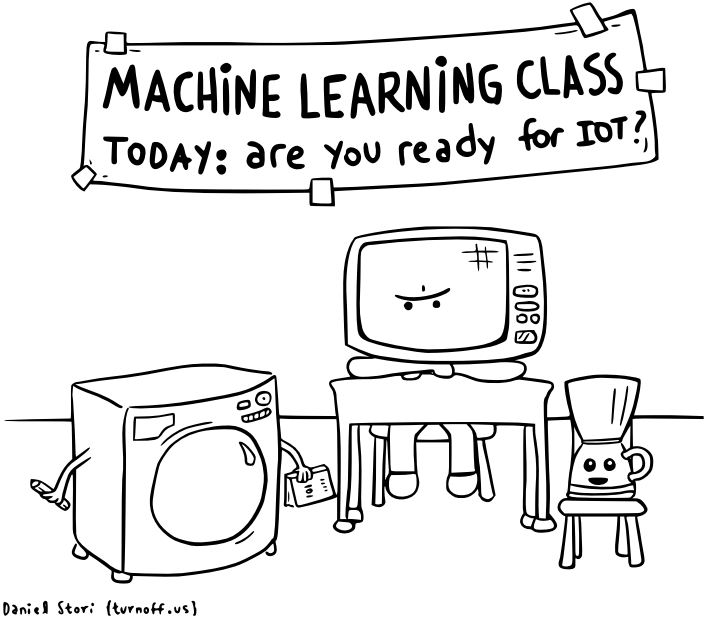
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

3A MRI



Informatique industrielle — P-F. Bonnefoi

Version du 13 juin 2023

Objectif et Moyens

But du module d'enseignement

Réalisation de **circuits électroniques** contrôlés par

- ▷ **circuits logiques** ;
- ▷ **programmation logicielle**.

Moyens pratiques

La **carte de développement** PSoC 4 pioneer kit de la société Cypress Semiconductor Corp. :

- ▷ disposer des **composants matériels** nécessaires :
  - ◇ **composants électroniques** : LEDs, micro-switch, resistances, alimentation, broches d'E/S, *etc.*
  - ◇ **circuit logique reconfigurable** : opérateurs logiques ET, OU, NON, XOR, routage configurable permettant **d'interconnecter** ces opérateurs aux composants électroniques ;
  - ◇ **micro-contrôleur** : processeur ARM disposant de mémoire de travail et de mémoire non-volatile ;
- ▷ offrir un **environnement logiciel** tout intégré permettant de configurer/programmer la carte de développement : PSoC Creator.

Évaluation

- ▷ Chacun des 3 TPs donnera lieu à un évaluation sur **papier sur 20 mins**.

Contenu de la présentation

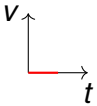
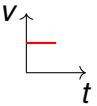
Plan

- Rappels sur la notion d'information :
  - Qu'est-ce qu'un bit d'information :
    - comment est-il créé dans un circuit logique ;
    - comment est-il manipulé dans un ordinateur ;
  - notion d'octet et d'adresse ;
- Présentation de la carte de développement PSoC ;
- Rappels sur le fonctionnement d'un ordinateur : illustration avec le processeur 8bits 6502 :
  - notion de bus de données et d'adresse ;
  - notion d'horloge ;
  - notion de programmation de bas niveau ;
  - notion de «*timer*» et d'interruption ;
- Qu'est-ce qu'un micro-contrôleur et un SoC ;
- Les communications :
  - vers l'extérieur : le port série ;
  - entre les composants : bius i2c et SPI.
- Présentation des outils de développement matériel et logiciel avec PSoC :
  - création de circuit logique électronique ;
  - programmation en C du micro-contrôleur Cortex M0+.

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.»

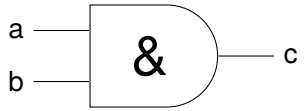
Chaque bit de mémoire correspond à une case dans laquelle on peut stocker un bit de données, soit la valeur 1, soit la valeur 0.

Et un circuit logique, c'est quoi ?

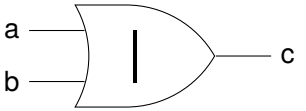
des opérateurs logiques...

et des composants électroniques !

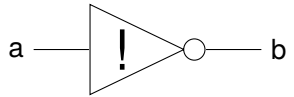
Le «*et*» : $c = a \,\&\, b$ ou $c = a \wedge b$



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

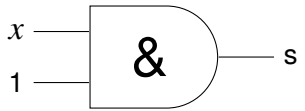


Le «*non*» : $b = !a$ ou $b = \neg a$

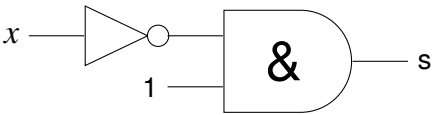


Créer des opérations

L'opération $x = 1$



L'opération $x = 0$

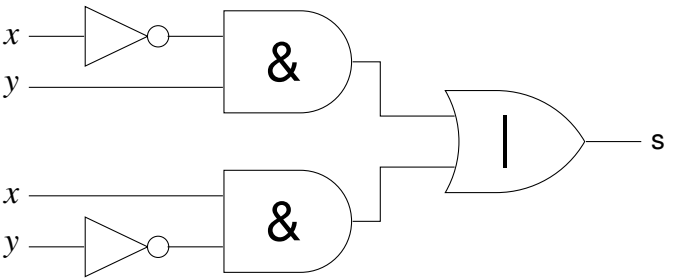


L'opérateur «*xor*»

Table de vérité du *xor*

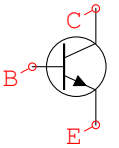
a	b	⊕
0	0	0
0	1	1
1	0	1
1	1	0

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



Le transistor

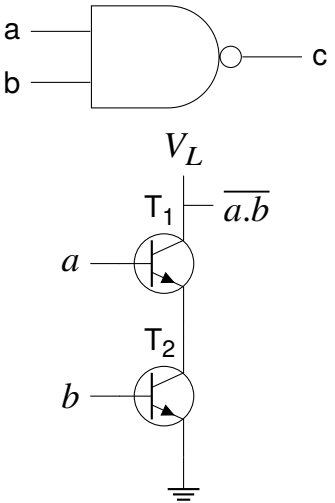
Le **transistor** agit comme un **interrupteur** : le courant peut circuler du «*collecteur*» vers «*l'émetteur*» uniquement si une tension est présente sur la «*base*» :



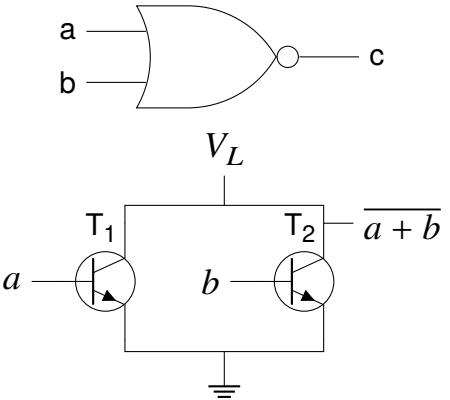
Simuler les portes logiques ?

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

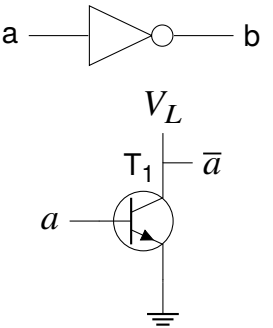
Le «*non-et*» ou NAND :



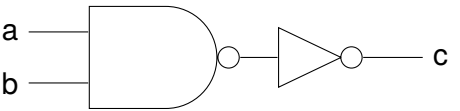
Le «*non-ou*» ou NOR :



Le «*non*» :



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

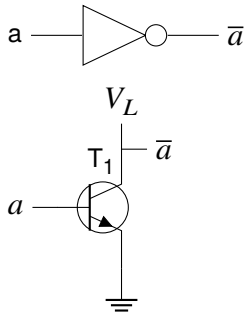


Et en vrai ?

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_L pour le «*1*» logique $\Longrightarrow$ ce qui est relié directement à V_L ;

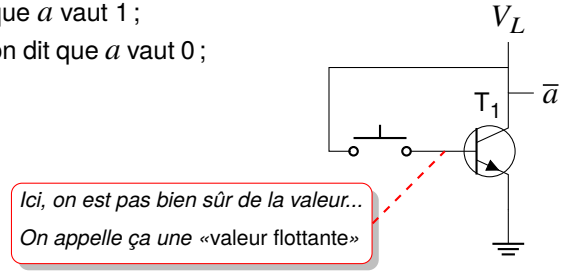
Exemple sur le «*non*» :



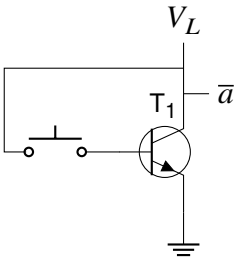
Si l'entrée a est connectée à

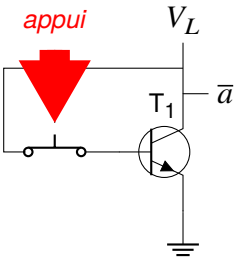
- ▷ V_L alors on dit que a vaut 1 ;
- ▷ «*ground*» alors on dit que a vaut 0 ;

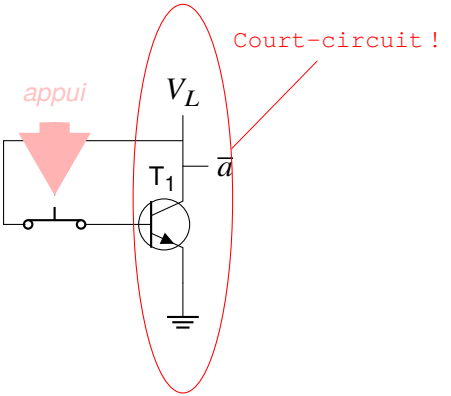
Ce qui donne :



Et électriquement, ça marche ?







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.

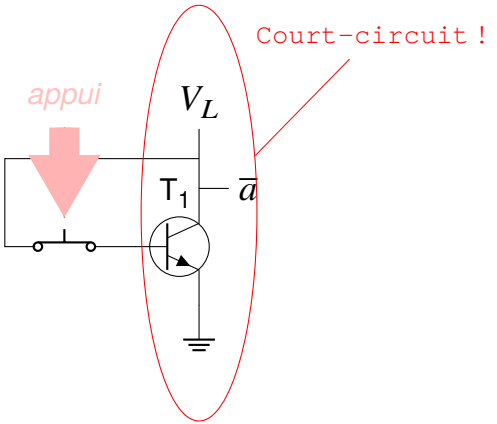
Loi d'Ohm $U = R * I$, ou «*volts et résistance crée l'ampérage*»

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

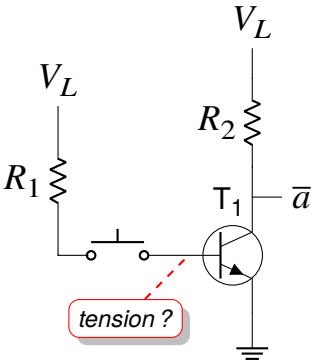
Ce qui permet de distinger **3 situations de panne** dans un circuit :

- ▷ 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...



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



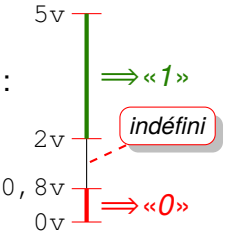
On évite deux court-circuits possibles avec R_1 et R_2.

Par contre, on ne connait pas la tension à l'entrée de la «base» 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_L, 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 ?

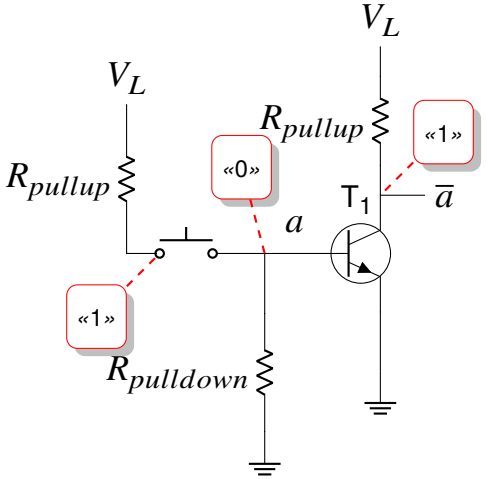
On rajoute des résistances de «*pull up*» pour :

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

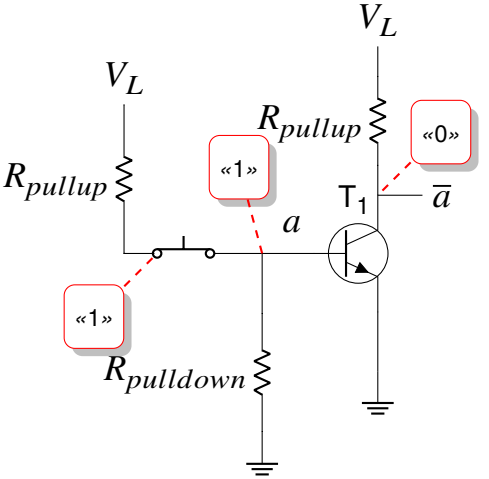
On rajoute des résistances de «*pull down*» pour :

- ▷ **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 :

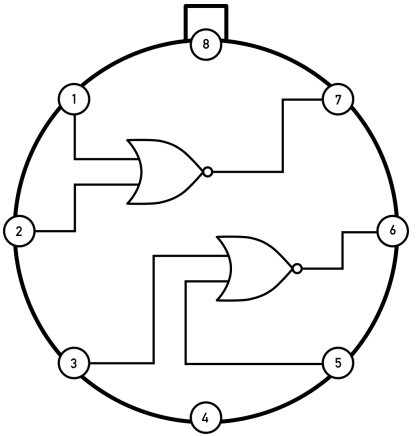


Si $a = 0$ alors $\overline{a} = 1$



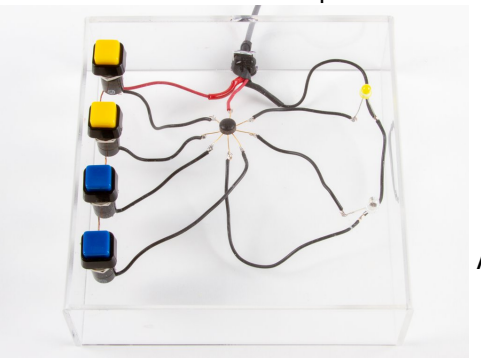
Si $a = 1$ alors $\overline{a} = 0$

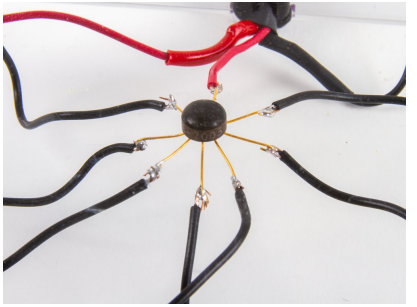
Et en réalité ? Exemple du composant µL914 de la société Fairchild



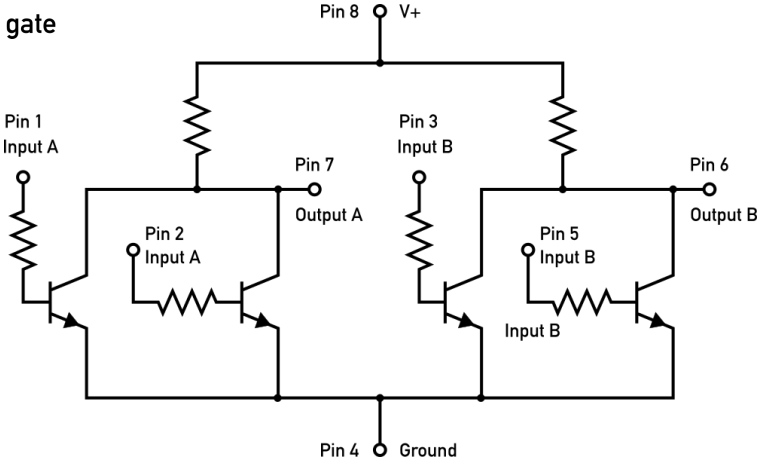
µL914
Dual 2-input NOR gate

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



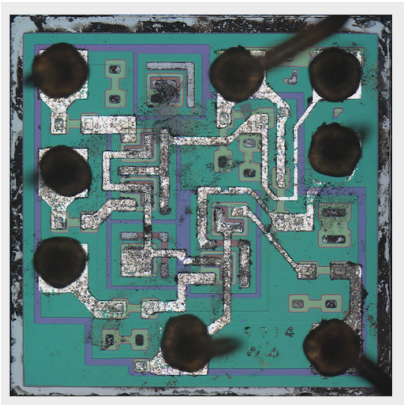


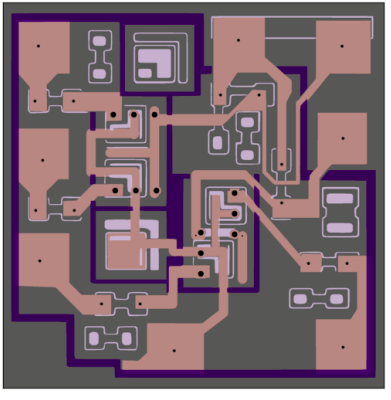
Agrandissement du composant.

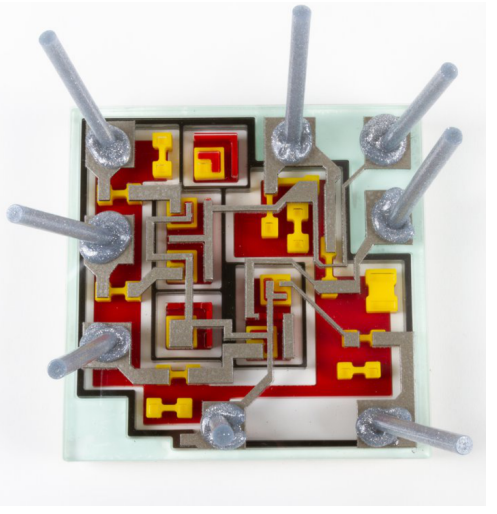


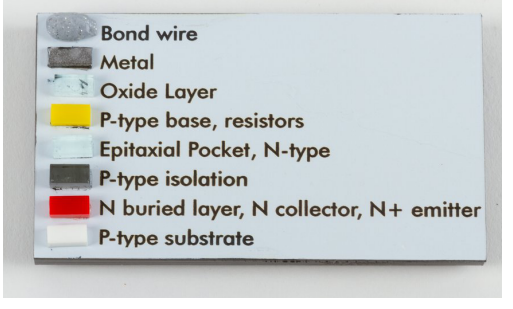
Et à l'intérieur ?

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





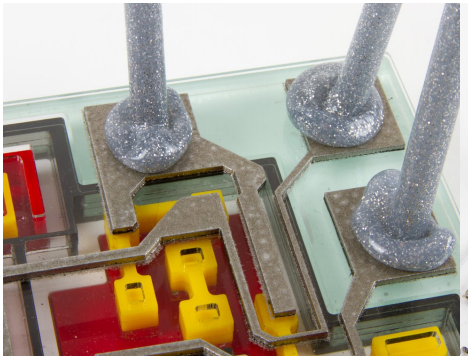




Le composant est constitué de différentes couches de matériaux différents, superposées les unes sur les autres.
On obtient chaque couche par dépôt de substrat ou par gravure (creusement d'une couche).

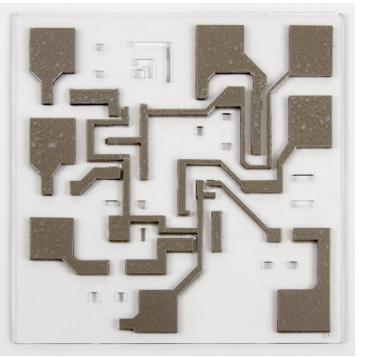
Et à l'intérieur ?

Fils de connexion vers l'extérieur :

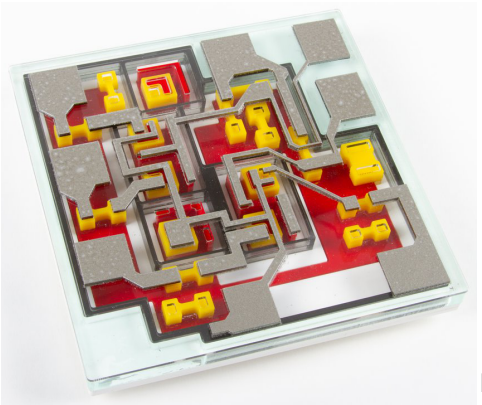


Connexion sur la couche conductrice :

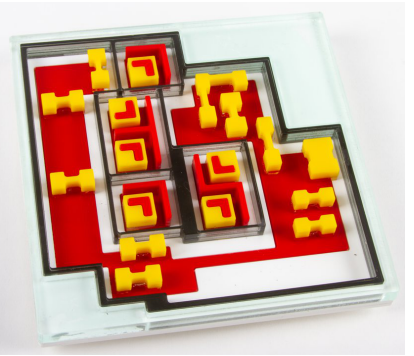
Couce conductrice :



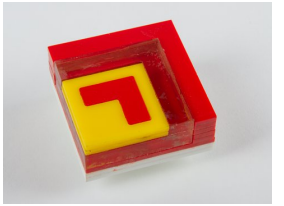
Toutes les couches :



Sans la couche conductrice :



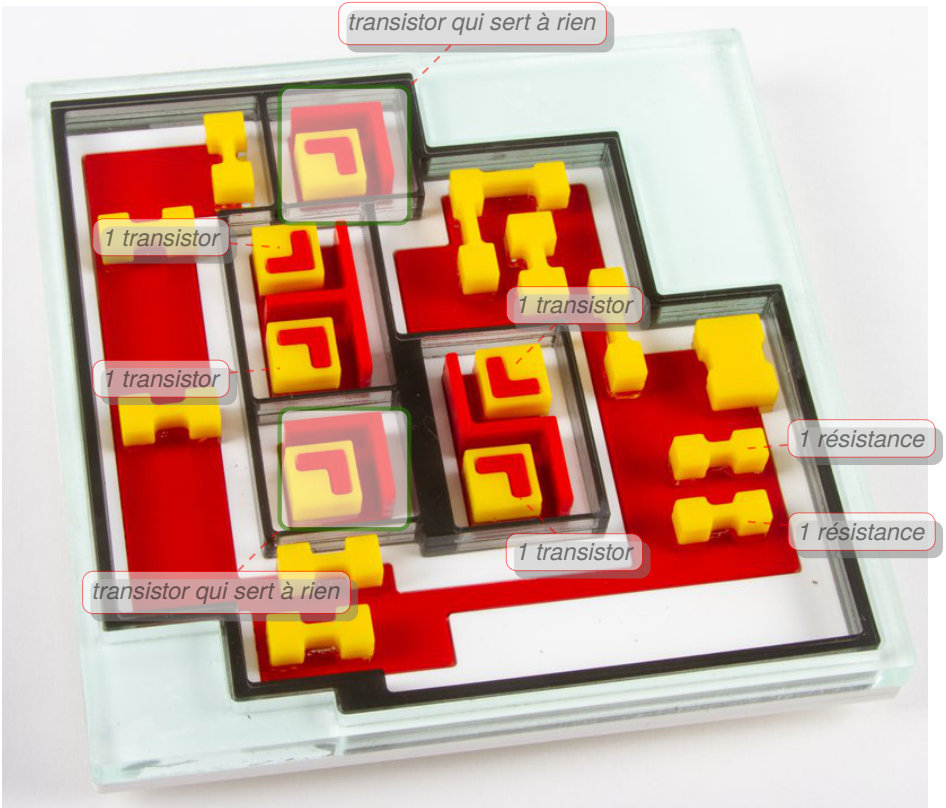
Les Transistors :



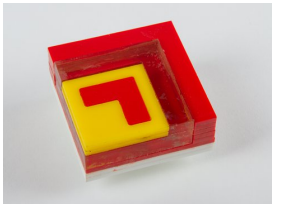
Les Résistances :

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.

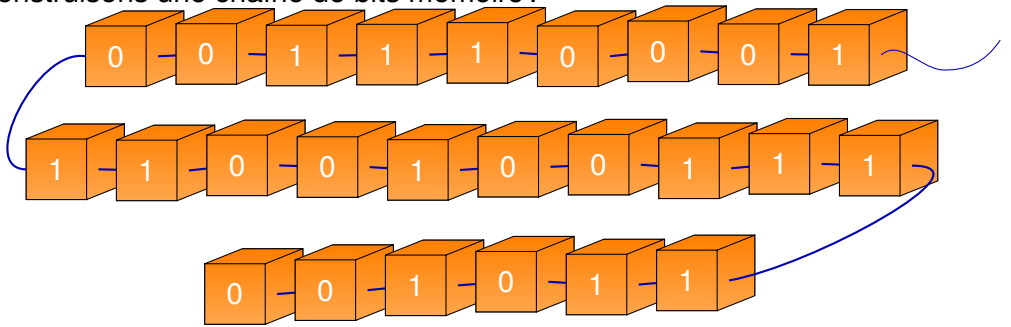
Mais dans un ordinateur

comment manipule-t-on des bits ?

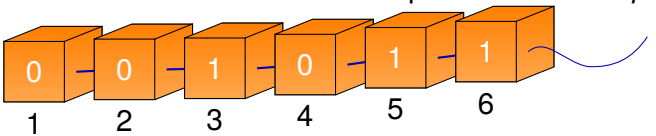
$\Longrightarrow$octet & adresses

Comment accéder à la valeur de chaque bit de mémoire ?

Construisons une chaîne de bits mémoire :

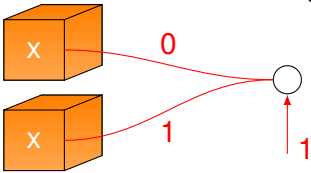


Comment accéder à une case précise ? ⟹ *On pourrait numéroter les cases !*



Mais, si le numéro de la case doit être **manipulé par l'ordinateur**, il doit **être mémorisé** dans des cases !

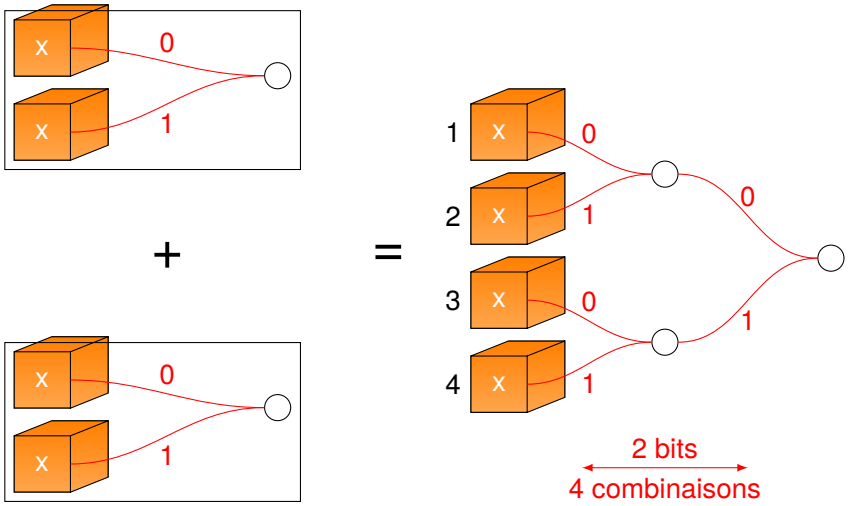
Combien de bits faut-il pour numéroter toutes les cases mémoires ?



⟹ 1 bit donne deux combinaisons : «*0*» et «*1*»

Un bit de «*comptage*» peut compter deux bits de mémoire... Si on veut 100 cases mémoires, il faut 50 cases pour les compter ? Non c'est un peu plus compliqué que cela...

Numérotation des cases



Numéro case	bits n
1	00
2	10
3	01
4	11

On remarque que si on lit la combinaison des bits n de droite à gauche, on arrive directement au bit désiré :

Numéro case	bits n
1	00
2	01
3	10
4	11

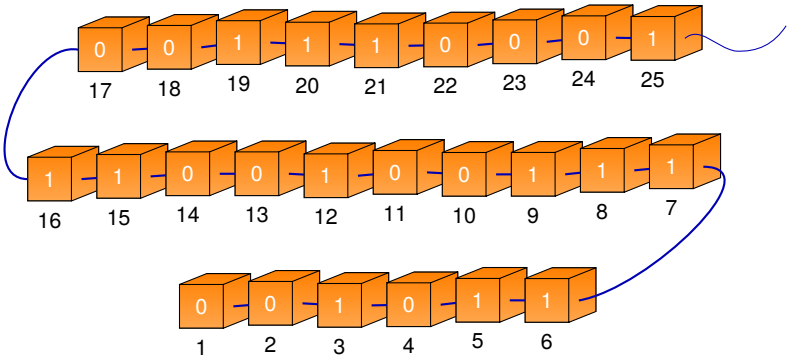
$\Longrightarrow$ Pour 4 cases, ou 4 bits de données, il faut 2 bits de numérotation.
$\Longrightarrow$ Pour 8 cases, il faut 3 bits de numérotation.
$\Longrightarrow$ Pour 16 cases, il faut 4 bits de numérotation.
$\Longrightarrow$ Pour 32 cases, ou 32 bits de données, il faut 5 bits de numérotation.
Il faut $log_2(N)$ bits de numérotation pour N cases
$\Longrightarrow$ optimal mais pas très économique !

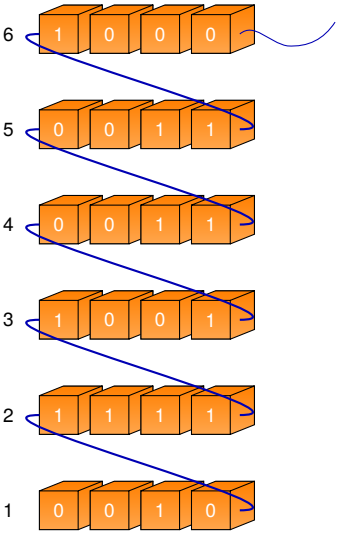
Si on numérote en partant de zéro, la numérotation correspond à la séquence binaire :

Numéro case	bits n
0	00
1	01
2	10
3	11

Comment diminuer la numérotation ?

On associe les bits mémoires par groupe de taille donnée, puis on compte ces groupes :





Ici, on a regroupé par groupe de 4 bits, appelé «*quartet*» ou «*nibble*».

Il faut :

- ▷ 1 bit de numérotation pour deux quartets, ou 8 bits mémoire ;
- ▷ 2 bits de numérotation pour 4 quartets, ou 16 bits de mémoire ;
- ▷ 3 bits de numérotation pour 8 quartets, ou 32 bits de mémoire ;

C'est mieux !

Quelle est la bonne taille de regroupement des bits de mémoire ?

Il faut trouver un compromis entre :

- le nombre de bits nécessaire à la **numérotation** de ces groupes ;
- l'intérêt qu'un groupe peut représenter pour **coder** de l'information.

Quel est l'intérêt d'un groupe ? $\Longrightarrow$ *le nombre de séquences de bits différentes qu'il peut exprimer*.

Avec un quartet ou «*nibble*», combien de **séquences différentes** de bits peut-on exprimer ?

$$2 * 2 * 2 * 2 = 2^4 = 16$$

Et si on voulait associer un caractère de l'alphabet à chacune de ces séquences ? On ne pourrait pas exprimer l'alphabet entier !

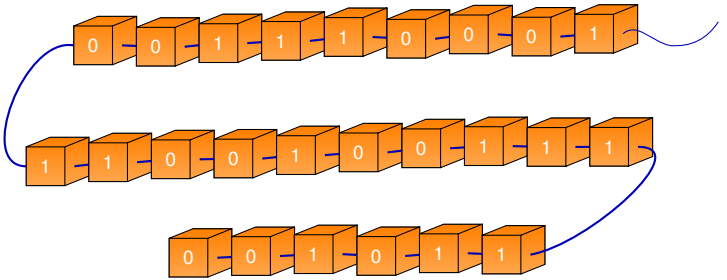
séq.	lettre	séq.	lettre	séq.	lettre	séq.	lettre
0000	a	1000	e	1111	i	1010	m
0001	b	1001	f	0110	j	1011	n
0010	c	1100	g	0101	k	1101	o
0100	d	1110	h	1001	l	0010	p

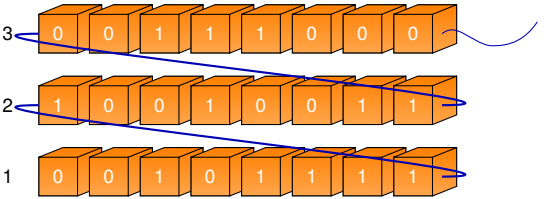
$\Longrightarrow$Il faut trouver un **meilleur compromis** !

- $2^5 = 32 \Longrightarrow$ on peut exprimer un alphabet mais il reste juste 6 caractères pour la ponctuation...
- $2^6 = 64 \Longrightarrow$ un alphabet + 10 chiffres + caractères accentués + ponctuation, encore un peu juste
- $2^7 = 128 \Longrightarrow$ plutôt satisfaisant.
- $2^8 = 256 \Longrightarrow$ le choix qui a été fait.

Regroupement par octet ou «*byte*»

On regroupe les bits de mémoires par groupe de 8, appelé octet ou «*byte*» :





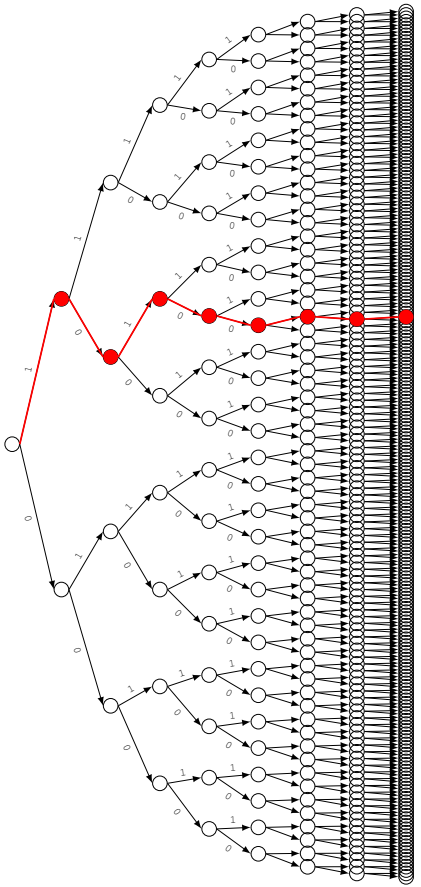
Il faut :

- ▷ 1 bit de numérotation pour 2 octets ou 16 bits de mémoire ;
- ▷ 2 bit de numérotation pour 4 octets ou 32 bits de mémoire ;
- ▷ 3 bits de numérotation pour 8 octets ou 64 bits de mêmoire ;
- ▷ 4 bits de numérotation pour 16 octets ou 128 bits de mémoire ;
- ▷ 8 bits de numérotation pour 256 octets ou 1024 bits de mémoire ;
- ▷ 16 bits de numérotation pour 65536 octets ou 64 Kilo-octets de mémoires ;

Attention

En informatique, «*kilo*» représente 1024 lorsqu'il s'agit de manipulation dans l'ordinateur.
Parce que «$2^{10} = 1024$».
Dans le SI, «*système international*», «*kilo*» signifie 1000 et **kibi** signifie 1024.
C'est l'unité utilisée pour les débits réseaux : 20 Mbits $\Longrightarrow 10^6$ bits ou la capacité de stockage.

Exemple : Numérotation sur un octet, ou «*adressage sur 8bits*»



l'octet désiré
à `10100101`

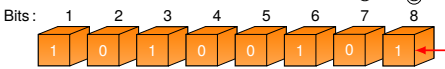
Adressage

La «*numérotation*» s'appelle «*l'adressage*».
Chaque octet dispose d'une **adresse**.

Le processeur manipule les octets par leur adresse.

Une séquence de bits d'au plus 8bits donne un octet, et si elle est plus longue que 8bits, elle peut s'étendre sur plusieurs octets consécutifs.

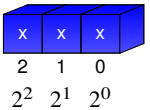
Un **pointeur en C** correspond à une adresse et indique également la longueur de la séquence de bits.
Exemple : `int *entier` est une variable qui contient l'adresse où débute une séquence de 32bits correspondant à un entier.



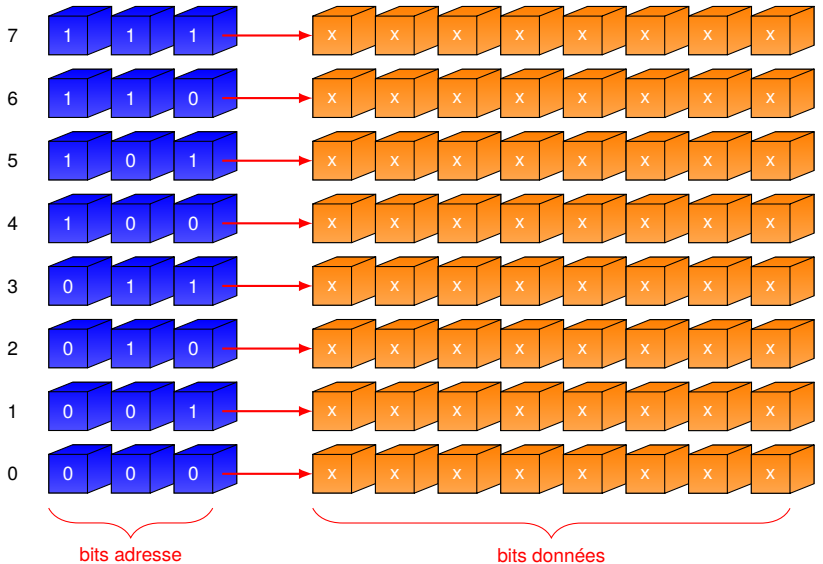
Ces 8 bits constituent un octet qui contient l'adresse de l'octet en mémoire.

Exemple : adressage sur 3 bits

- il y a $2^3 = 8$ octets adressables ;
- chaque adresse peut être codée sur 3 bits :

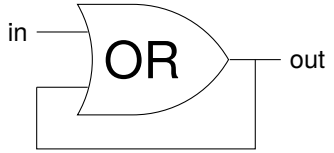


Ce qui donne une mémoire de **8 octets** ou 64 bits :



Mais comment les bits sont mémorisés ?

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

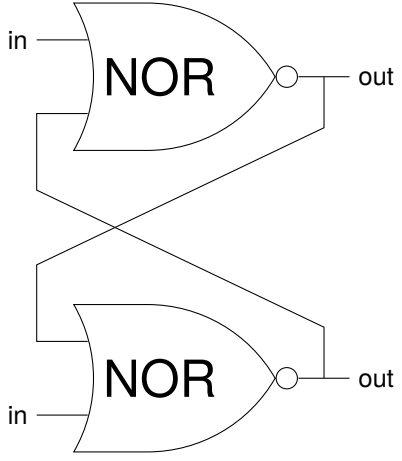


a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

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 !*

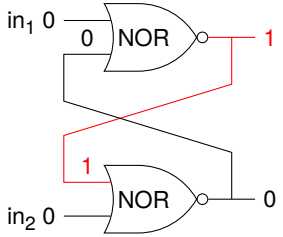
Comment faire pour «*l'éteindre*» ?



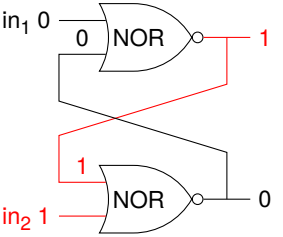
a	b	NOR
0	0	1
0	1	0
1	0	0
1	1	0

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

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

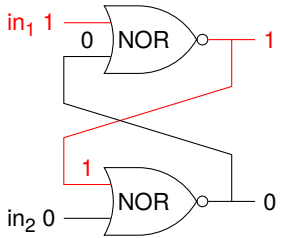


Si on mets l'entrée «in_2» à un :
⟹ l'état du circuit ne change pas :

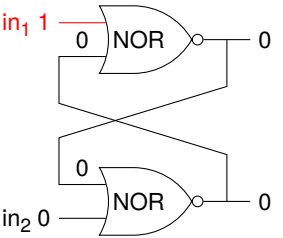


a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

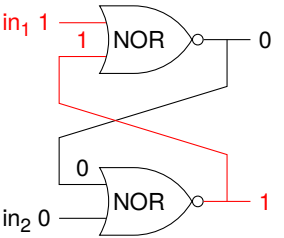
Si on mets l'entrée «in_1 *à un*» :
⟹ on obtient :



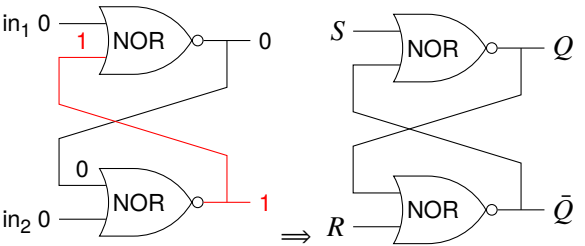
Si on mets l'entrée «in_1» à un :
⟹ l'état du circuit change vers :



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

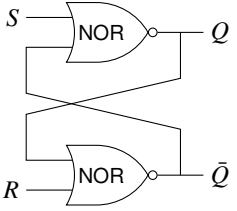


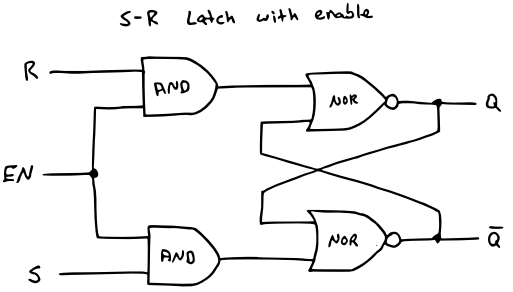
Si on remets l'entrée «in_1» à zéro :
⟹ le circuit ne change pas ;
⟹ on est dans l'image inversée de l'état précédent où l'état ne changeait plus...
⟹ **On vient de construire une «*S-R Latch*», une «*set/reset*» latch ! : un bouton «*set*» et l'autre «*reset*»**
⟹ **On vient de mémoriser un état !**



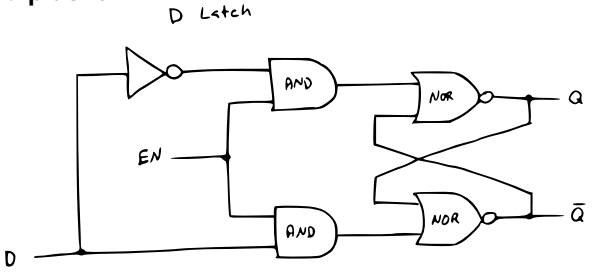
Améliorer le circuit ? Ajouter une entrée «*enable*»

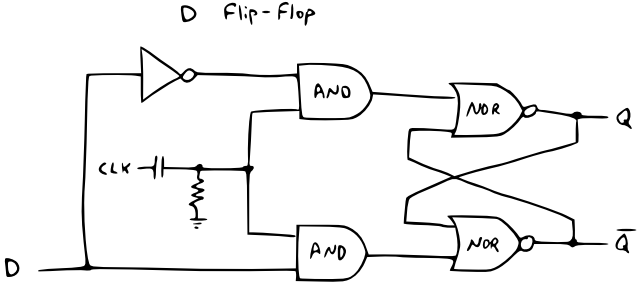
On ajoute le «*enable*» qui permet d'activer ou non la SR-latch :





Si on va plus loin





À chaque fois que :

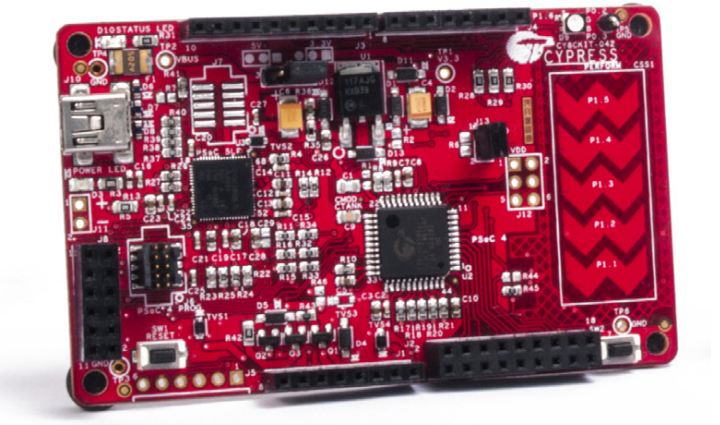
- ▷ l'entrée «*enable*» est active : la sortie «Q» reproduit la valeur de l'entrée «D» ;
- ▷ l'entrée «*enable*» est inactive : la sortie «Q» conserve sa valeur quelque soit la valeur de D ;

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

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

Si on utilise l'horloge pour l'entrée «*enable*» on obtient une **mémoire synchronisée** sur l'horloge !
$\Longrightarrow$ la «*D-flipflop*»
La valeur D est mémorisée/accessible au moment où l'horloge change $\Longrightarrow$ on peut synchroniser différents circuits entre eux.

Et si on regardait ce que propose PSoC ?



- **32-bit MCU subsystem**
  - 48 MHz Arm Cortex-M0 CPU with single cycle multiply
  - Up to 32 KB of flash with read accelerator
  - Up to 4KB of SRAM

un micro-contrôleur de type ARM

- **Programmable analog**

des composants de mesures

  - Two opamps with reconfigurable high-drive external and high-bandwidth internal drive, comparator modes, and ADC input buffering capability
  - 12-bit 1-Msps SAR ADC with differential and single-ended modes; channel sequencer with signal averaging
  - Two current DACs (IDACs) for general-purpose or capacitive sensing applications on any pin
  - Two low-power comparators that operate in deep sleep
- **Programmable digital**

de la logique reconfigurable

  - Four programmable logic blocks called universal digital blocks (UDBs), each with eight Macrocells and data path
  - Cypress-provided peripheral component library, user-defined state machines, and Verilog input

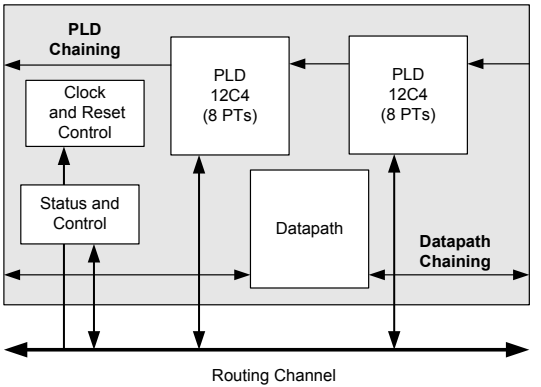
Features

- **Low power 1.71 to 5.5 V operation** *alimentation*
  - 20-nA Stop mode with GPIO pin wakeup
  - Hibernate and Deep-Sleep modes allow wakeup-time versus power trade-offs
- **Capacitive sensing**
  - Cypress Capacitive Sigma-Delta (CSD) provides best-in-class SNR (greater than 5:1) and water tolerance
  - Cypress-supplied software component makes capacitive sensing design easy
  - Automatic hardware tuning (SmartSenseTM)
- **Serial communication** *communication avec l'extérieur*
  - Two independent run-time reconfigurable serial communication blocks (SCBs) with reconfigurable I2C, SPI, or UART functionality
- **Timing and pulse-width modulation** *circuit de contrôle de moteur/LED*
  - Four 16-bit Timer/Counter Pulse-Width Modulator (TCPWM) blocks
  - Center-aligned, Edge, and Pseudo-random modes
  - Comparator-based triggering of Kill signals for motor drive and other high-reliability digital logic applications
- **Up to 36 programmable GPIOs** *des broches d'E/S*
  - 44-pin TQFP, 40-pin QFN, and 28-pin SSOP packages
  - Any GPIO pin can be CapSense, LCD, analog, or digital
  - Drive modes, strengths, and slew rates are programmable
- **PSoC Creator design environment**
  - Integrated development environment (IDE) provides schematic design entry and build (with analog and digital automatic routing)
  - Applications Programming Interface (API) component for all fixed-function and programmable peripherals
- **Industry-standard tool compatibility**
  - After schematic entry, development can be done with Arm-based industry-standard development tools

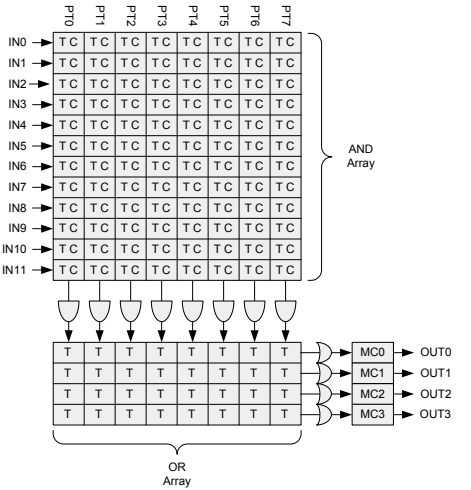
D'après la documentation

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



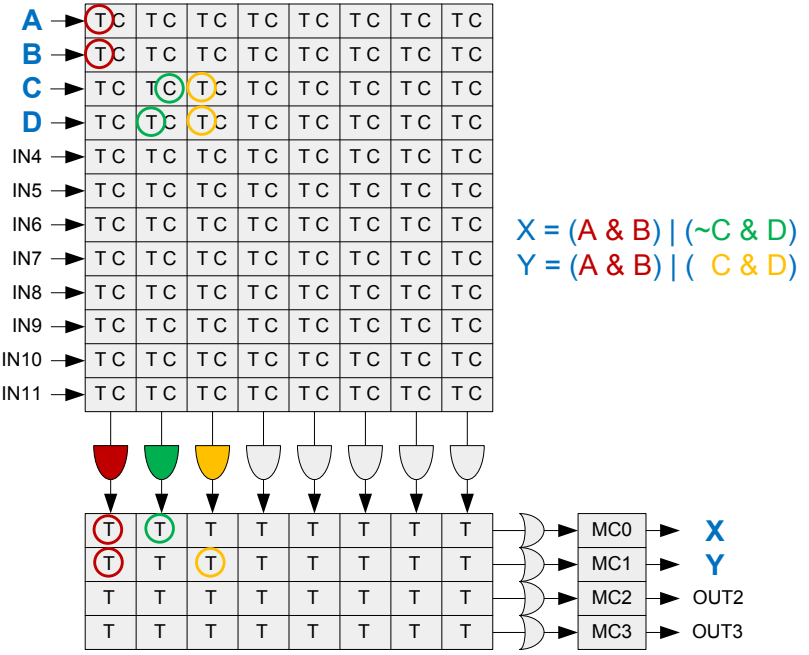
PLD



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.

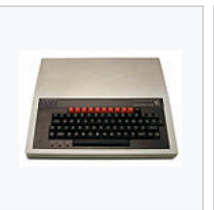
Mais ça marche comment un processeur ?

exemple le processeur 6502

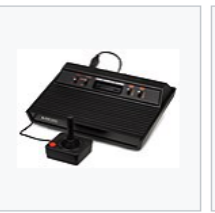
- processeur développé par Chuck Peddle pour la société MOS Technology ;
- introduit en 1975 ;
- très populaire :

Apple IIe

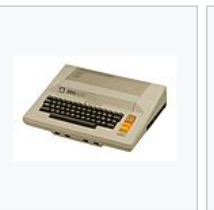
Commodore PET



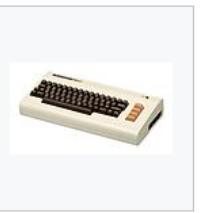
BBC Micro



Atari 2600



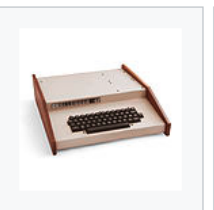
Atari 800



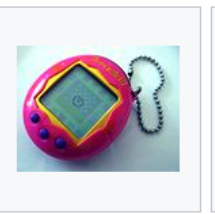
Commodore VIC-20

Commodore 64

Family Computer



Ohio Scientific Challenger 4P



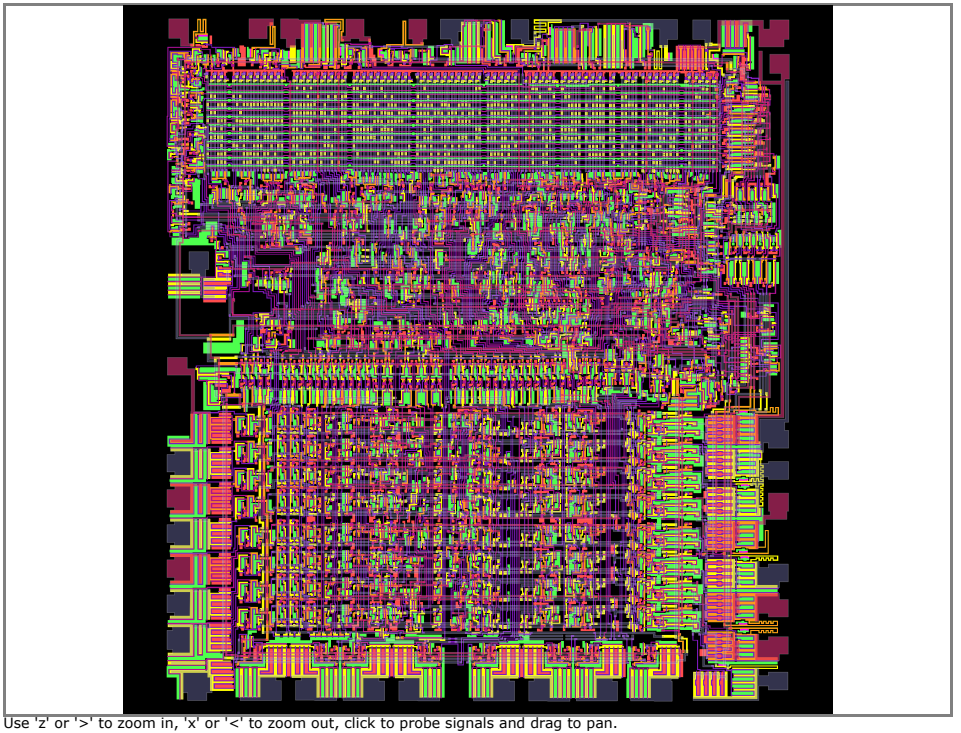
Tamagotchi digital pet[53]

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

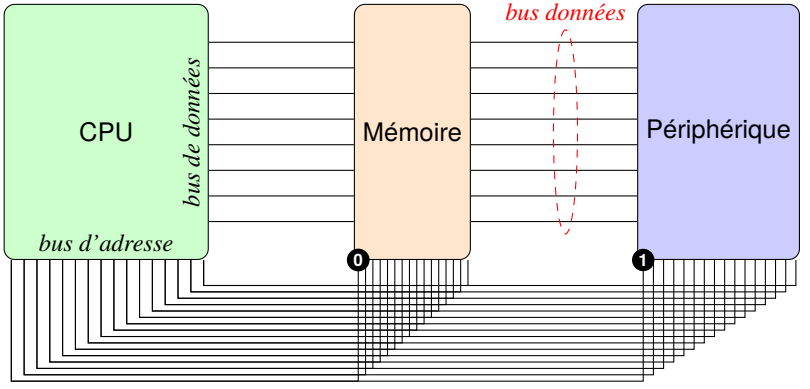
FAQ Blog Links Source easy6502 assembler mass:werk disassembler



Use 'z' or '>' to zoom in, 'x' or '<' to zoom out, click to probe signals and drag to pan.
Show: (diffusion) (grounded diffusion) (powered diffusion) (polysilicon) (metal) (protection)
Find: Clear Highlighting Animate during simulation:
Hide Chip Layout Link to this location

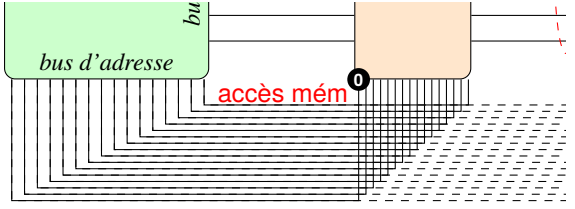
```
halfcyc:372 phi0:0 AB:0015 D:69 RnW:1
PC:0015 A:12 X:07 Y:f9 SP:fb nv-BdIzc
Hz: 3.3 Exec: SEC(T0+T2)

0000: a9 00 20 10 00 4c 02 00 00 00 00 0
0010: e8 88 e6 0f 38 69 02 60 00 00 00 0
0020: 00 00 00 00 00 00 00 00 00 00 00 0
0030: 00 00 00 00 00 00 00 00 00 00 00 0
0040: 00 00 00 00 00 00 00 00 00 00 00 0
0050: 00 00 00 00 00 00 00 00 00 00 00 0
0060: 00 00 00 00 00 00 00 00 00 00 00 0
0070: 00 00 00 00 00 00 00 00 00 00 00 0
0080: 00 00 00 00 00 00 00 00 00 00 00 0
0090: 00 00 00 00 00 00 00 00 00 00 00 0
00a0: 00 00 00 00 00 00 00 00 00 00 00 0
00b0: 00 00 00 00 00 00 00 00 00 00 00 0
00c0: 00 00 00 00 00 00 00 00 00 00 00 0
00d0: 00 00 00 00 00 00 00 00 00 00 00 0
00e0: 00 00 00 00 00 00 00 00 00 00 00 0
00f0: 00 00 00 00 00 00 00 00 00 00 00 0
0100: 00 00 00 00 00 00 00 00 00 00 00 0
0110: 00 00 00 00 00 00 00 00 00 00 00 0
0120: 00 00 00 00 00 00 00 00 00 00 00 0
0130: 00 00 00 00 00 00 00 00 00 00 00 0
0140: 00 00 00 00 00 00 00 00 00 00 00 0
0150: 00 00 00 00 00 00 00 00 00 00 00 0
0160: 00 00 00 00 00 00 00 00 00 00 00 0
0170: 00 00 00 00 00 00 00 00 00 00 00 0
0180: 00 00 00 00 00 00 00 00 00 00 00 0
0190: 00 00 00 00 00 00 00 00 00 00 00 0
01a0: 00 00 00 00 00 00 00 00 00 00 00 0
01b0: 00 00 00 00 00 00 00 00 00 00 00 0
01c0: 00 00 00 00 00 00 00 00 00 00 00 0

@AABBCCDDEEFFG

Trace more  Trace less  Trace these too:
Clear Log

cycle ab db rw Fetch pc a x y s p
```

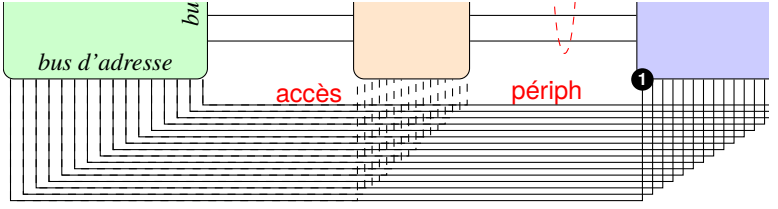
http://www.visual6502.org/JSSim/expert.html



Pour accéder à :

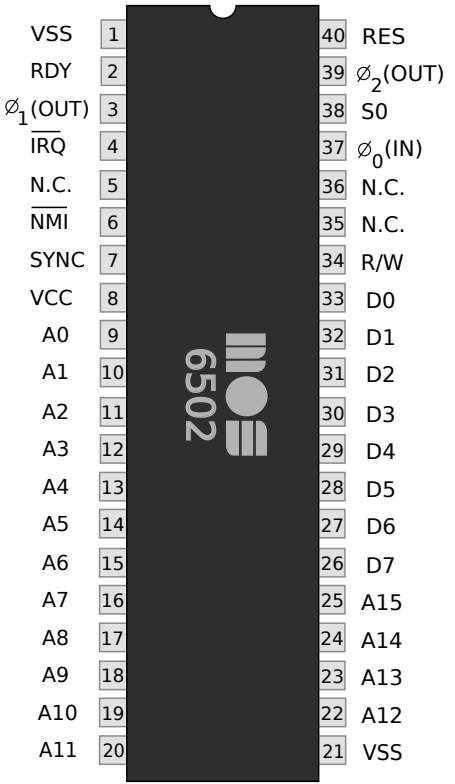
- ▷ la **mémoire** :
  - ◇ on va de l'adresse $(\mathbf{0}000\ 0000\ 0000\ 0000)_2$ à l'adresse $(\mathbf{0}111\ 1111\ 1111\ 1111)_2$;
  - ◇ ce qui donne de l'adresse $(00\ 00)_{16}$ à l'adresse $(7F\ FF)_{16}$;





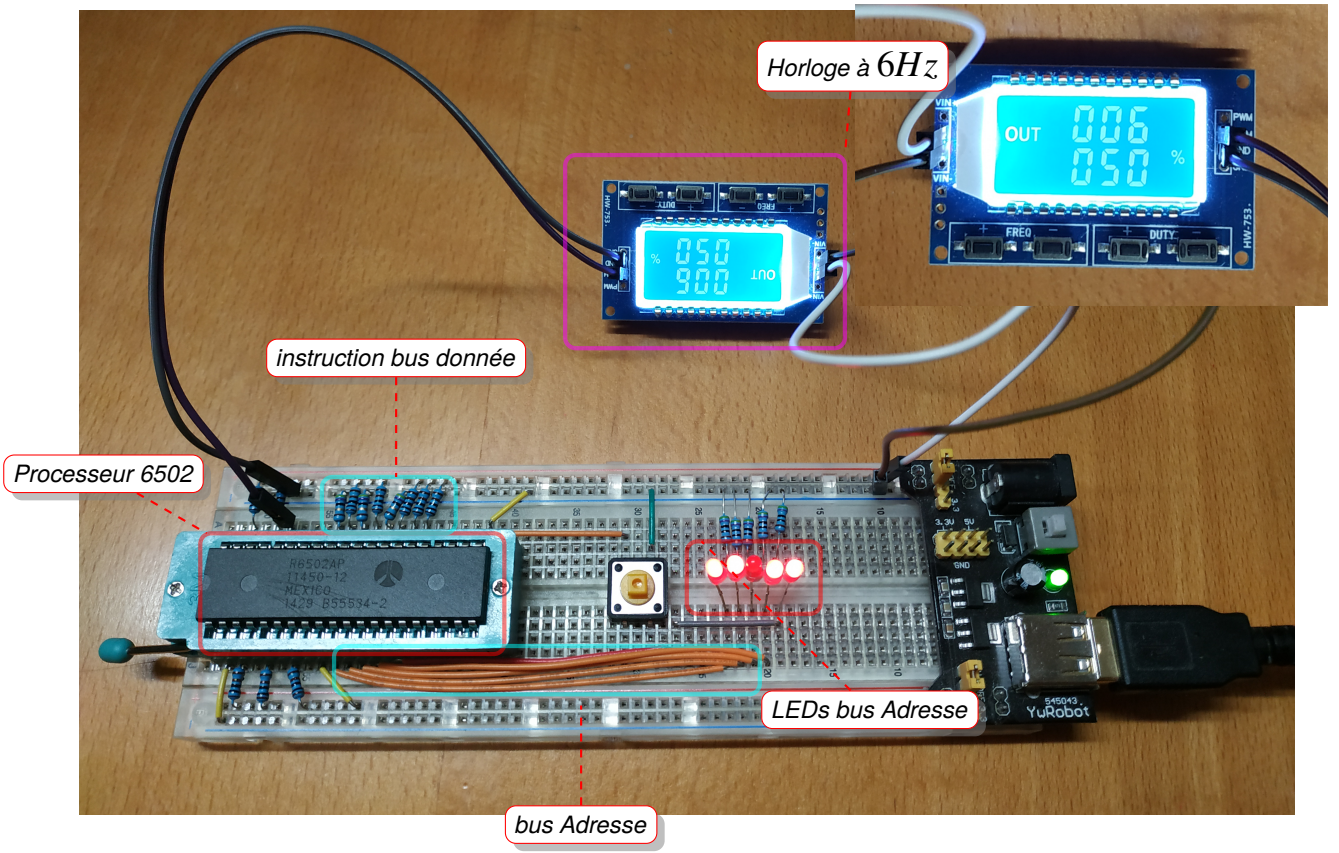
- ▷ aux **périphériques** :
  - ◇ on va de l'adresse $(\mathbf{1}000\ 0000\ 0000\ 0000)_2$ à l'adresse $(\mathbf{1}111\ 1111\ 1111\ 1111)_2$;
  - ◇ ce qui donne de l'adresse $(80\ 00)_{16}$ à l'adresse $(FF\ FF)_{16}$;

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

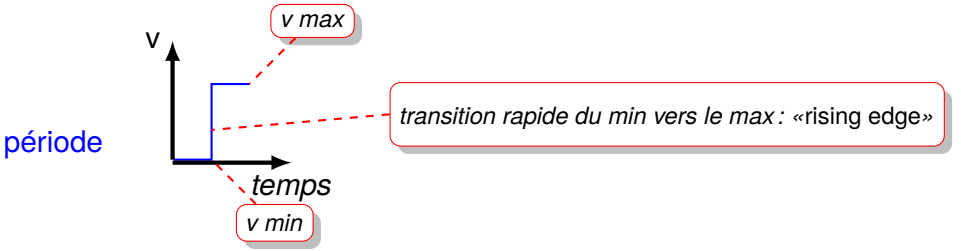


Mais c’est quoi cette horloge ?

Qu'est-ce que l'horloge ?

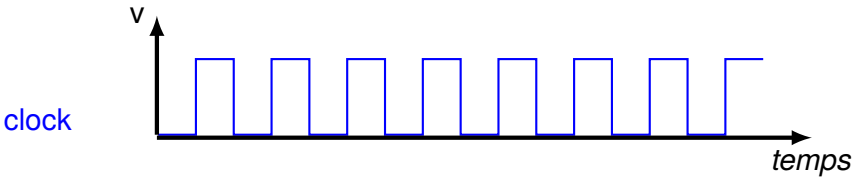
Un **signal** électrique :

- 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 :



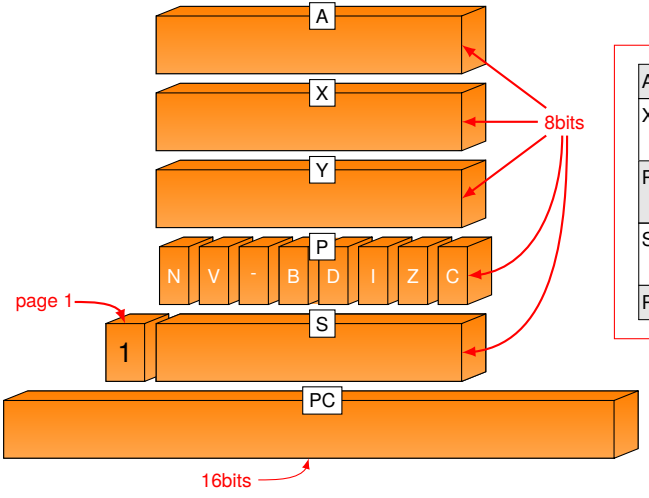
où le temps passé avec le voltage maximal est égal au temps passé avec le voltage minimal.

- qui se reproduit à l'identique :



- qui sert de **référence globale** dans le circuit et permet de **synchroniser** les différentes parties de ce circuit ;
- on utilise le front montant ou «*rising edge*» pour effectuer cette **synchronisation**.

Et la programmation d’un processeur ?

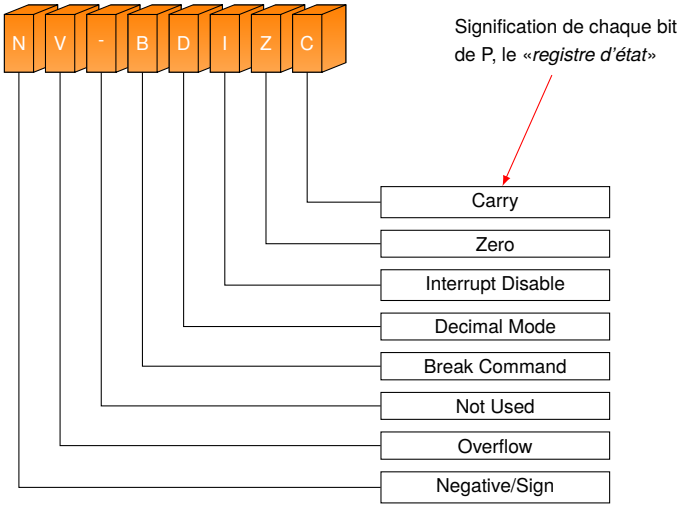


Registres

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

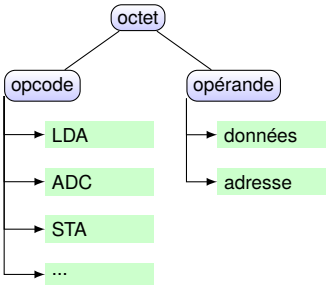
Chaque instruction est **codée sur un octet** en fonction du mode choisi.

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

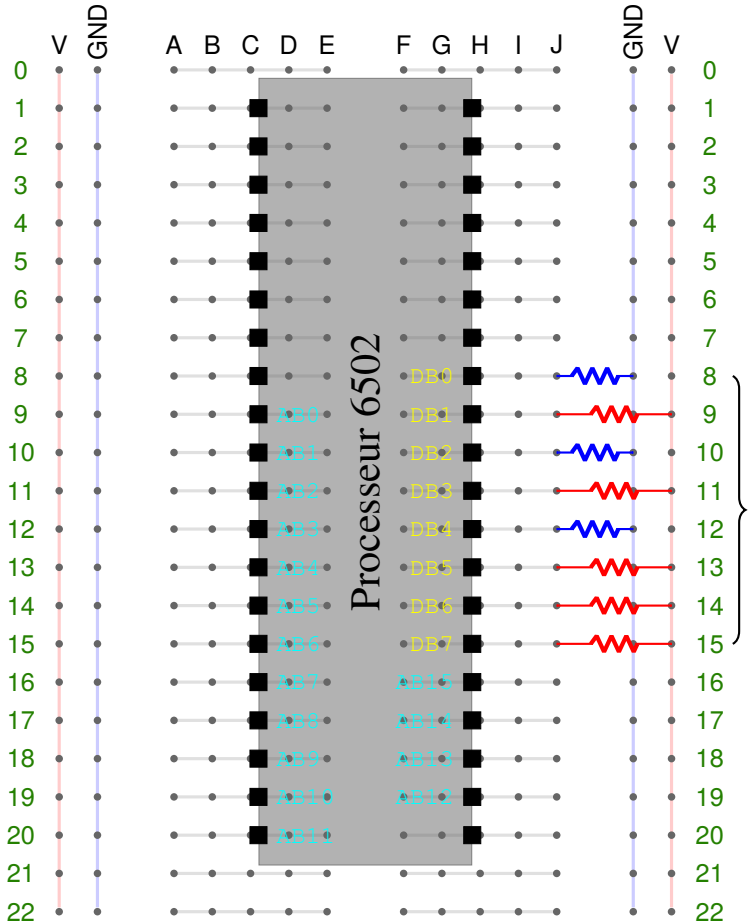
`69 01` *signifie additionner la valeur 1 dans l'accumulateur.*

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 :



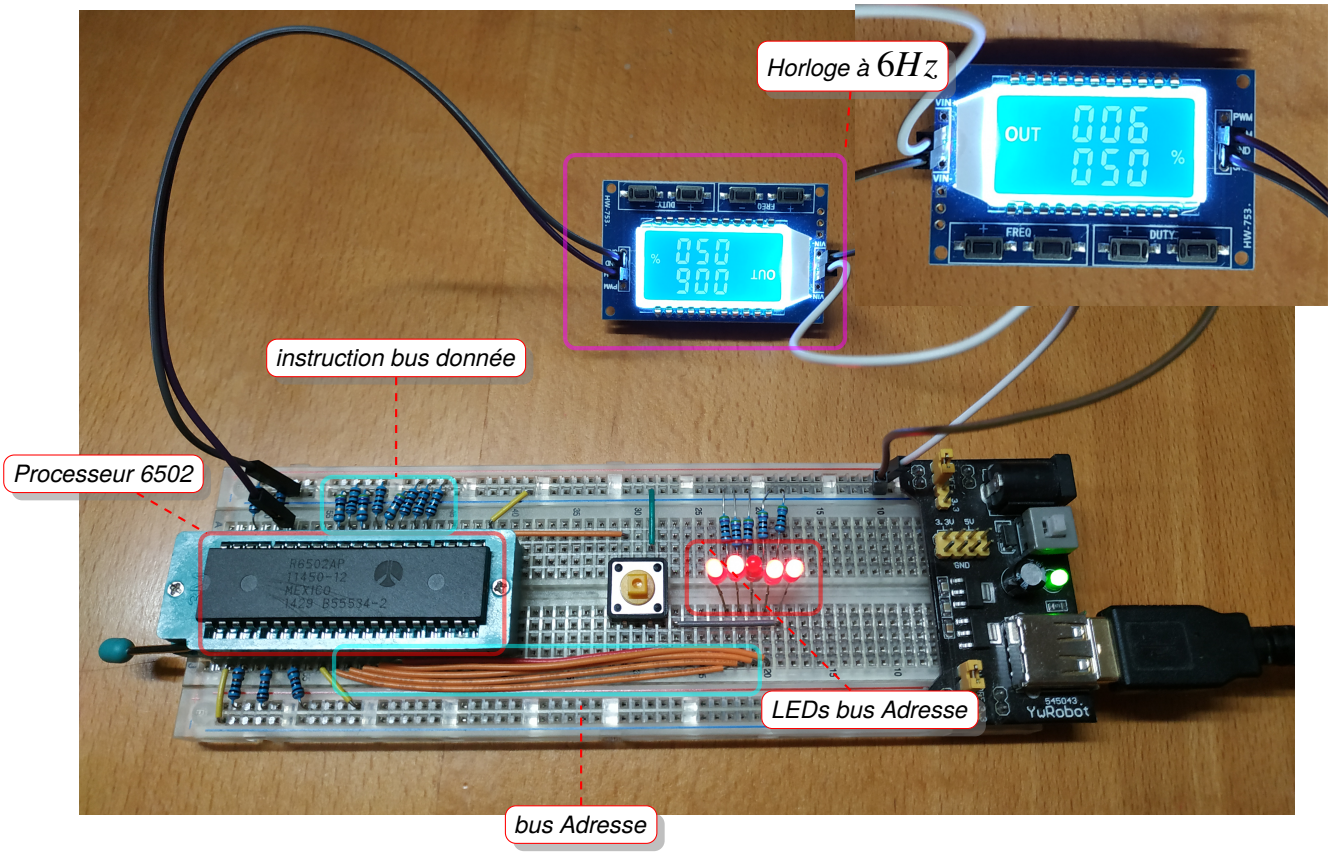
Ins	description	mode adressage						
		immédiat	absolu	page zéro	indexé X	indexé Y	implicite	relatif
ADC	ajoute un octet avec le bit de retenu dans l'accumulateur	69	6D	65	7D	79		
BEQ	«*Branch if EQual*», saut vers une adresse si vrai							F0
BNE	«*Branch if Not Equal*», saut vers une adresse si faux							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`

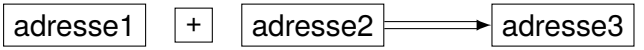
6502 sur «*breadboard*» : un processeur, une horloge et deux bus



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

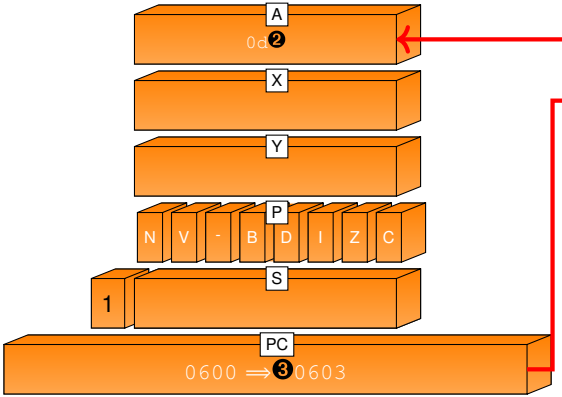
```
Address   Hexdump    Dissassembly
-------------------------------
$0600     ad 2e 06   LDA $062e
$0603     8d 00 02   STA $0200
$0606     69 01      ADC #$01
$0608     85 00      STA $00
$060a     ad 3c 06   LDA $063c
$060d     69 01      ADC #$01
$060f     85 01      STA $01
$0611     a2 01      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`

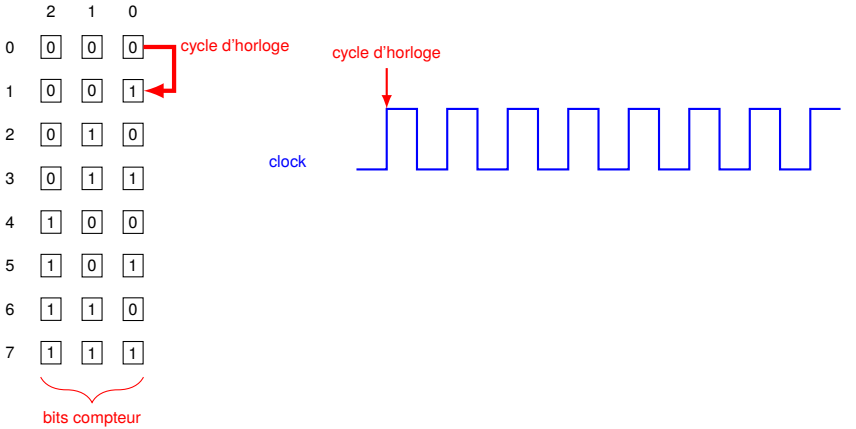
⟹ l'adresse `062e` contient l'octet `0d` qui va être chargé dans le registre A❷ ;

⟹ 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.

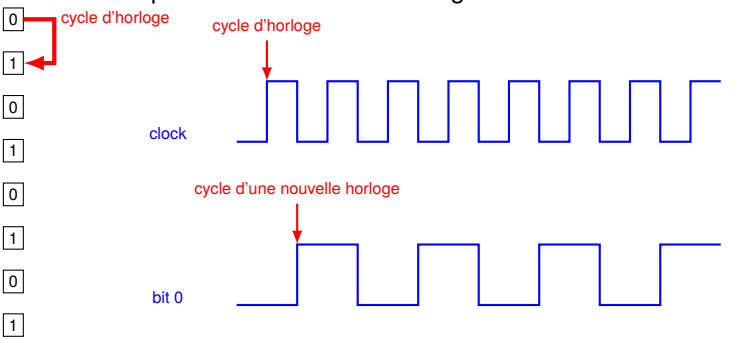
Combinaison logique & processeur : le «*timer*»

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.*)

Que peut-on faire avec un compteur/timer ?

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 !***

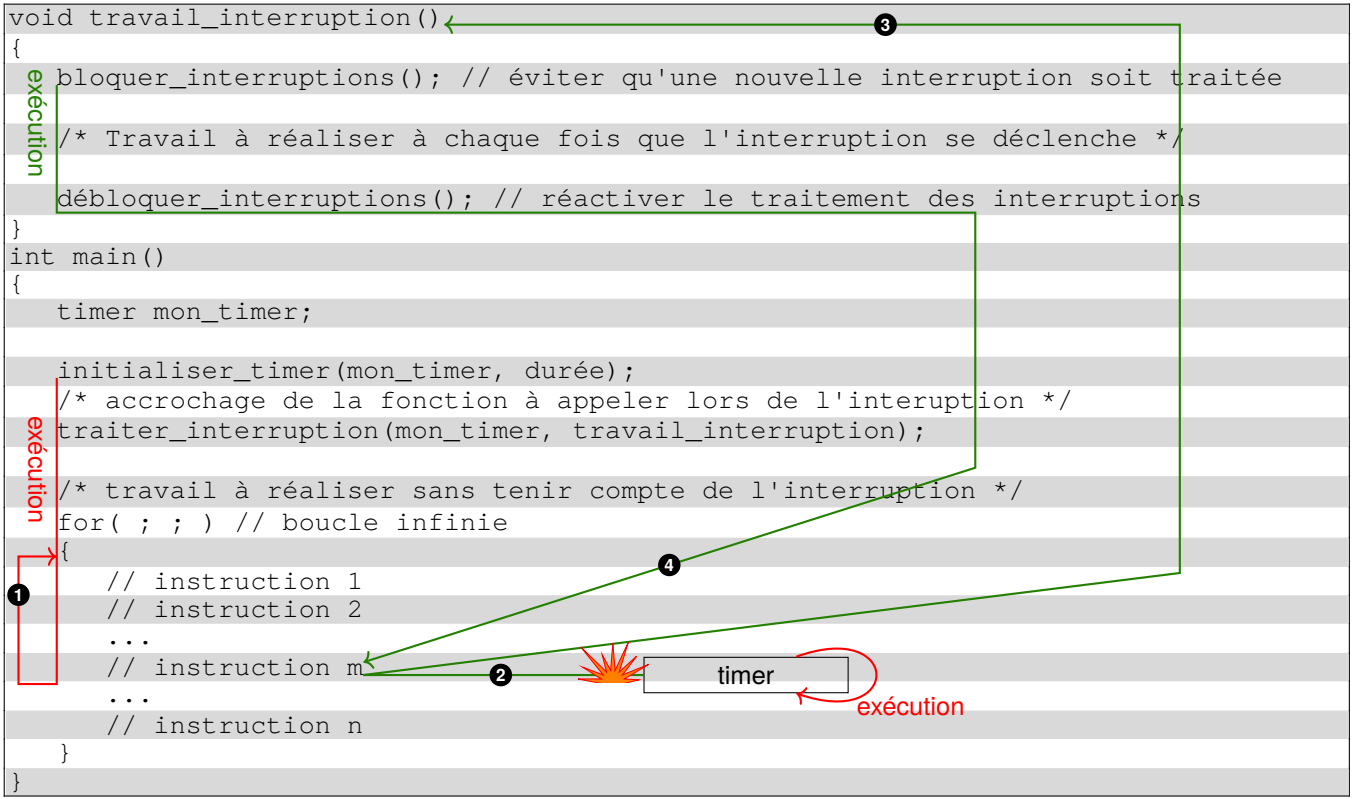
- ▷ utiliser un «*timer*» ou **compteur matériel** :
  - ◇ c'est un circuit **indépendant** du processeur ;
  - ◇ il peut être de grande dimension comme par exemple sur 32bits ;
  - ◇ il compte suivant les cycles de l'horloges qu'il reçoit comme le processeur ;

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;

Qu’est-ce que c’est un micro-contrôleur et un «*SoC*» ?

CPU

- exécution du code ;
- tout le reste est externe : mémoire RAM d'exécution, mémoire contenant le programme ;

Micro Contrôleur

- CPU ;
- périphériques intégrés : un peu de mémoire RAM, un contrôleur d'interruptions, un timer, de l'EPROM pour contenir le programme ;

SoC, «*System-on-a-Chip*» : un «*Core*» (CPU nouvelle génération) et de **nombreux** périphériques.

CPU Core	Unité programmable
MMU	Gestion de la mémoire virtuelle nécessaire pour un OS «*High End*»
DSP	Analyse de signal
Power Consumption	Batterie, génération de chaleur
Peripherals	A/D, UART, MAC, USB, Bluetooth, WiFi
Built-in RAM	vitesse et simplicité
Built-in cache	vitesse
Built-in EEPROM or FLASH	mise à jour en exploitaiton, «Field upgradeable»
JTAG Debug Suppot	Deboguage matériel
Tool-Chain	Compilateur, débogueur, ...

Différents usages

- **Application** : processeurs 32 ou 64 bits, permet de faire des calculs poussés (présence de DSPs), du multi-média, peuvent faire tourner des OS comme Linux.
- **Temps réel** : contrôle de moteurs, robotique : latence basse et sûreté de fonctionnement élevée. Adaptés à des routeurs réseau, des lecteurs multimédias où les données doivent être disponible à un instant donné ;
- **Micro-contrôleur** : gestion de matériel (fournis comme «*softcore*» dans des FPGAs), dépourvu de MMU (pas de Linux) mais intègrent de la mémoire et des périphériques.

- **Interrupt Controller** : gérer les différentes interruptions et leur priorités ;
- **DMA**, «*Direct Memory Access*» : bouger des zones mémoires indépendamment du processeur :
  - «*burst-mode*» : le circuit DMA prend le contrôle complet du bus aux dépends du CPU ;
  - «*cycle-stealing*» : négociation entre le DMA et le CPU ;
  - «*transparent*» : le DMA n'utilise le bus que lorsque le CPU ne l'utilise pas ;
- **MAC**, «*Medium Access Control*» : contrôle la couche 2 d'une interface réseau ;
- convertisseur **A/D** : numérise une valeur analogique en une valeur numérique suivant une résolution de 10 à 12 bits (avec un taux bas d'échantillonnage et un fort *jitter*.
- **UART**, «*Universal Asynchronous Receive/Transmit*» : liaison série de faible vitesse (par exemple RS232), en général de 9600 baud à 115200 baud/s avec des données sur 8bits, pas de contrôle hardware et 1 bit stop : «57600 N 8 1».
  3 fils pour relier deux appareils : le GND partagé, la broche TX de l'un reliée à la broche RX de l'autre et vice-versa.
- **USB** : liaison série haut débit, offrant différents «*Device Classes*» : périphérique HID, «*Human Interface Device*» : clavier/souris, tunnel TCP/IP, mémoire de masse, son *etc.* USB OTG, «*On The Go*», permet dávoir le rôle de maître ou de périphérique.
- **CAN**, «Controller Area Network» : bus inventé par Bosch pour les communications entre les différents circuits dans une voiture et utilisé dans les usines, entre des capteurs, *etc.*
- **WiFi** : échange continue d'information : débit élevé et données de taille quelconque mais consommateur d'énergie. l'antenne peut être externe ou incorporée dans le PCB, «printed circuit board» du circuit ;
- **Bluetooth**, BLE, «*Bluetooth Low Energy*» : échange intermittent d'information : faible débit de données réduites mais avec une très faible consommation.

- **bus** :
  - I^2C, SPI communication intelligente de données entre composants électroniques : associés à de la mémoire locale sur le périphérique : décharge le CPU de la gestion d'interruptions de composants disposant de leur propre rythme de fonctionnement (mesure de température, écran, autre CPU *etc.*) ;
  - GPIOs, «*General Purpose I/O*» : PWM, «*Pulse Width Modulation*» : contrôle de périphérique/moteur/radio (télécommande en 433Mhz, ou IR), , «*bit-banging*» : émulation de bus exotique ou connexion directe de composant (détecteur PIR de mouvement, interrupteurs *etc.*)

- **RTC**, «*Real Time Clock*» : maintenir l'heure et la date (utilisation d'une batterie séparée).
  Si le composant est «connecté» il peut utiliser un serveur NTP, «*Network Time Protocol*».

- **Timers** : compteur incrémentés ou décrémentés en fonction du temps gérés de manière indépendante du CPU
  - «*watchdog timer*» : un compteur qui doit être réinitialisé, «*kicked*», de manière logicielle avant qu'il n'atteigne zéro $\Longrightarrow$ s'il atteint zéro, le CPU subit un reset : l'idée est qu'il est dans une boucle infinie ou bien dans un interblocage ;
  - «*fast timers*» : mesurer la longueur d'impulsion ou pour les générer (PWM) ;

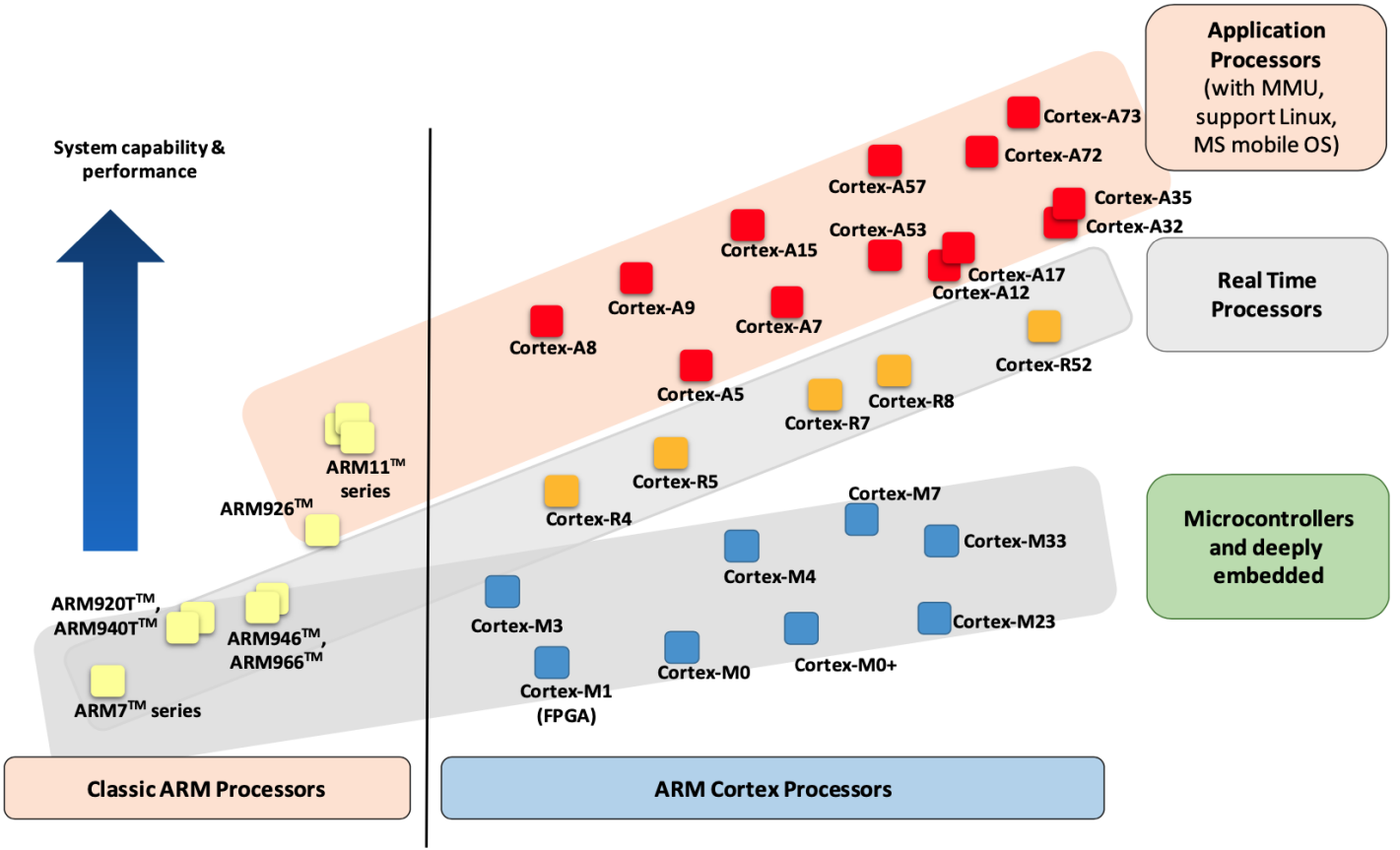
- **Memory controller** : obligatoire pour la DRAM, «dynamic RAM» : rafraîchissement de la mémoire de manière régulière (souvent intégré au CPU). Gérer la mémoire FLASH persistente.

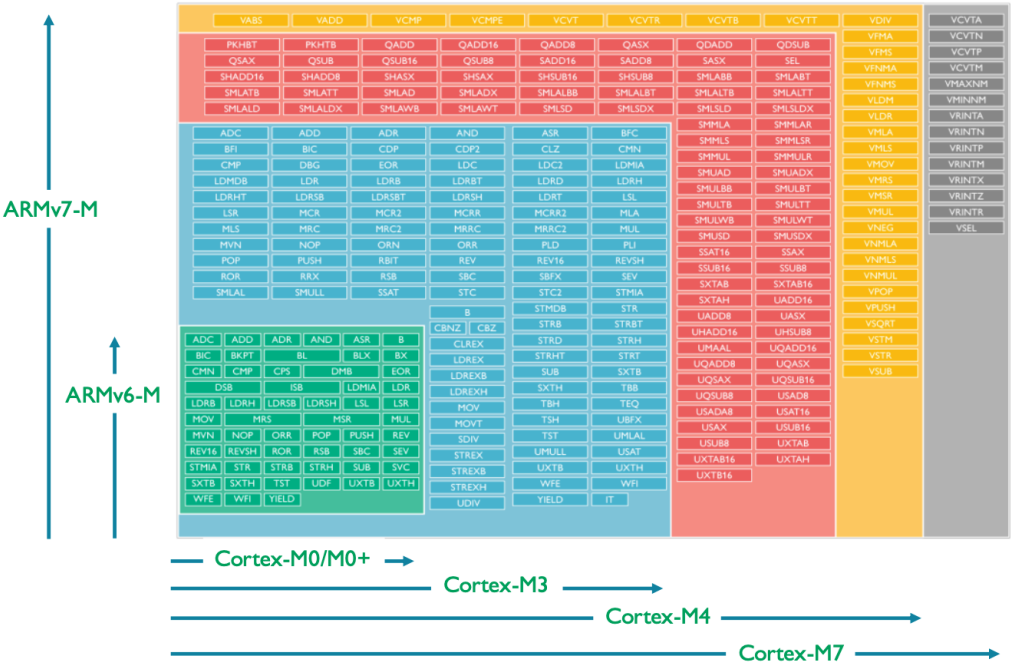
- **co-processeur cryptographique** : réaliser des opérations de chiffrement/déchiffrement et signature avec des algorithmes symétriques et surtout asymétriques (coutêux pour le CPU).
  Embarque des clés de chiffrement qui peuvent être figées dans sa mémoire
  (exemple ATECC608 : propose du chiffrement sur courbe elliptique).

- système de **localisation satellitaire** : GPS américain, Glonass russe, Beidou chinois et Galileo européen.
  Permet de disposer de la position et de l'heure à une précision de 50 ns.

Qu’apporte l’achitecture ARM ?

ARM Cortex et ISA





Floating Point

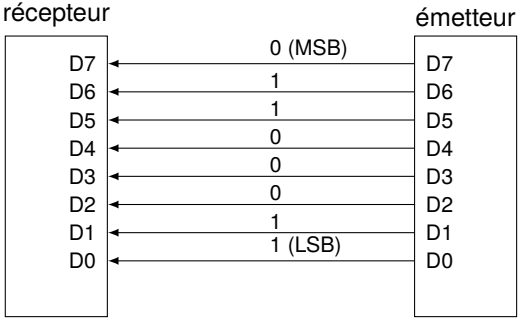
DSP (SIMD, fast MAC)

Advanced data processing
bit field manipulations

General data processing
I/O control tasks

Et les communications vers l'extérieur ?

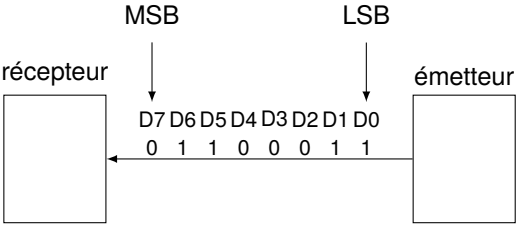
Transmission parallèle



- 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.

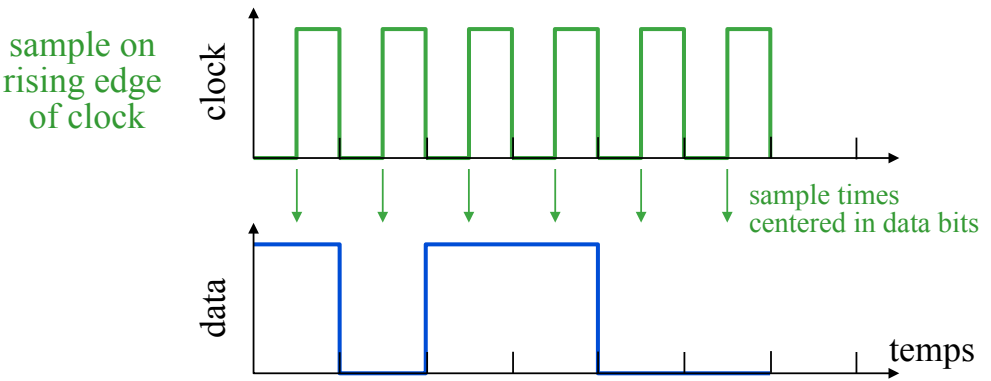
Inconvénient :

▷ 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

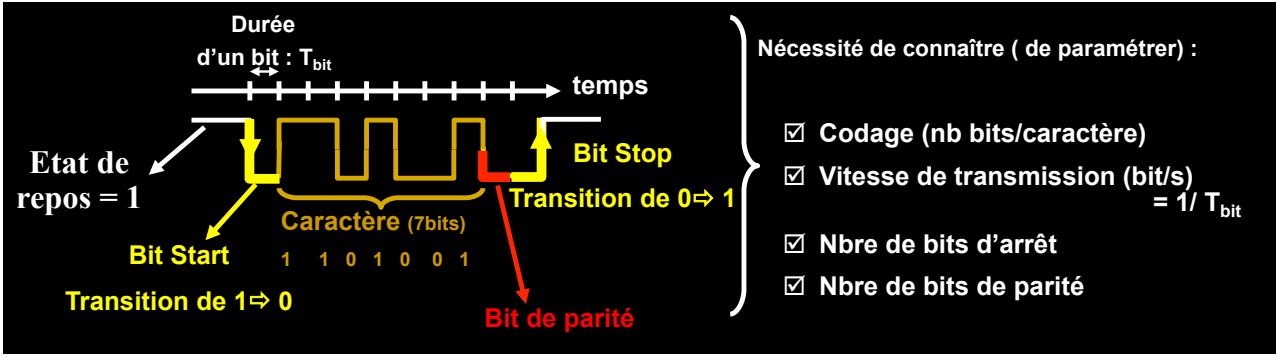
L'émetteur et le récepteur ne sont pas *synchronisés*.
Le récepteur doit détecter des **transitions** au sein des données reçues.

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



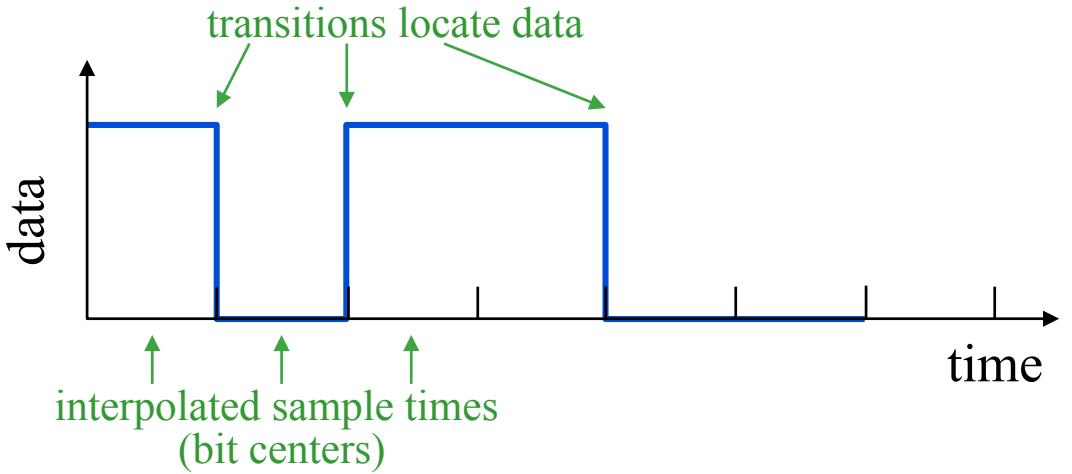
Ici, le codage consiste à passer d'une tension à l'autre seulement si on veut transmettre un bit de valeur différente.

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** :

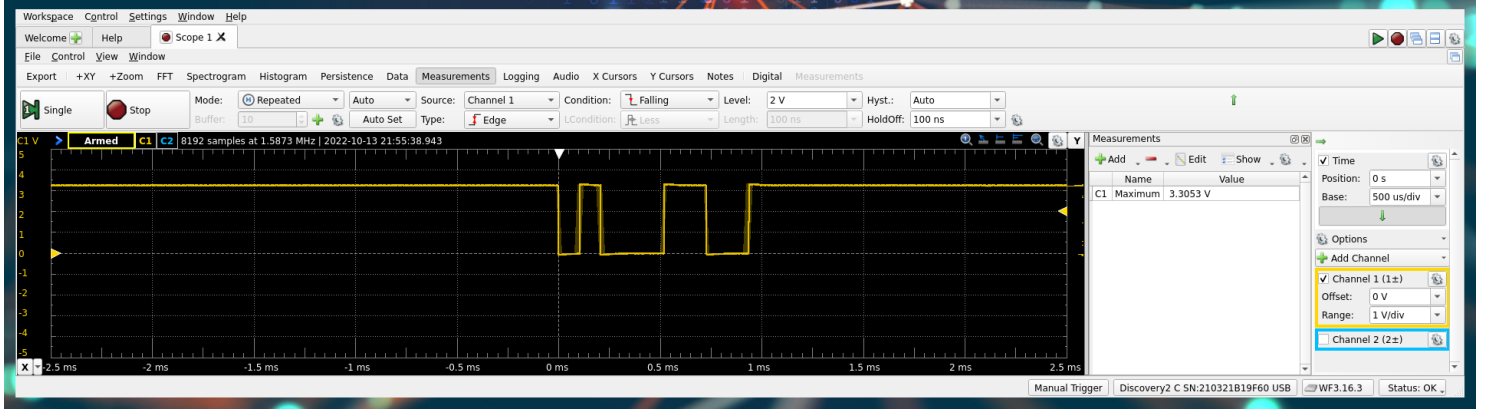
- ▷ **connaît le débit** de transmission ;
- ▷ **recherche des transitions** pour se synchroniser et interpoler des mesures d'échantillonnage...
- ▷ **extrait l'horloge** des données :



Et concrètement le port série ça donne quoi ?

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

```
MPY: soft reboot                                                    [22/8976]
MicroPython v1.19.1 on 2022-10-11; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from machine import Pin, Timer, UART
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
>>> def envoi(timer):
...     uart.write(b'1')
...
...
...
>>> timer = Timer()
>>> timer.init(freq=.5, mode=Timer.PERIODIC, callback = envoi)
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```



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** :

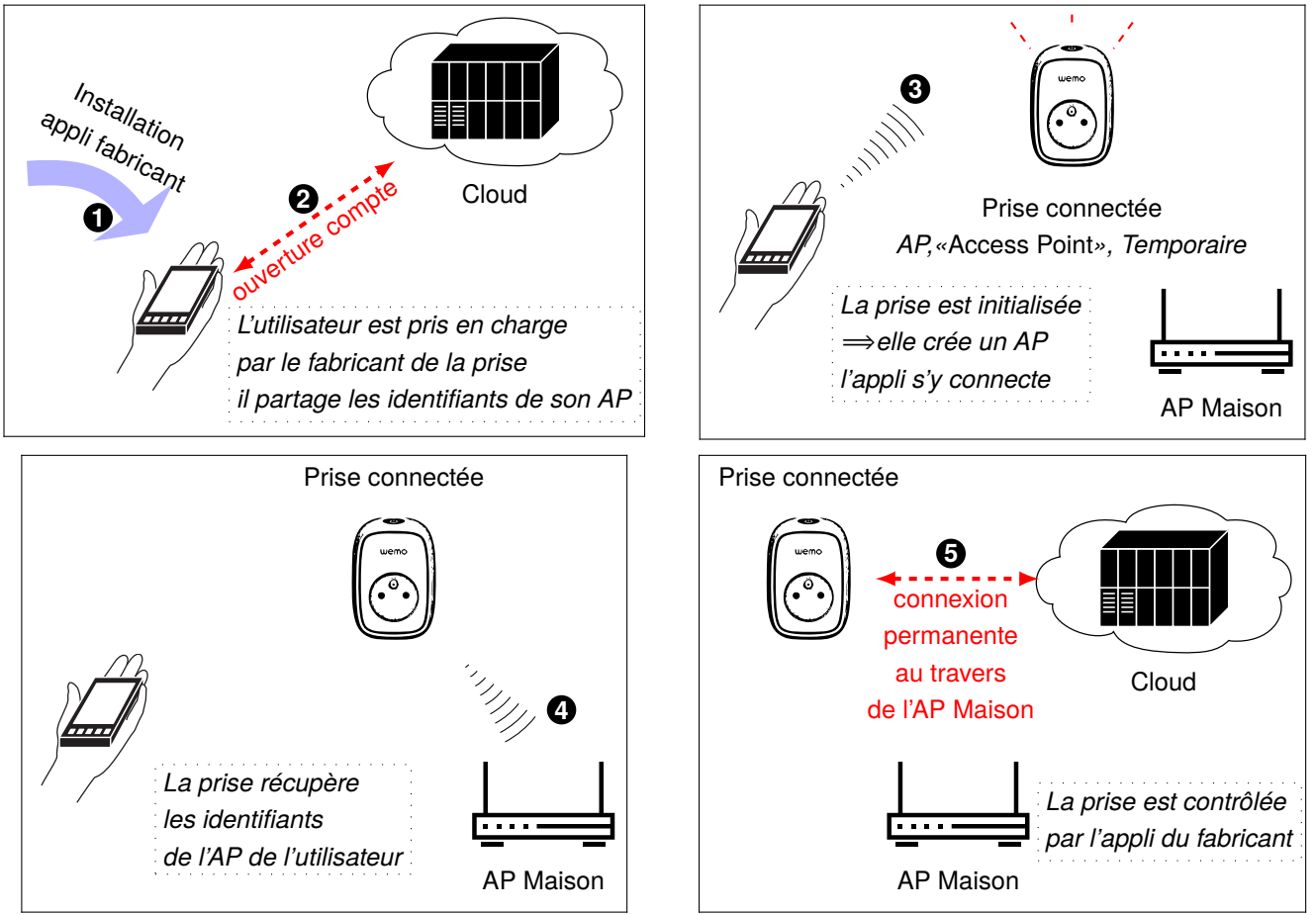
Ici, on voit que $1/\Delta X = 9,6KHz$ ce qui est le cas : on a configuré le port série pour une transmission à 9600bps :

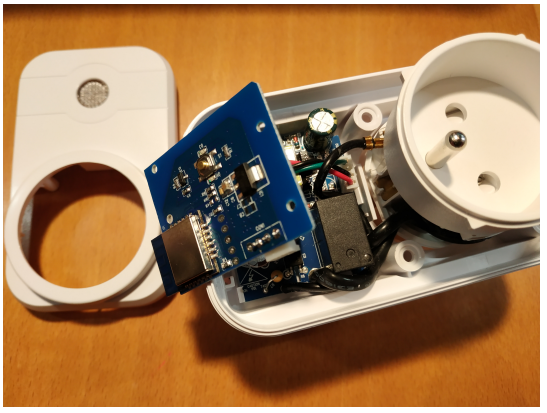
xterm

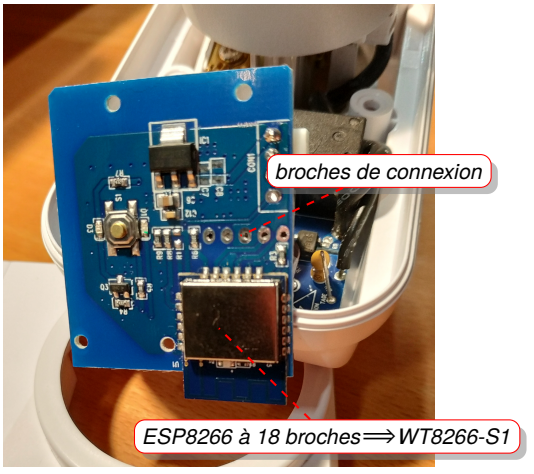
```
...
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
...
```

Importance du «*port série*» ou UART «*Universal Asynchronous Receiver/Transmitter*»

Scénario de l'intégration d'une prise connectée à la maison

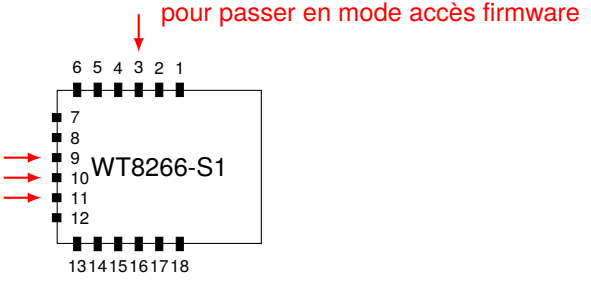






Identifier les broches du WT8266-S1 à l'aide d'un multimètre et de la doc constructeur

Pin	Info
3	IO0
9	URXD
10	GND
11	UTXD



Pour récupérer le firmware présent dans la mémoire flash de 16Mb ou 2MB avec un adaptateur USB/série :

xterm
```
$ esptool.py --port /dev/ttyUSB0 read_flash 0x00000 0x200000 tuya.bin
```

Une fois le firmware récupéré, on peut regarder ce qu'il contient :

xterm

```
$ strings tuya.bin | less
http://a.gw.tuyaeu.com/gw.json
http://a.gw.tuyaus.com/gw.json
...
mq.gw.airtakeapp.com
mq.gw1.airtakeapp.com
...
mqtt_client.c
...
ESP.ty_ws_mod.dev_ap_ssid_key={"ap_ssid":"SmartLife","ap_pwd":null}
ESP.ty_ws_mod.gw_active_key={"token":null,"key":null,"lo
cal_key":null,"http_url":null,"mq_url":null,"mq_url_bak":null,"timeZone":null,"re
gion":null,"reg_key":null,"wxappid":null,"uid_acl":null}
ESP.ty_ws_mod.dev_if_rec_key={"id":"04200063b4e62d00468a","sw_ver":"1.0.0","sche
ma_id":null,"etag":null,"product_key":"n8iVBAPLFKAAAszH","ability":0,"bind":false,"sync":false}
ESP.ty_ws_mod.wf_nw_rec_key={"ssid":null,"passwd":null,"wk_mode":0,"mode":0,"type":0,"source":0,
"path":0,"time":0,"random":0}
ESP.ty_ws_mod.gw_sw_ver_key={"sw_ver":"1.0.0","bs_ver":"5.06","pt_ver":"2.1"}
:ESP.device_mod.dp_data_key={"relay_switch":false,"switch":true,"work_mode":1,"bright":180,"tem
per":255,"colour_data":"1900000000ff19","scene_data":"00ff0000000000","rouguang_scene_da
ta":"ffff500100ff00","binfeng_scene_data":"ffff8003ff000000ff000000ff00000000000000000","xuan
cai_scene_data":"ffff5001ff0000","banlan_scene_da
ta":"ffff0505ff000000ff00ffff00ff00ff0000ff000000"}
ESP.device_mod.fsw_cnt_key={"fsw_cnt_key":0}
ESP.device_mod.appt_posix_key={"appt_posix":0}
ESP.device_mod.power_stat_key={"power":0}
{"mac":"68a","prod_idx":"04200063","auz_key":"TJJ7AGs644QGICVFovUcpeVeGz0JTbR1","prod_test":false}
```

l'URL de la requête

un FQDN

une bibliothèque utilisée

le SSID et le mot de passe non encore renseigné

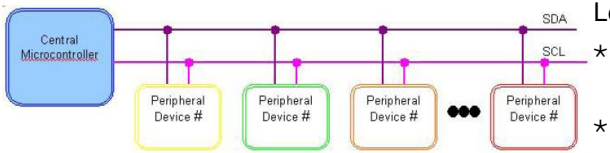
une clé produit

une clé d'API

xterm

```
$ dig +short mq.gw.airtakeapp.com
120.55.106.107
$ curl http://a.gw.tuyaus.com/gw.json
{"t":1537555117,"e":false,"success":false,"errorCode":"API_EMPTY","errorMsg":"API"}
```

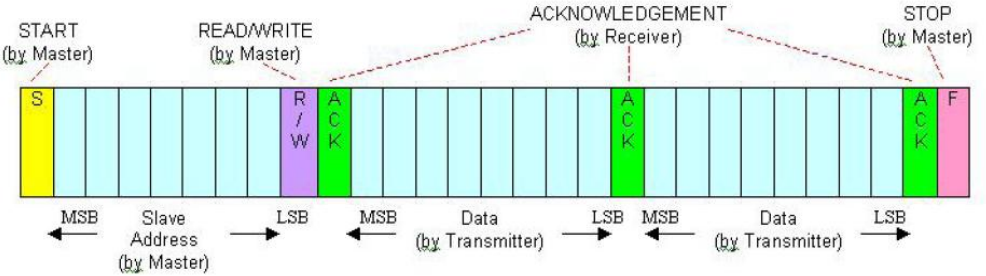
Et les communications entre les composants ?



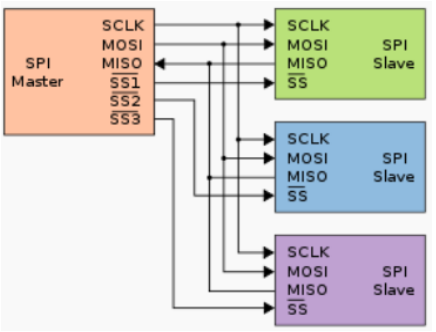
Le bus I2C, «*Inter-Integrated Circuit*» :

- ⋆ un bus générique proposé par Philips dans les années 80, beaucoup utilisé dans les télévisions ;
- ⋆ synchrone ;
- ⋆ **débit** : jusqu'à 400 Kbps ;

- ⋆ seulement 2 signaux :
  - ◇ `SCL`, «*Signal Clock*» : le contrôleur «Master» génère l'horloge ;
  - ◇ `SDA`, «*Signal Data*» : le «Master» transmets les informations et le «Slave» transmets l'acquittement : si aucun acquittement n'est reçu la communication peut être stoppée ou réinitialisée.



- ▷ plusieurs «*Slaves*» peuvent être connectés au même bus ;
- ▷ chaque *Slave* doit disposer d'une **adresse** sur 8bits, composée de :
  - ◇ une partie fixe qui dépend du constructeur ;
  - ◇ une partie configurable ;
  - ◇ le dernier bit qui définit le sens de la communication : 0 pour écrire et 1 pour lire ;
  - ◇ les communications commencent par un bit de début, «*start bit*», suivi de l'adresse sur 8 bits, le bit d'acquittement, un octet de donnée, un autre bit d'acquittement and à la fin un bit d'arrêt



- un bus générique proposé par Motorola dans les années 80 ;
- communications :
  - full duplex ;
  - synchrone ;
  - lien «Master/Slave» : c'est le master qui initie le transfer des trames de données ;
  - plusieurs liens simultanés possibles : un fil par slave permet de sélectionner celui avec lequel on veut communiquer ;
- **Débit** : quelques dizaines de Mbps ;

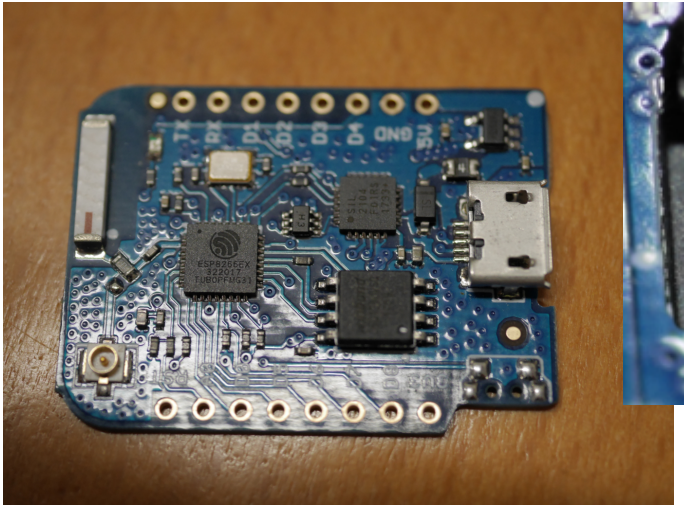
- 4 signaux :
  - `SCLK`, «*Clock*» : l'horloge obligatoire pour la transmission synchrone ;
  - `MOSI`, «*Master Out Slave Input*» : communication Master $\Longrightarrow$ Slave ;
  - `MISO`, «*Master Input Slave Output*» : communication Slave $\Longrightarrow$ Master ;
  - `SS`, «*Slave Select*» : un fil par Slave pour pouvoir le sélectionner ;

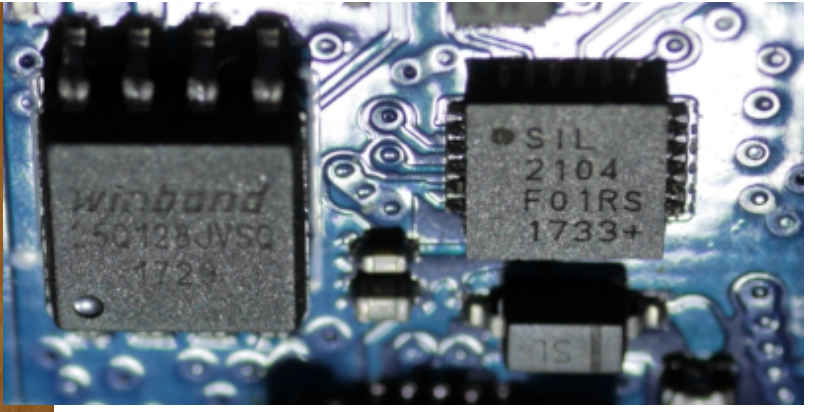
«SPI *vs* I2C» : Quel bus choisir ?

- le bus SPI permet des débits plus rapides ;
- le bus I2C ne nécessite que 2 fils **mais** nécessite un protocole de communication plus complexe : adressage, définition de trames, gestion de l'acquittement.

	SPI	I2C
Application	Better suited for data streams between processors	Occasional data transfers. Generally used for slave configuration
Data rates	>10 Mb/s	< 400 kb/s
Complexity	3 bus lines More wires more complex wiring More pins on a chip	Simple, only 2 wires Complexity does not scale up with number of devices
Addressing	Hardware (chip selection)	Built-in addressing scheme
Communication	No acknowledgment mechanism, Only for short distances	Better data integrity with collision detection, acknowledgment mechanism, spike rejection
Specification	No official specification	Existing official specifications
Licensing	free	free

Et concrètement une communication SPI ça donne quoi ?



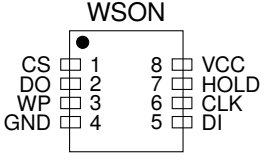


On peut lire la référence du composant mémoire :
Winbond 25Q128JVSQ

On peut récupérer la documentation à :

`https://www.winbond.com/resource-files/W25Q128JV%20RevI%2008232021%20Plus.pdf`

On obtient la description des broches du composant :

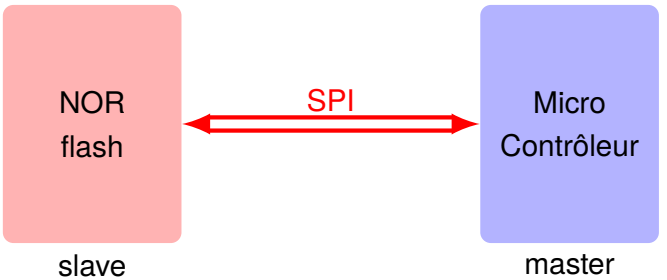


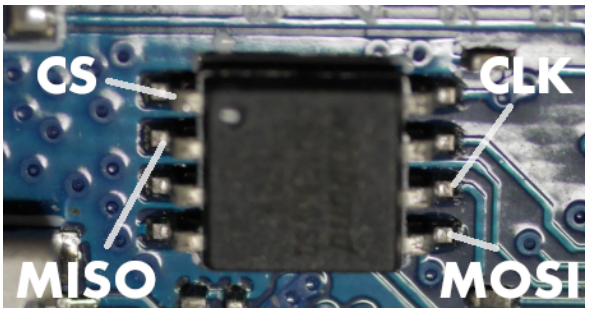
Ce qui permet de le connecter au **Bus Pirate**.

WSON	Bus Pirate
VCC	3.3v
GND	GND
CS	CS
CLK	CLK
DO	MISO
DI	MOSI

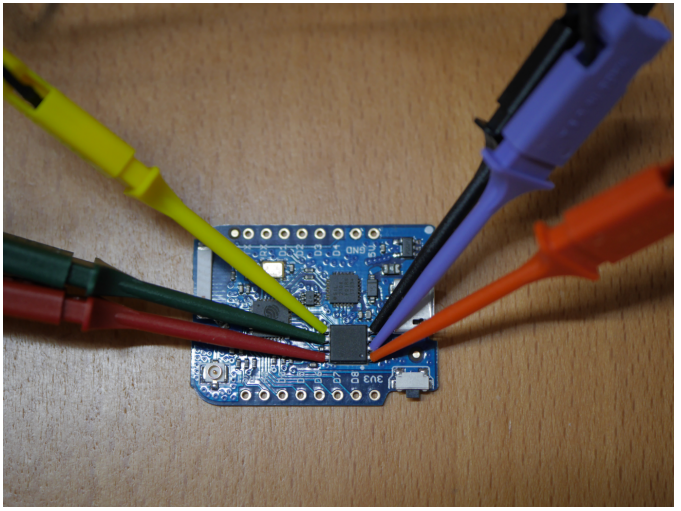
NOR Flash : mémoire accessible par SPI

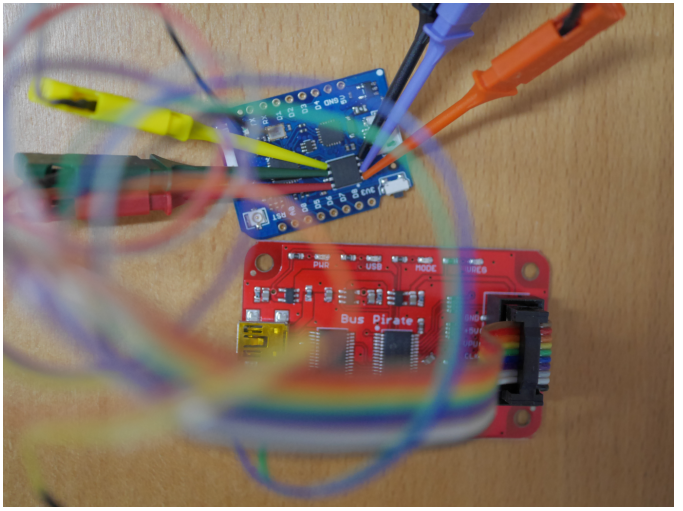
- Support de stockage pour données **non volatiles** : les données restent dans le composant jusqu'à ce qu'elles soient réécrites et elles sont conservées même si le circuit embarqué est éteint ;
- Les données peuvent être lues **octet par octet** : avec de la mémoire NAND, on ne peut lire qu'un, dix ou 100 octets à la fois : par page de 4096 ;
- Mémoire **sans défaut** : pas de système de correction d'erreur nécessaire contrairement à la mémoire de type NAND ;
- **faible latence** d'accès : utilisable pour l'exécution directe de code ;
- utilisée en général pour stocker des données en faible quantités ;
- communication par le bus SPI : communication **synchrone**, **full duplex** avec 4 signaux : CS, CLK, MISO et MOSI.



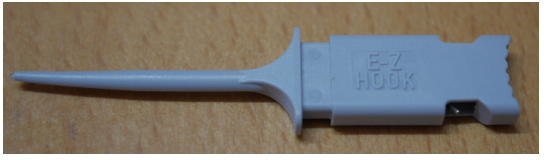


On utilise des sondes, «*probes*» pour se connecter à chaque broche du composant mémoire :





Chacune de ces sondes est reliées au **Bus Pirate**.



Le **Bus Pirate** est un circuit opensource qui permet :

- ▷ d'alimenter le composant à tester en 3,3v ou 5v ;
- ▷ de disposer d'une interface USB/série ;
- ▷ de disposer d'un **micro-contrôleur** dédié à :
  - ◇ lire des **ordres** en provenance de la machine de l'utilisateur par l'intermédiaire du port série simulé par USB ;
  - ◇ **interpréter** ces ordres avec un firmware spécialisé opensource ;
  - ◇ de gérer **différents protocoles de communication** inter-composants : I2C, SPI, 1-Wire, ... ;
  - ◇ **d'envoyer et recevoir des données** du protocole choisi sur des broches connectées au composant à tester ;
- ▷ *Ici, on va utiliser le bus SPI pour communiquer avec le composant mémoire.*

Identification du composant en utilisant le bus SPI :

8.1 **Device ID and Instruction Set Tables**

8.1.1 Manufacturer and Device Identification

MANUFACTURER ID	**(MF7 - MF0)**	
Winbond Serial Flash	EFh	
Device ID	**(ID7 - ID0)**	**(ID15 - ID0)**
Instruction	**ABh, 90h, 92h, 94h**	**9Fh**
W25Q128JV-IN/IQ/JQ	17h	4018h
W25Q128JV-IM*/JM*	17h	7018h

Note: For DTR, QPI supporting, please refer to W25Q128JV-M DTR datasheet.

8.1.2 Instruction Set Table 1 (Standard SPI Instructions)(1)

Data Input Output	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Number of Clock(1-1-1)	**8**	**8**	**8**	**8**	**8**	**8**	**8**
Write Enable	**06h**						
Volatile SR Write Enable	**50h**						
Write Disable	**04h**						
Release Power-down / ID	**ABh**	Dummy	Dummy	Dummy	(ID7-ID0)(2)		
Manufacturer/Device ID	**90h**	Dummy	Dummy	00h	(MF7-MF0)	(ID7-ID0)	
JEDEC ID	**9Fh**	(MF7-MF0)	(ID15-ID8)	(ID7-ID0)			
Read Unique ID	**4Bh**	Dummy	Dummy	Dummy	Dummy	(UID63-0)	

La commande SPI est `0x9f` et la réponse devrait être : `(MF)(ID)(ID)` où :

- `MF` vaut `EF` ;
- `ID` soit `4018` ou `7018`.

xterm

```
$ minicom
Welcome to minicom 2.8
OPTIONS: I18n
Port /dev/ttyUSB0, 17:08:40
Press CTRL-A Z for help on special keys
HiZ>m
1. HiZ
2. 1-WIRE
3. UART
4. I2C
5. SPI
6. 2WIRE
7. 3WIRE
8. KEYB
9. LCD
10. PIC
11. DIO
x. exit(without change)
(1)>5
Set speed:
 1.  30KHz
 2. 125KHz
 3. 250KHz
 4.   1MHz
 5.  50KHz
 6. 1.3MHz
 7.   2MHz
 8. 2.6MHz
 9. 3.2MHz
10.   4MHz
11. 5.3MHz
12.   8MHz
(1)>4
Clock polarity:
 1. Idle low *default
 2. Idle high
(1)>
```

xterm

```
Output clock edge:
 1. Idle to active
 2. Active to idle *default
(2)>
Input sample phase:
 1. Middle *default
 2. End
(1)>
CS:
 1. CS
 2. /CS *default
(2)>
Select output type:
 1. Open drain (H=Hi-Z, L=GND)
 2. Normal (H=3.3V, L=GND)
(1)>2
Clutch disengaged!!!
To finish setup, start up the power
supplies with command 'W'
Ready
SPI>W
POWER SUPPLIES ON
Clutch engaged!!!
SPI>[0x9f r:3]
/CS ENABLED
WRITE: 0x9F
READ: 0xEF 0x40 0x18
/CS DISABLED
SPI>
```

On :

- ▷ se connecte au **Bus Pirate** avec `minicom` ;
- ▷ envoie la commande `[0x9f r:3]` qui envoie la commande `0x9f` SPI et lit 3 octets en réponse ;
- ▷ reçoit les 3 octets `0xEF 0x40 0x18`

⟹ **le composant est bien identifié** !

Utilisation de l'outil `flashrom`

xterm
```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,spispeed=1M
flashrom v1.2 on Linux 5.15.0-47-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
No EEPROM/flash device found.
Note: flashrom can never write if the flash chip isn't found automatically.
```

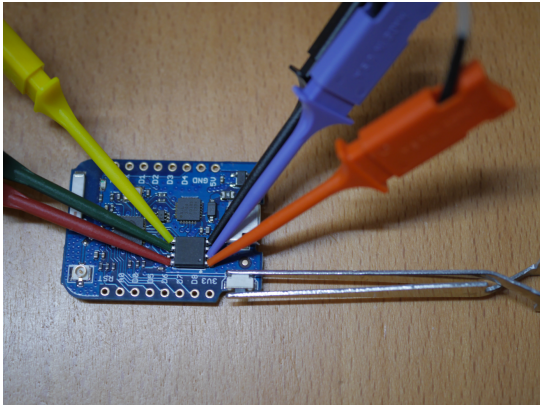
Il y a un **conflit d'accès** au composant mémoire :

- le processeur accède à la mémoire pour **exécuter le firmware** ;
- le Bus Pirate accède à la mémoire pour **l'identifier**.

$\Longrightarrow$ il faut **bloquer le processeur** en le maintenant dans l'état RESET !

*Heureusement, sur la carte de développement un **bouton RESET** permet de le faire.*

$\Longrightarrow$On va utiliser une **pince** pour le bloquer.



xterm
```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0
flashrom v1.2 on Linux 5.15.0-47-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on buspirate_spi.
No operations were specified.
```

Le composant est correctement identifié !

Récupération finale du firmware

xterm

```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0 -r dump_esp8266.bin
flashrom v1.2 on Linux 5.15.0-47-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on buspirate_spi.
Reading flash... done.
```

Contenu du firmware

xterm

```
$ strings dump_esp8266.bin
...
Access denied
WebREPL connected
>>>
Password:
wH&@wH&@SH&@#H&@#H&@;H&@wH&@
9(@_
...
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
258EAFA5-E914-47DA-95CA-C5AB0DC85B11
Sec-WebSocket-Key:
Not a websocket request
Sec-WebSocket-Key
...
Welcome to MicroPython!
For online docs please visit http://docs.micropython.org/en/latest/esp8266/ .
For diagnostic information to include in bug reports execute 'import port_diag'.
Basic WiFi configuration:
```

la commande `strings` permet d'extraire les chaînes de caractères

Peut-être un token secret...

Le firmware est basé sur microPython !

Comment fonctionne l'outil `flashrom` ?

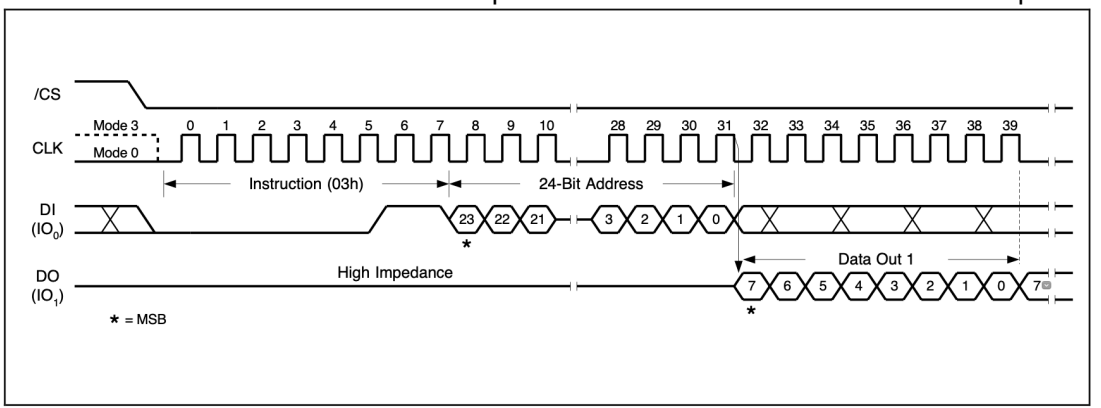
Il utilise le **Bus Pirate** pour effectuer les échanges sur le bus SPI.

Il utilise la commande de lecture `0x03` pour lire la mémoire octet par octet :

Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0)	

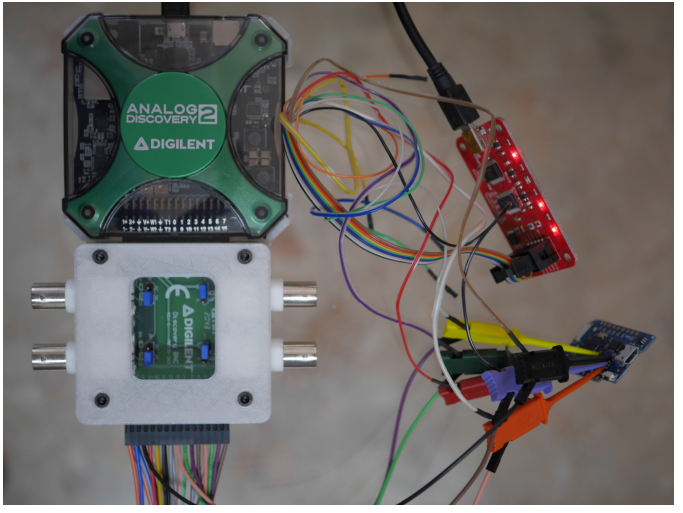
Il transmet `0x03 0x00 0x00 0x00` pour obtenir l'octet à l'adresse `0x000000`.

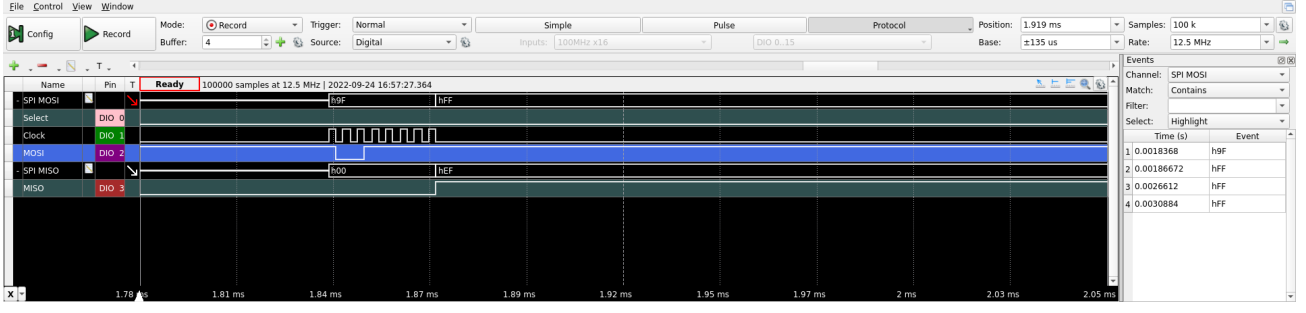
Il transmet la commande sur le bus SPI et récupère l'octet de mémoire à l'adresse indiquée :



Ici, il balaye l'ensemble des 16Mo de mémoire pour récupérer l'intégralité du firmware.

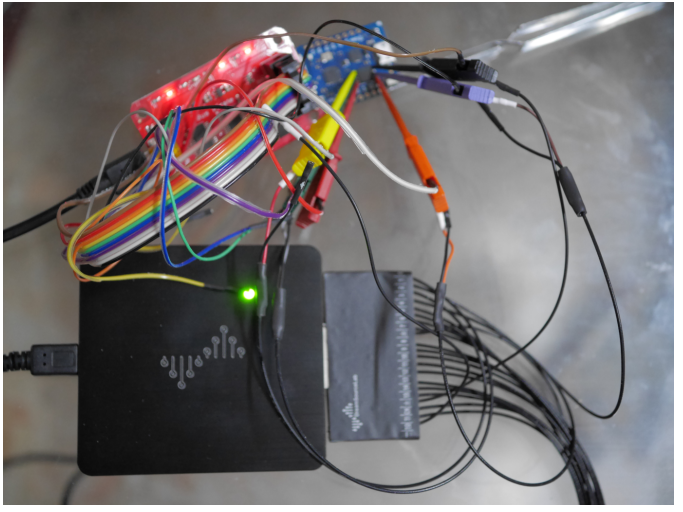
Et dans un analyseur logique ?



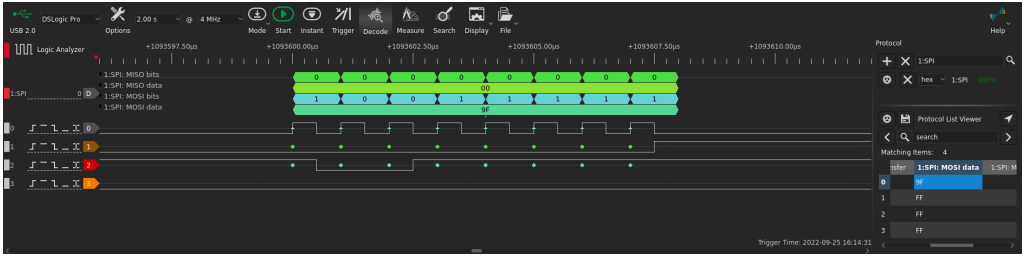


Ici on peut voir le `0x9f` transmis sur la broche MOSI et durant la réponse du composant, le master transmets `0xff`.

Et dans un autre analyseur logique ?

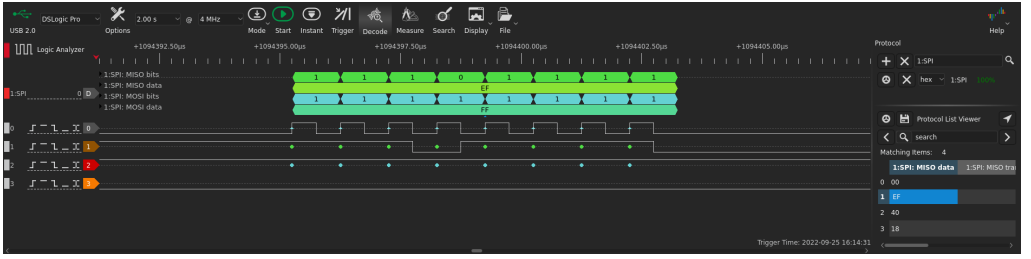


Ici on peut voir le `0x9f` transmis sur la broche MOSI :

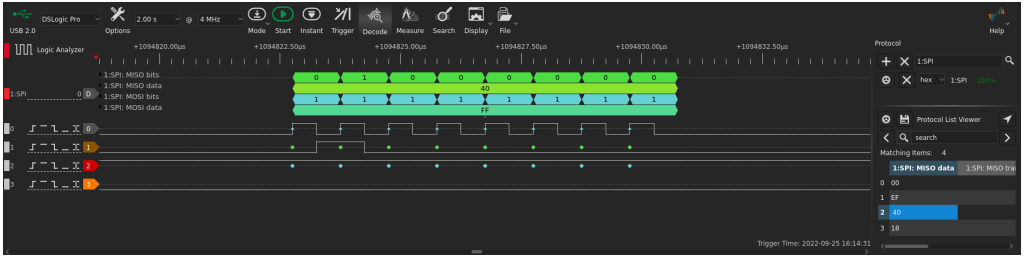


La réponse du composant mémoire :

- le `0xEF` :



- le `0x40` :



- et enfin le `0x18` :