r_type_insn(0b0000101, 0, regnum_ ## _rs, 0b110, regnum_ ## _rd, 0b0001011)
```

```
#include "irq_functions.h"
#include <stdint.h>

void __attribute__((naked)) _picorv32_setmask(uint32_t to)
{
    picorv32_maskirq_insn(a0, a0);
    asm __volatile__ ("ret\n");
}

void __attribute__((naked)) _picorv32_timer(uint32_t to)
{
    picorv32_timer_insn(a0, a0);
    asm __volatile__ ("ret\n");
}

uint32_t *__attribute__((naked)) _picorv32_waitirq(void)
{
  picorv32_waitirq_insn(a0);
  asm __volatile__ ("ret\n");
}
```

Finalement la fonction `main` utilise les **nouvelles instructions** par l'intermédiaire de «*wrapers*» : des fonctions C pour charger automatiquement les paramètres passés à la fonction dans les registres utilisés par ces nouvelles instructions :

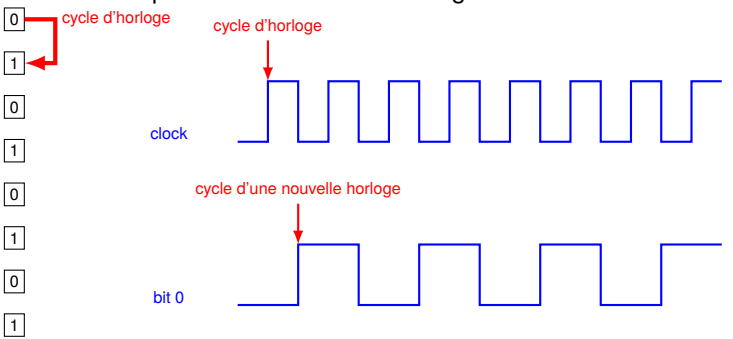
```
int main(void)
{
  _picorv32_setmask(0);
  _picorv32_timer(COUNT);
  print_str("\r\nHello !\r\n");
  for(;;)
  {
      _picorv32_waitirq();
      print_str("Go !\r\n");
  }
}
```

Exemple de compteur sur 3 bits : un circuit additionneur avec des flip-flops



Que peut-on faire avec ? Diviser le signal d'horloge

Si on regarde comment varie le bit 0 du compteur en fonction de l'horloge :



$\Longrightarrow$ On vient de diviser par 2 le signal d'horloge ! (le bit 1 du compteur diviserait par 4, *etc.*)

Comment compter le temps qui passe ?

- ▷ écrire un **programme** qui contient une **boucle infinie** (sans condition d'arrêt) dans laquelle :
  - ◇ on incrémente un variable qui contient le nombre de fois que la boucle a été exécutée ;
  - ◇ en connaissant :
    - ⋆ le nombre cycles d'horloge nécessaire à l'exécution ;
    - ⋆ la vitesse de l'horloge du processeur ;

    On peut calculer le **temps d'une occurence** de la boucle et le **temps écoulé** depuis que l'on compte.

  ⟹***Problème le processeur ne fait rien d'autre et devient inutile !***

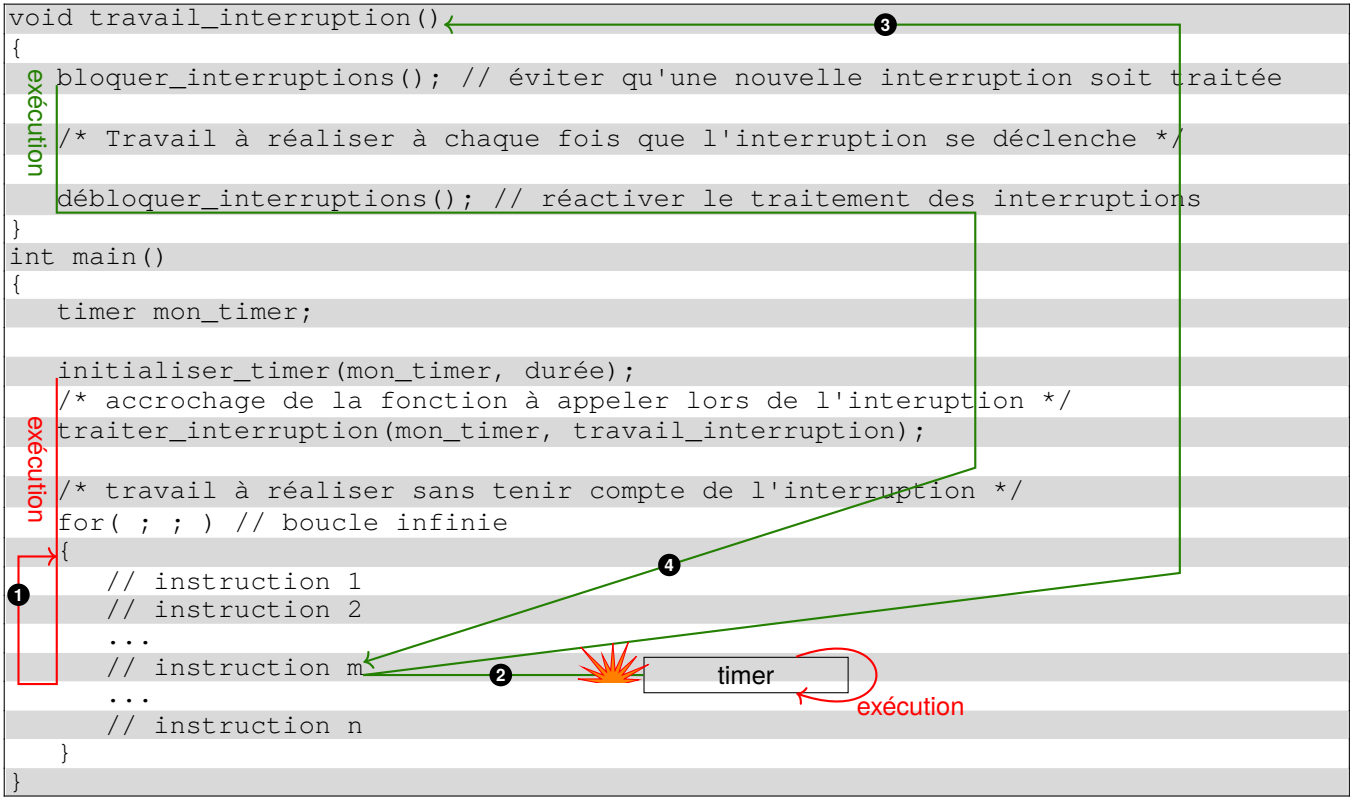
- ▷ utiliser un «*timer*» ou **compteur matériel** :
  - ◇ c'est un circuit **indépendant** du processeur ;
  - ◇ il peut être de grande dimension comme par exemple sur 32bits ;
  - ◇ il compte suivant les cycles de l'horloges qu'il reçoit comme le processeur ;

Attendre un certain délai

- ▷ le processeur peut **consulter régulièrement** la valeur du «*timer*»... mais on se retrouve un peu dans la même situation que précédemment...
- ▷ permettre au «*timer*» de **dérouter le processeur de son travail courant** vers un travail particulier au moment où le «*timer*» atteint une valeur particulière :

  ⟹On utilise le **mécanisme d'interruption** !

Mécanisme d'interruption : dérouter le travail du processeur



- ▷ Programme tourne en boucle ❶;
- ▷ Interruption survient ❷;
- ▷ Programme est interrompu pour exécuter la fonction ❸;
- ▷ une fois la fonction finie, on revient au programme ❹;