



# FPGA

Créer du matériel en programmant

FPGA, créer du matériel en programmant.

Introduction aux FPGA à destination des développeurs et développeuses,  
par Frédéric BISSON en 2019.

# SOMMAIRE

- Il était une fois : les CPU
- C'est quoi un FPGA ?
- Comment programmer un FPGA ?
- Un exemple : SimpleVGA
- Déboguer un circuit
- Et maintenant ?

Cette présentation débute par un retour sur l'architecture de base des processeurs.

Elle donne ensuite quelque bases sur les FPGA et comment ils se programment, le tout illustré par un exemple basique de programmation d'un FPGA : la génération d'un signal vidéo à la norme VGA.

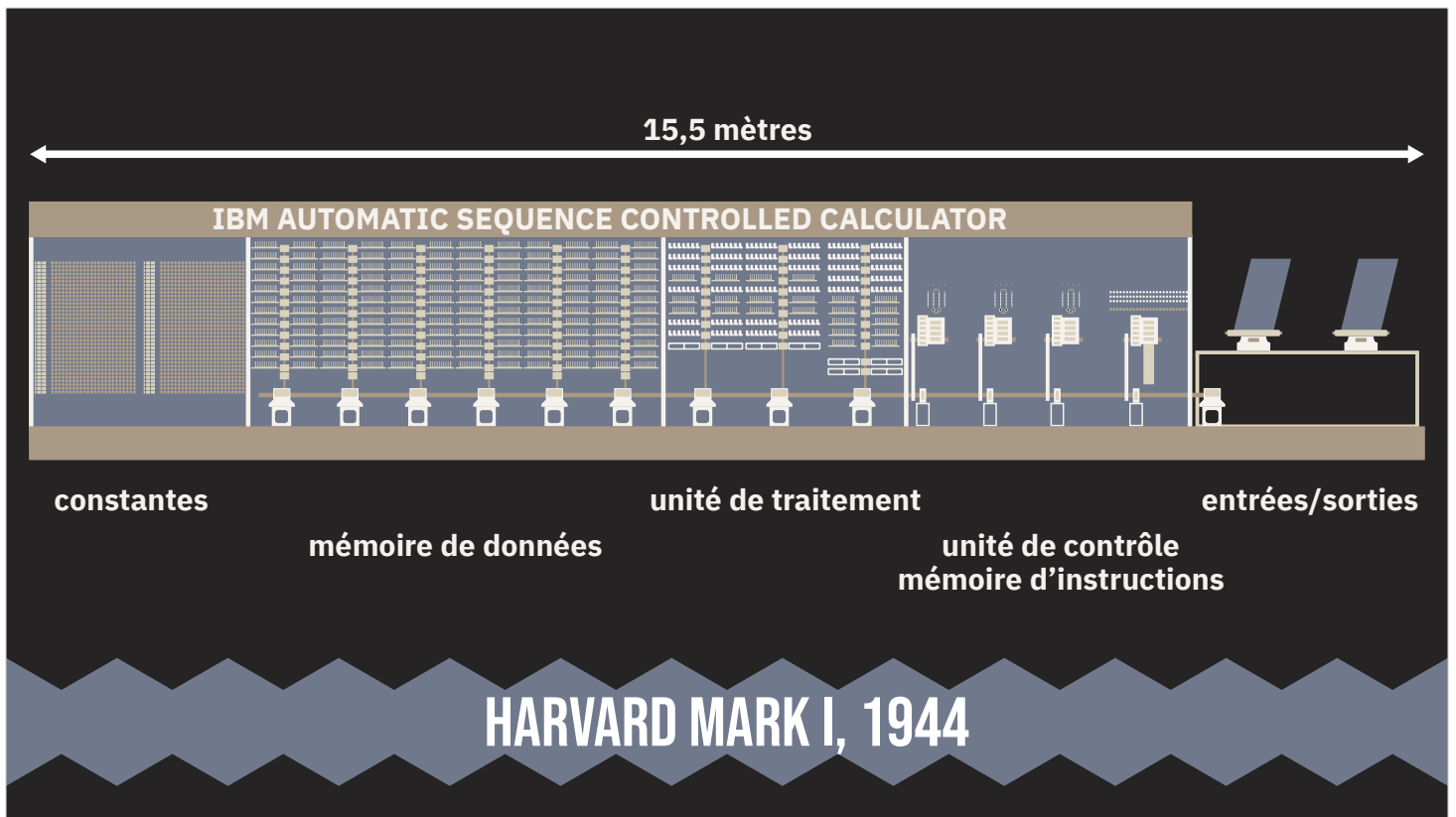
Elle explique également comment déboguer.

Elle se termine par quelques conseils et interrogations sur l'avenir du développement.

# IL ÉTAIT UNE FOIS : LES CPU

Pour comprendre l'intérêt des FPGA aujourd'hui, il faut remonter dans le passé avec les débuts des processeurs.

L'architecture générale utilisée aujourd'hui est issue des toutes premières architectures.



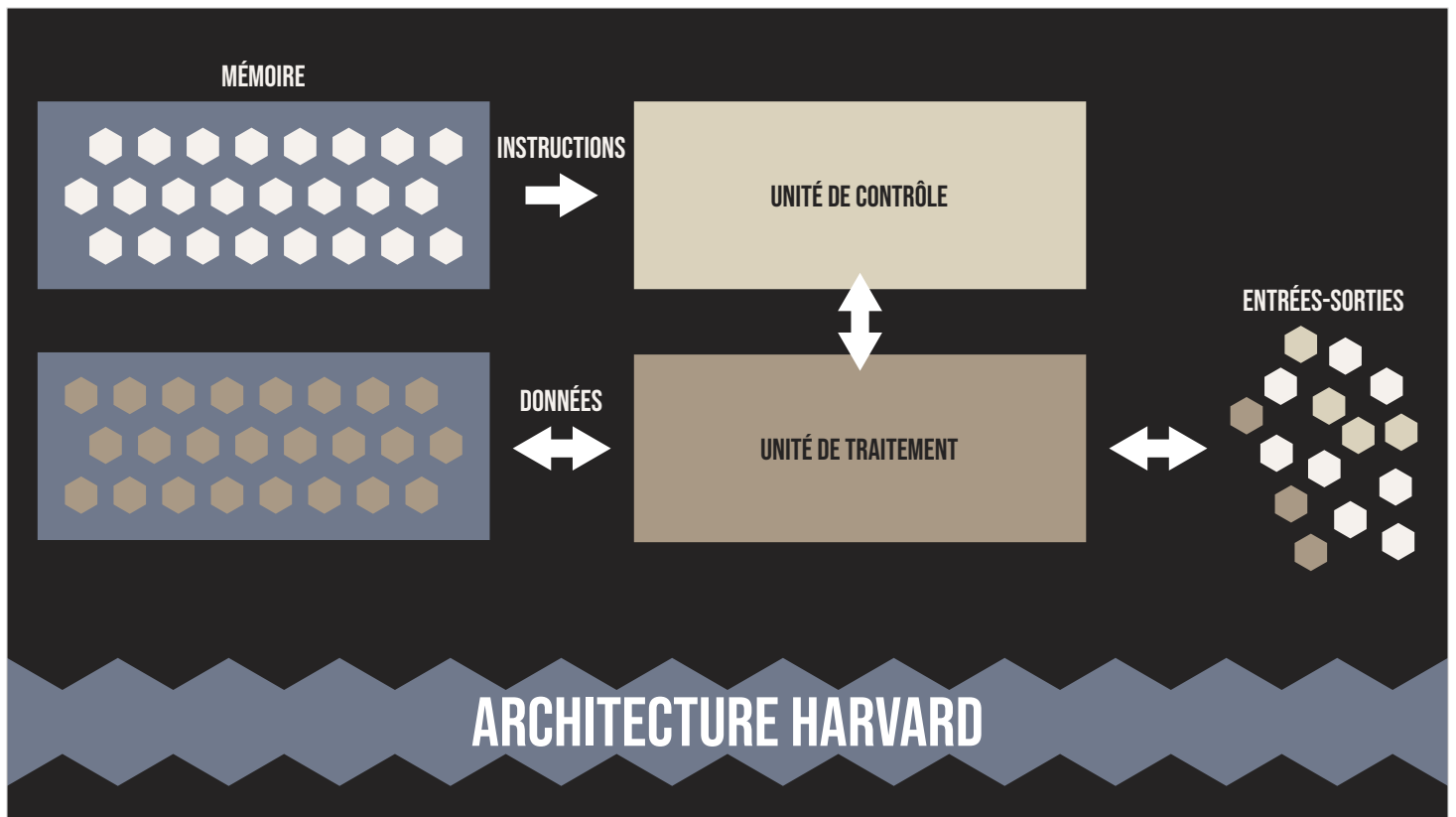
En 1944, l'université d'Harvard et IBM ont mis au point le Harvard Mark I, un monstre de 15,5 mètres de long pour un peu plus de 2 mètres de hauteur.

De gauche à droite, il utilise des constantes et une mémoire de données qui sont utilisées par l'unité de traitement pour les calculs.

Celle-ci est pilotée par l'unité de contrôle qui exécute des programmes lus sur des bandes perforées.

L'unité de traitement est également reliée aux entrées-sorties.

C'est ce qu'on appelle l'architecture Harvard.

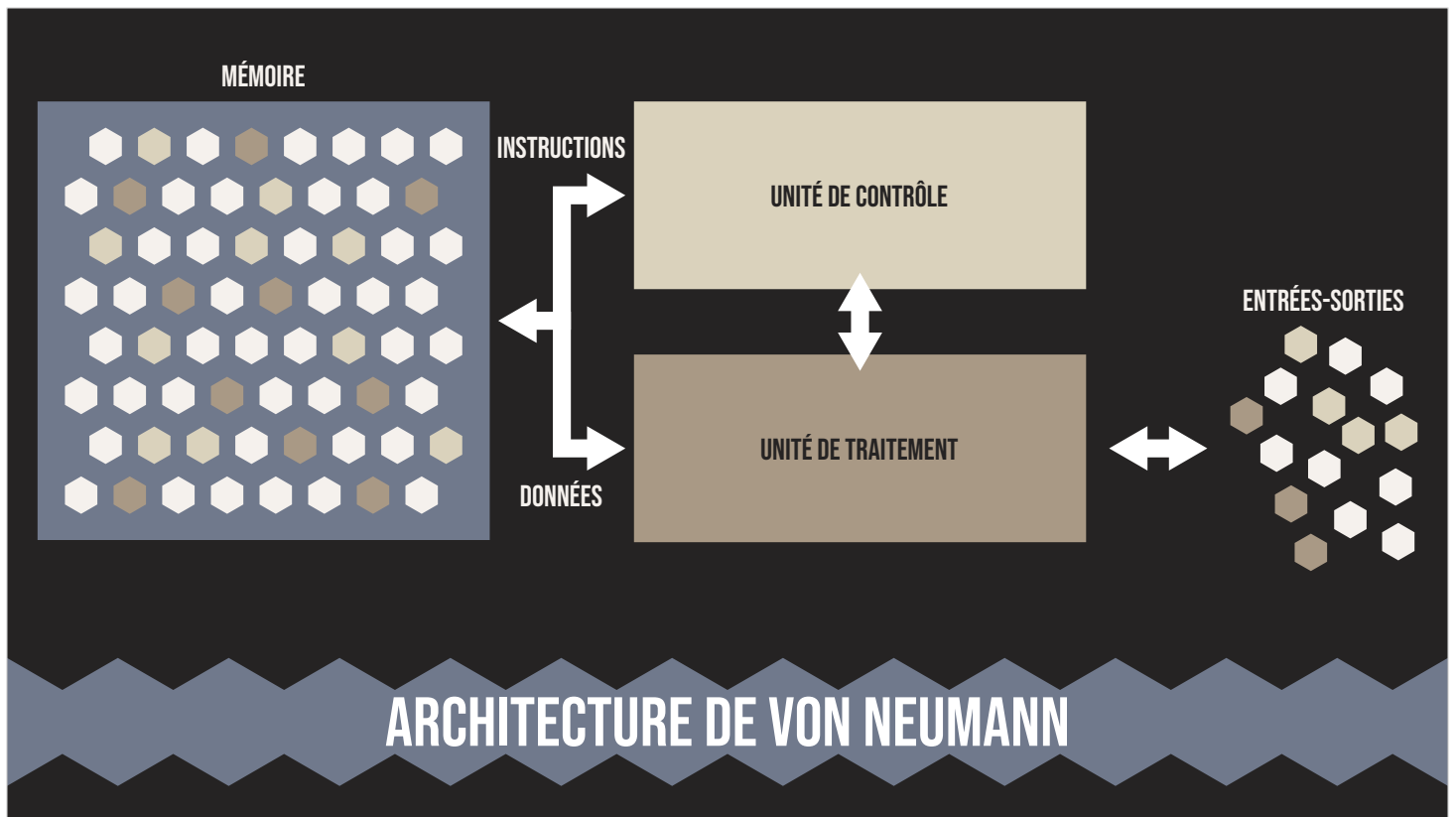


Dans cette architecture, le code et les données sont séparés.

L'unité de contrôle lit les instructions et commande l'unité de traitement.

Celle-ci récupère alors les données et interagit avec les entrées-sorties.

La séparation du code et des données impose certaines contraintes.

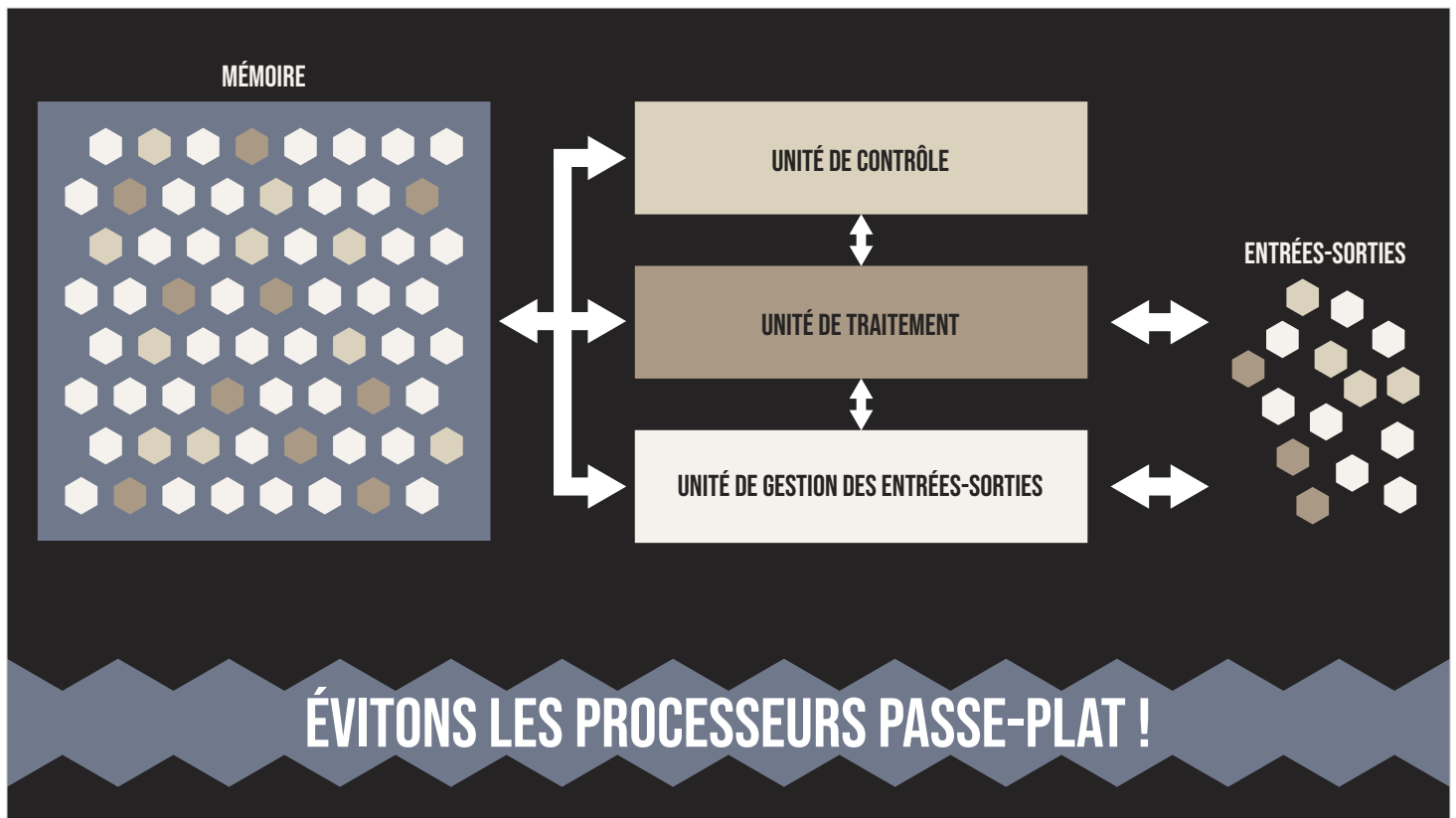


C'est pourquoi l'architecture de Von Neumann a été développée pour l'EDVAC en 1949

Dans cette architecture, la mémoire contient à la fois le code et les données, ce qui permet le code auto-modifiable.

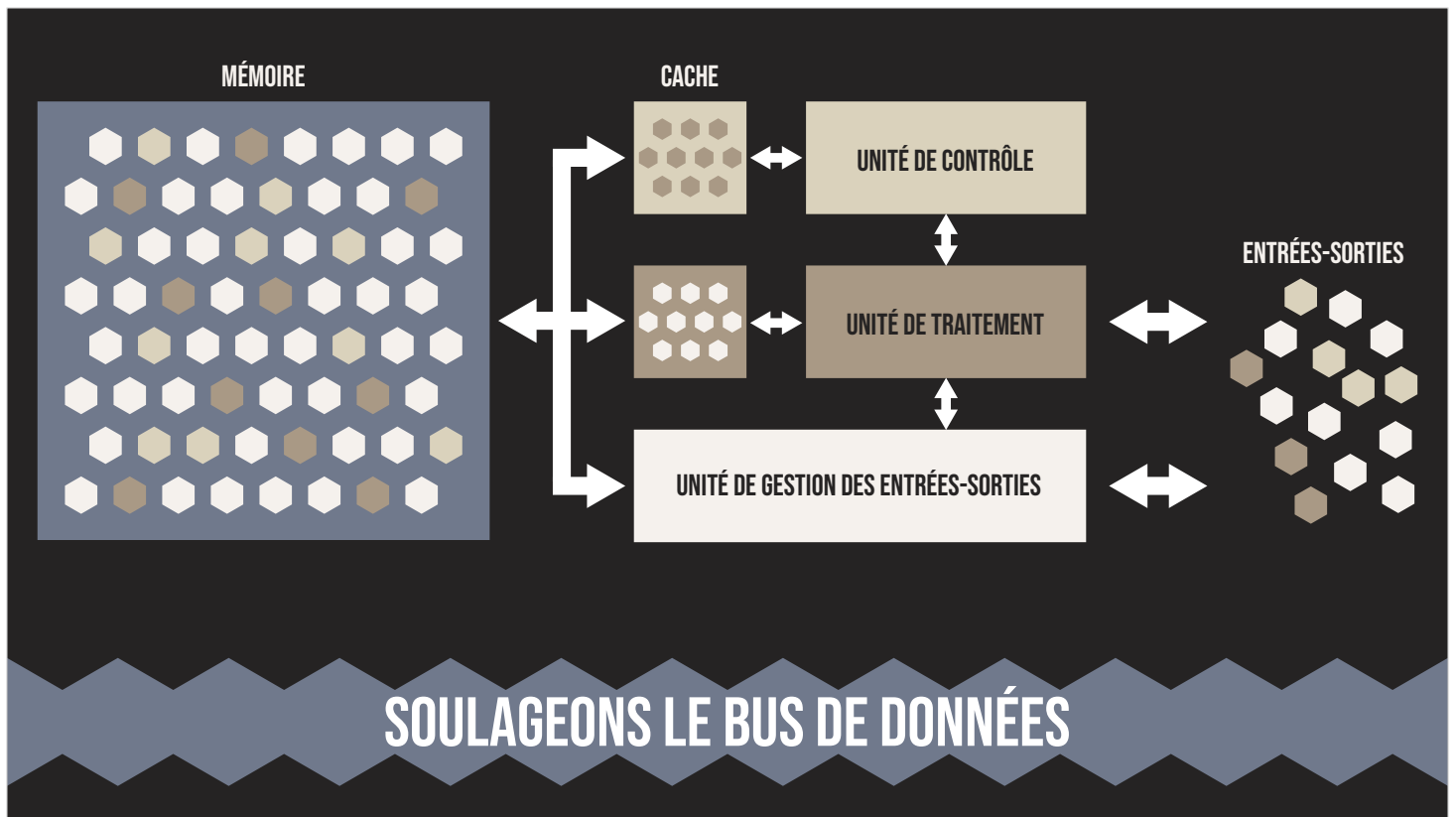
Elle crée malheureusement un goulot d'étranglement puisqu'il est impossible, à la fois, de lire une instruction et d'échanger avec les données.

Autre défaut de ces architectures : le processeur sert de passe-plat entre la mémoire et les entrées-sorties.



Défaut que l'on peut éviter en ajoutant une unité de gestion des entrées-sorties. Les DMA en sont un exemple.

Mais on voit très rapidement que l'accès au bus de données devient critique, d'autant plus quand on traite une masse de données énormes.



C'est pour ça qu'on a ajouté des mémoires caches, plus petites mais plus rapides.

Et elles permettent de revenir à un semblant d'architecture Harvard en séparant le cache d'instructions du cache de données.

Mais on peut encore faire mieux...

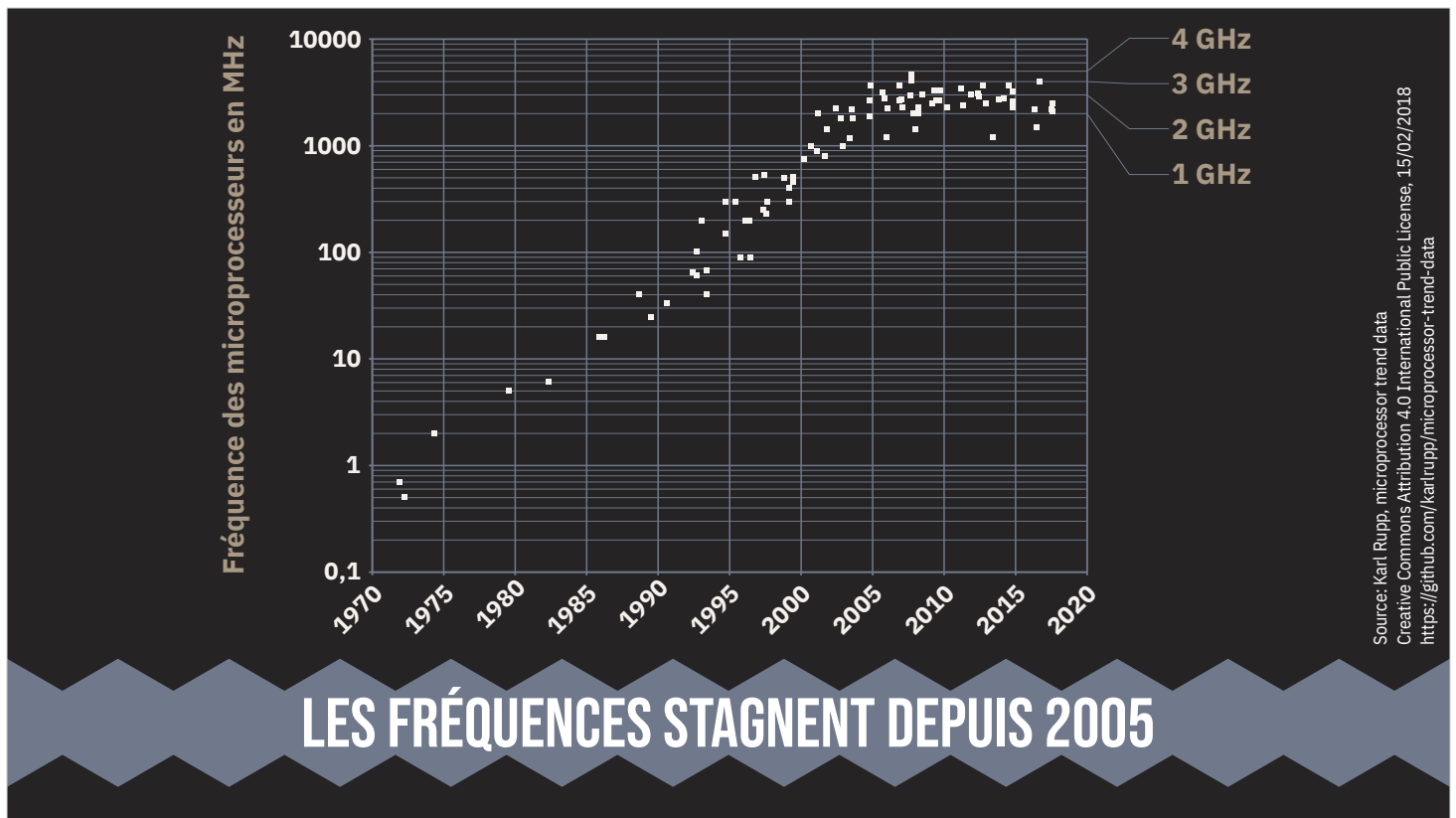




On peut augmenter la largeur du bus de données, la fréquence de fonctionnement, le nombre d'unités de traitement, optimiser le parallélisme, rajouter d'autres caches, des instructions vectorielles...

Seulement... accélérer une 2 chevaux, ça n'en fait pas une fusée !

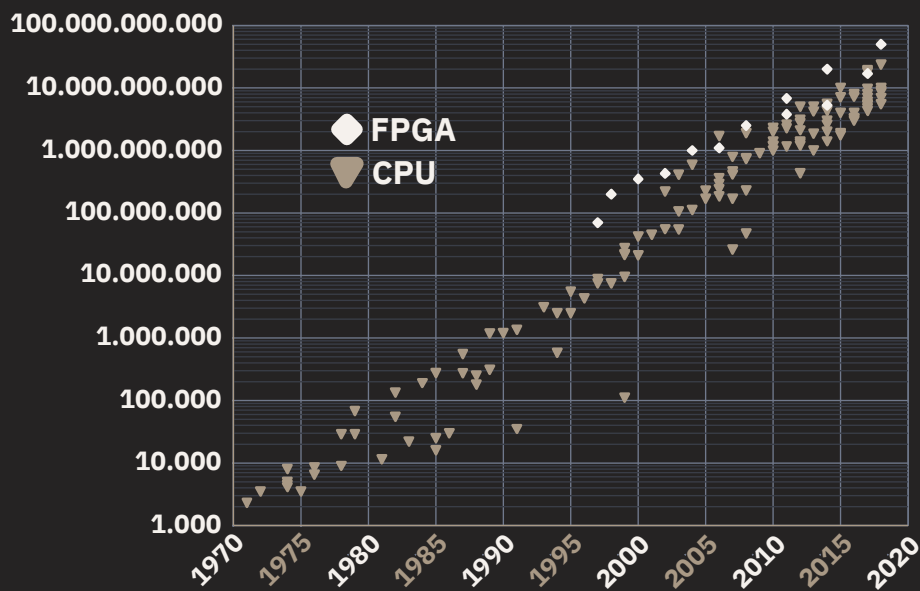
Les CPU sont fortement contraints par l'utilisation d'un bus de données.



Si le matériel continue de progresser, ce n'est pas sans peine car on a déjà rencontré certaines limites technologiques.

Par exemple, les fréquences des processeurs n'évoluent plus depuis 2005, époque à partir de laquelle les multi-cœurs font leur apparition.

Et ceci, après avoir explosé pendant 30 ans.



## LA LOI DE MOORE A PERDURÉ JUSQU'EN 2017

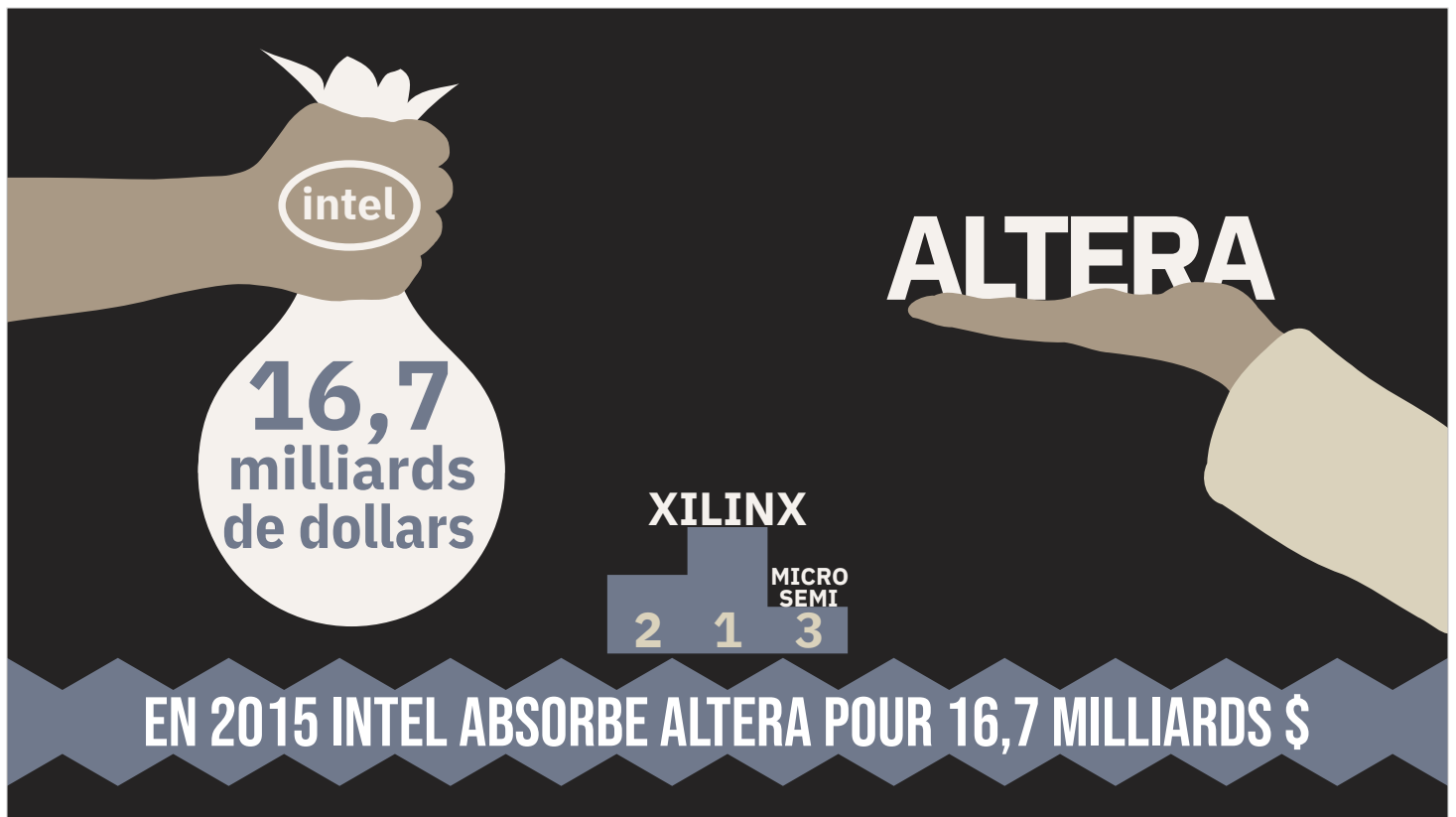
Aujourd'hui, nous atteignons une nouvelle limite : la fin de la loi de Moore.

Le plus gros FPGA comporte aujourd'hui 50 milliards de transistors.

La loi de Moore est limitée par la finesse de gravure. Celle-ci est aujourd'hui très proche de la taille du silicium. Les coûts imposés par cette finesse ne sont plus couverts par les économies d'échelle et pose des problèmes croissants de fuite de courant.

Le modèle économique du matériel informatique étant basé sur une croissance continue, comment continuer à avoir du matériel plus performant ?

Une solution est d'opter pour des architectures hybrides qui font un meilleur usage des ressources disponibles comme les GPU ou les FPGA.

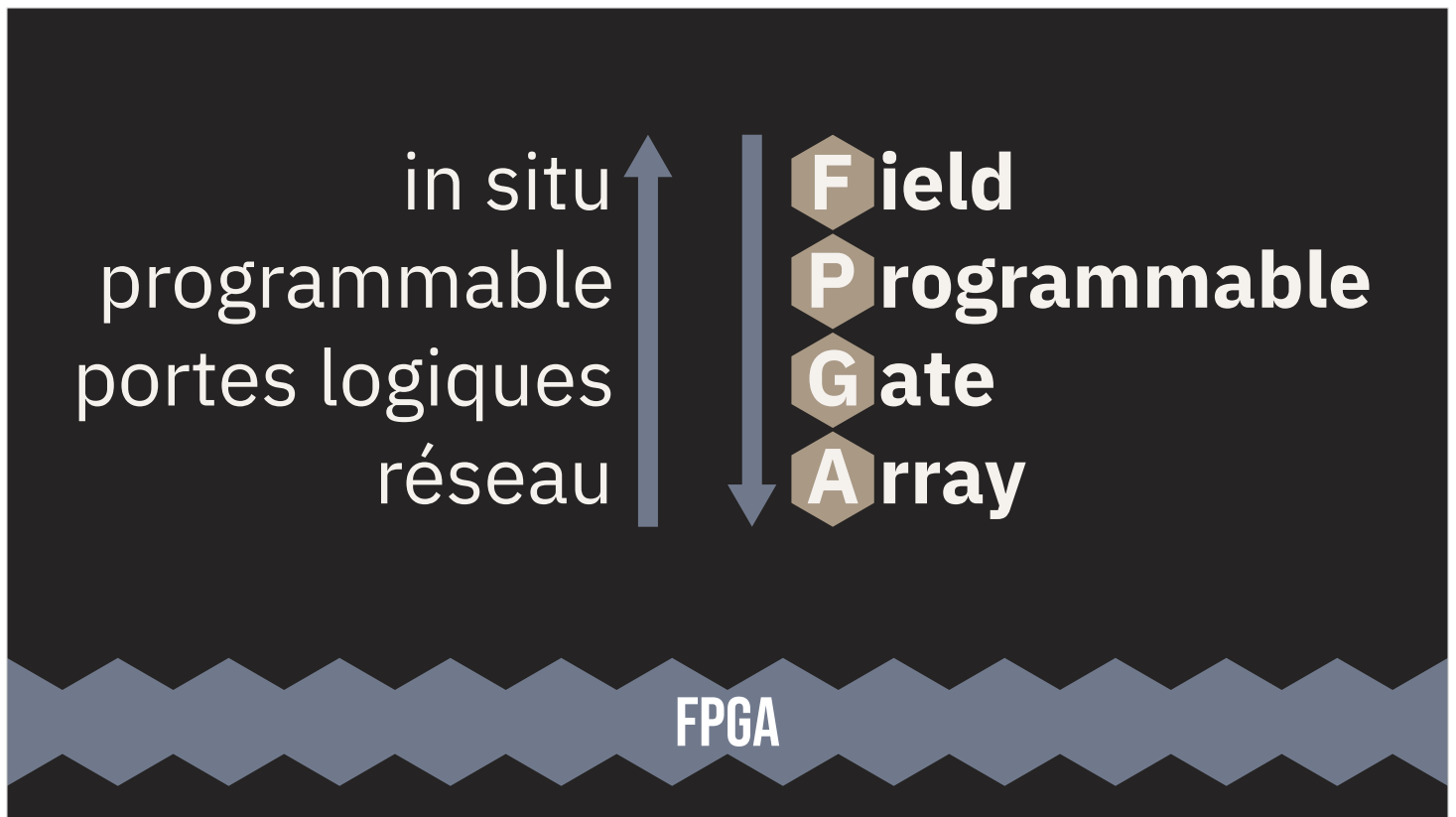


En 2015, Intel s'est offert Altera pour 16,7 milliards de dollars, le numéro 2 des fabricants de FPGA, derrière Xilinx et devant MicroSemi.

Alors, qu'est-ce que le numéro 1 des circuits intégrés a bien pu trouvé d'intéressant dans une technologie issue des années 80 ?

# C'EST QUOI UN FPGA ?

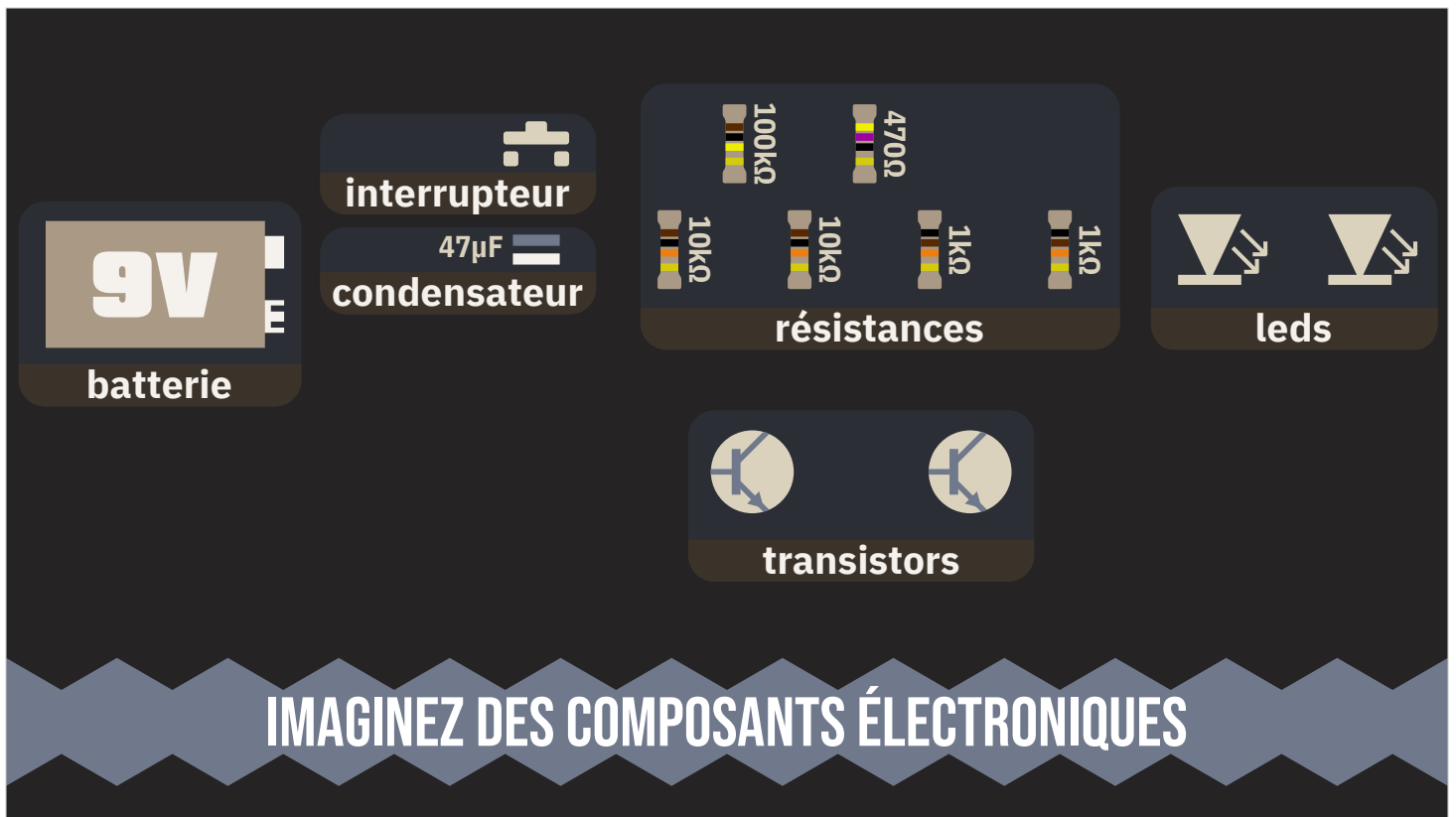
Cela nous amène à nous poser la question : c'est quoi un FPGA ?



FPGA, c'est l'acronyme de Field Programmable Gate Array.

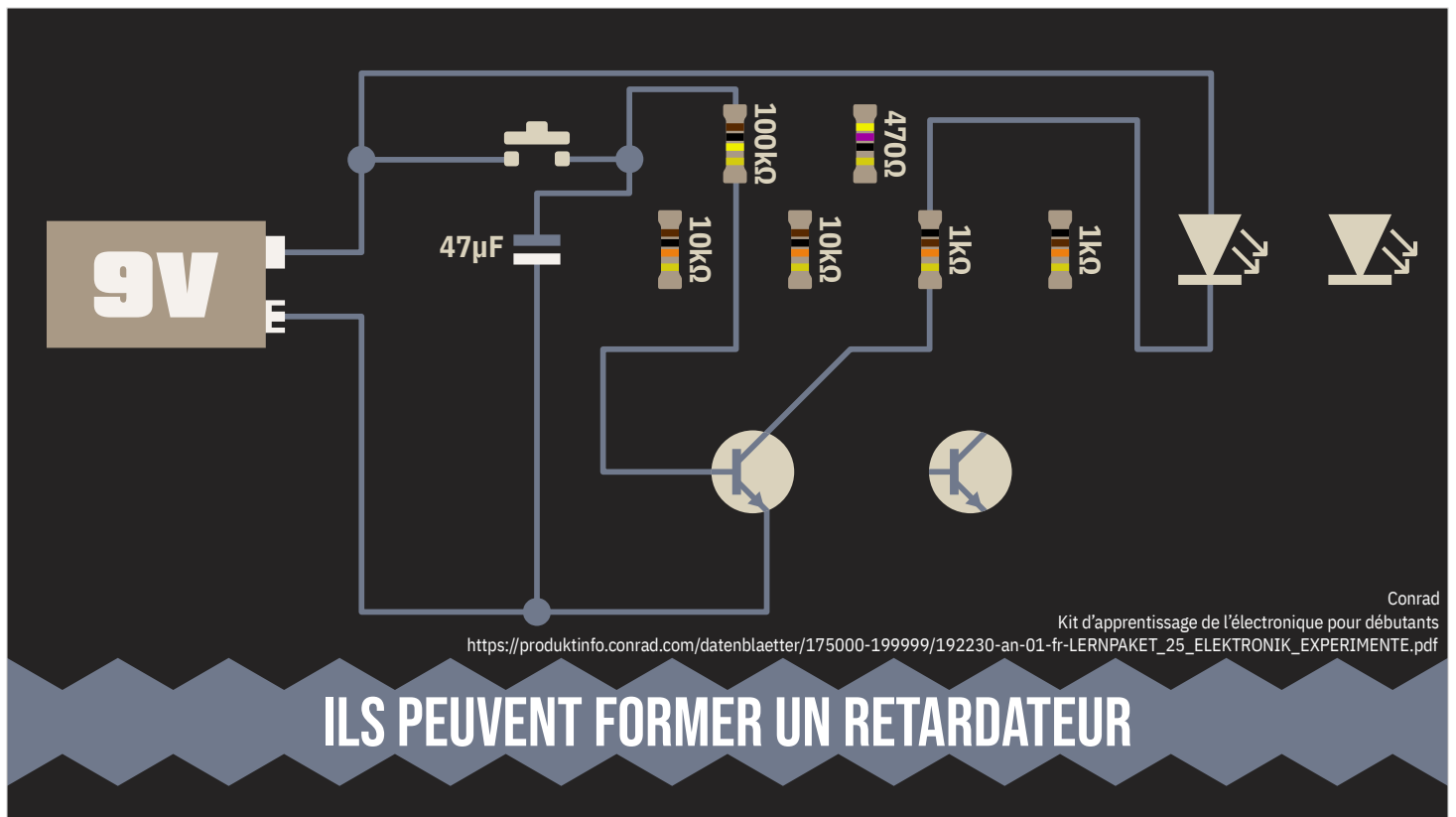
Ce qui donne « réseau de portes logiques programmable in situ » en français.

Ok ! Ça ne nous en dit pas plus et pourtant, chacun de ces termes n'est pas là par hasard.



Pour mieux comprendre, nous allons recourir à une analogie.

Imaginez des composants électroniques basiques : une batterie, un interrupteur, un condensateur, six résistances, deux transistors et deux leds.



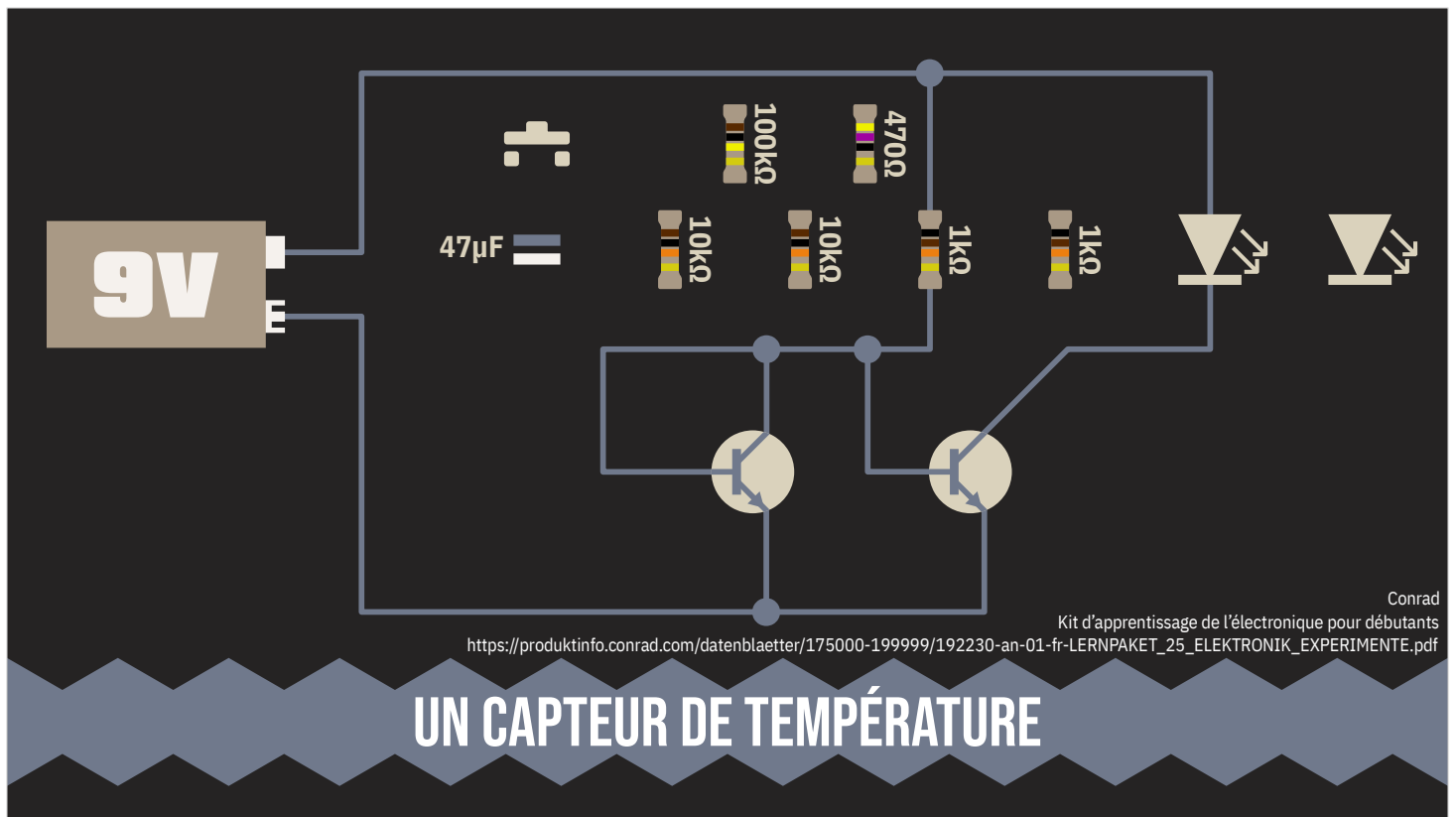
En les reliant comme sur le schéma on obtient un retardateur.

Quand on appuie sur l'interrupteur, le condensateur va se charger d'énergie.

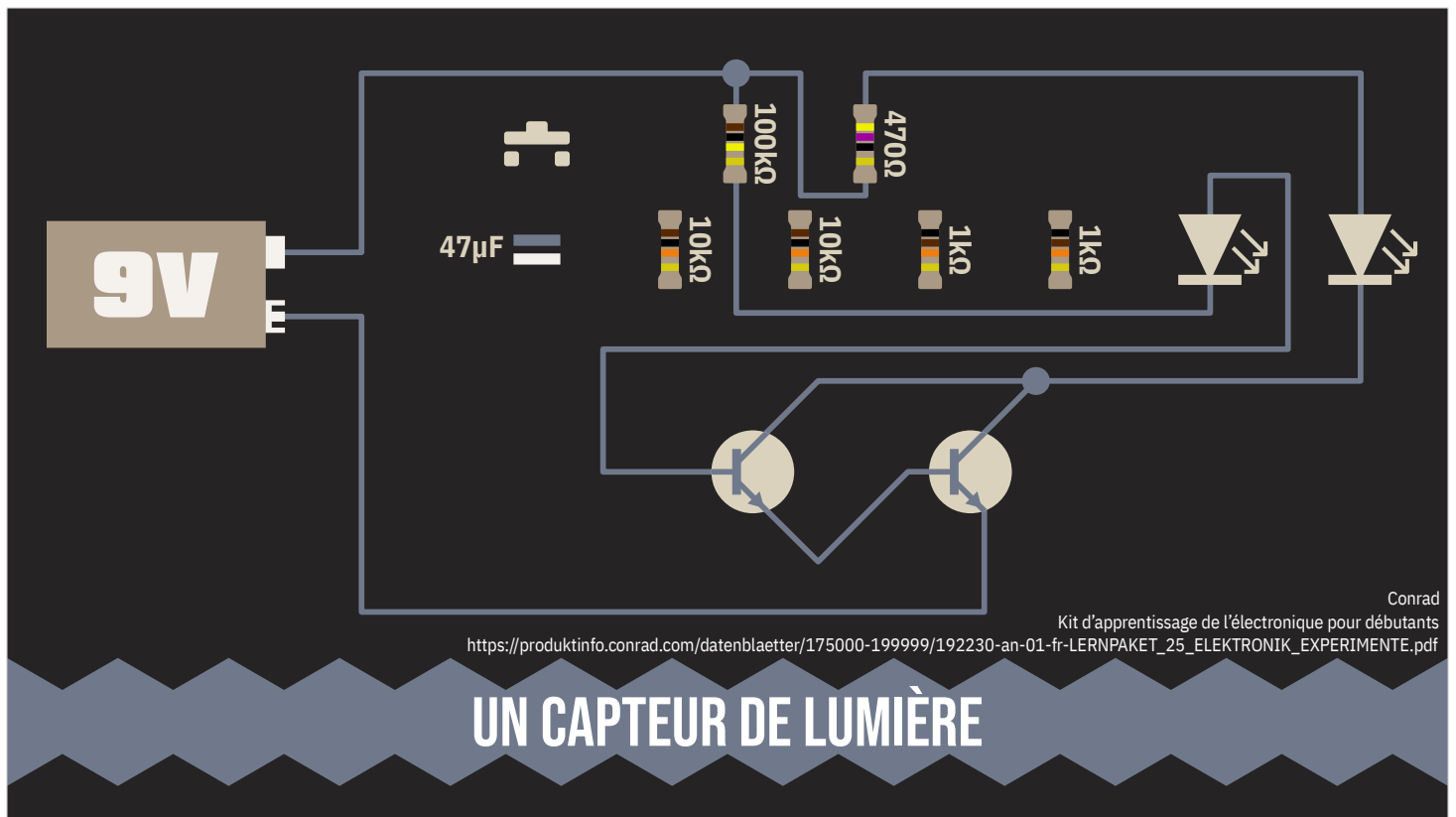
Quand on le relâche, il va piloter le transistor pour que celui-ci ouvre les vannes et que la led s'allume.

Une fois le condensateur vidé, les vannes sont fermées et la led s'éteint.



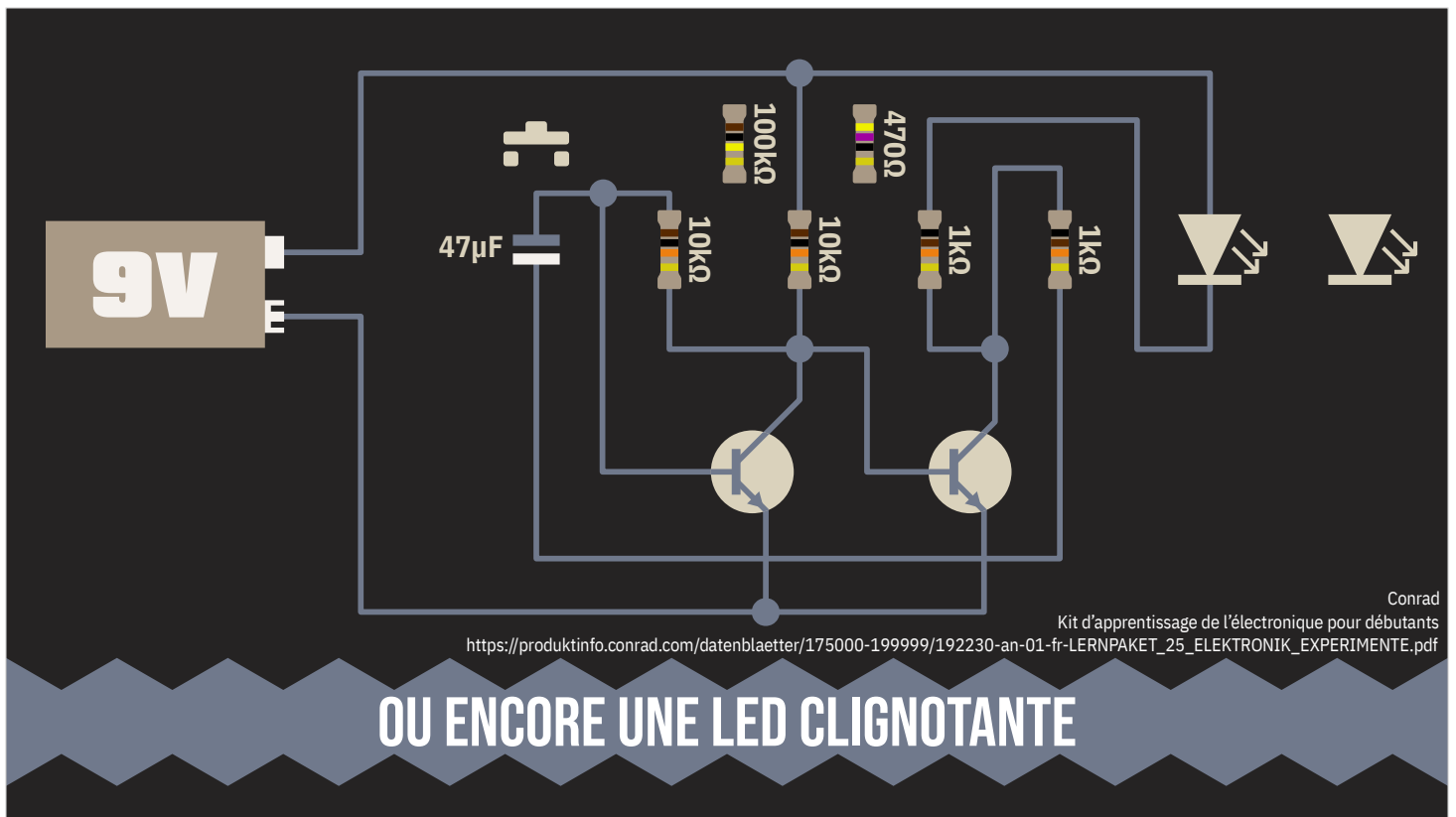


En utilisant le fait que la réponse d'un transistor varie en fonction de sa température, on peut créer un circuit qui va détecter une variation de température.



En utilisant une led à l'envers, on peut la transformer en capteur de lumière.

Le courant à mesurer est si faible qu'il nécessite d'être fortement amplifié par un montage à deux transistors.



On peut aussi réaliser un circuit qui fera clignoter une led.

Je pense que vous avez deviné le motif commun à ces schémas.

## QUELQUES REMARQUES

- Le câblage définit le fonctionnement du circuit
- Les composants
  - restent fixes entre les différents schémas
  - ne sont pas tous utilisés pour un schéma donné
- C'est le principe d'un FPGA !

Sur ces quatre schémas, c'est le câblage qui a défini le fonctionnement du circuit.

À aucun moment les composants n'ont été déplacés.

Et ils n'ont pas tous été utilisés dans tous les schémas.

C'est le principe d'un FPGA.

# RÉSEAU DE PORTES LOGIQUES PROGRAMMABLE IN SITU

- Un FPGA a 3 éléments constitutifs
  - des élément logiques (ALM/CLB, mémoire, DSP...)
  - un réseau de pistes
  - une mémoire de configuration
- Le réseau de pistes est figé
  - un FPGA reste un circuit intégré
  - toutes les combinaisons ne sont pas possibles
  - il est reconfigurable à volonté grâce à la mémoire

Un FPGA a 3 éléments fondamentaux : des éléments logiques chargés d'effectuer les traitements (ALM chez Intel, CLB chez Xilinx...), un réseau de pistes pour relier ces éléments logiques et une mémoire de configuration qui définit la configuration du réseau.

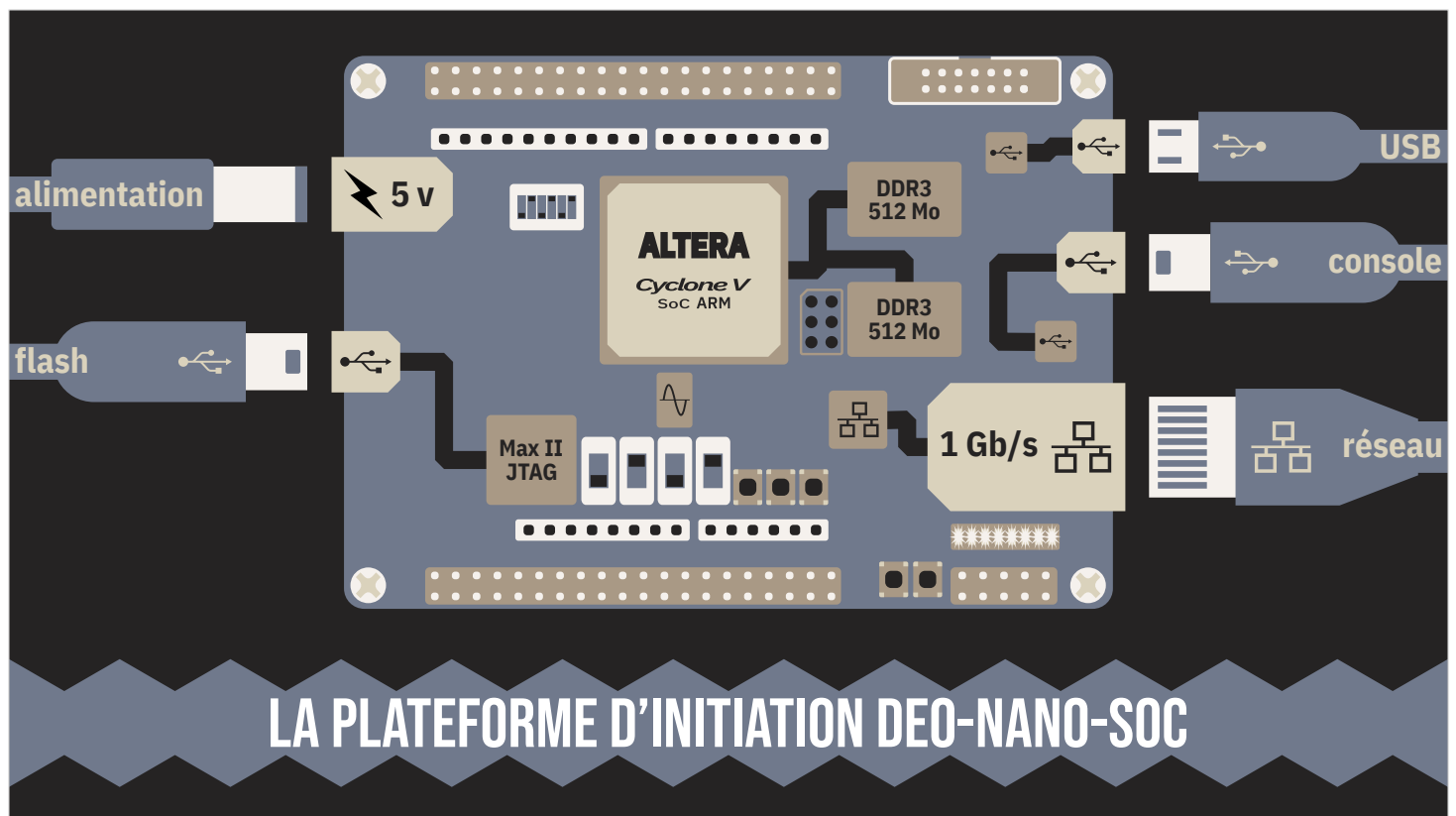
Le FPGA étant un circuit intégré, le réseau de pistes est figé mais reste reconfigurable à volonté.

Ses ressources sont limitées ce qui implique que toutes les combinaisons de routes imaginables ne sont pas possibles.

# LE DE0-NANO-SOC

Assez parlé de généralité !

Voici le DE0-Nano-SoC, une plateforme d'initiation fabriquée par Terasic.

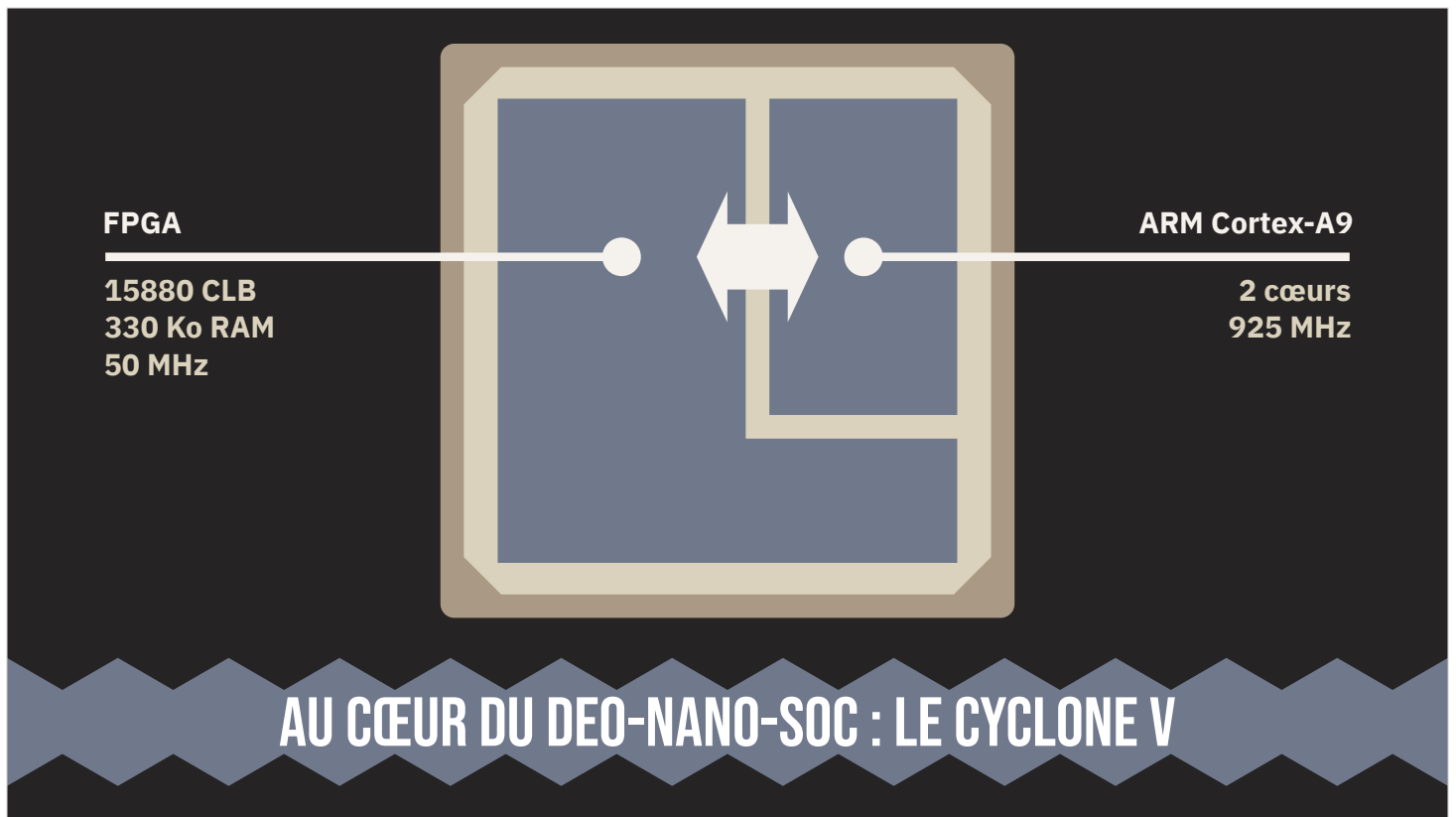


Le DE0-Nano-SoC se présente sous la forme d'une carte aux dimensions proches d'un Raspberry Pi ou d'un Arduino.

Il est alimenté en 5 volts et dispose d'un port USB qu'il suffit de connecter sur le PC pour pouvoir programmer le FPGA.

Il est également équipé d'un port gigabit ethernet, d'une prise USB série pour connecter une console série.

Le nombre de broches disponibles est largement supérieur au nombre de broches d'un Arduino Mega.



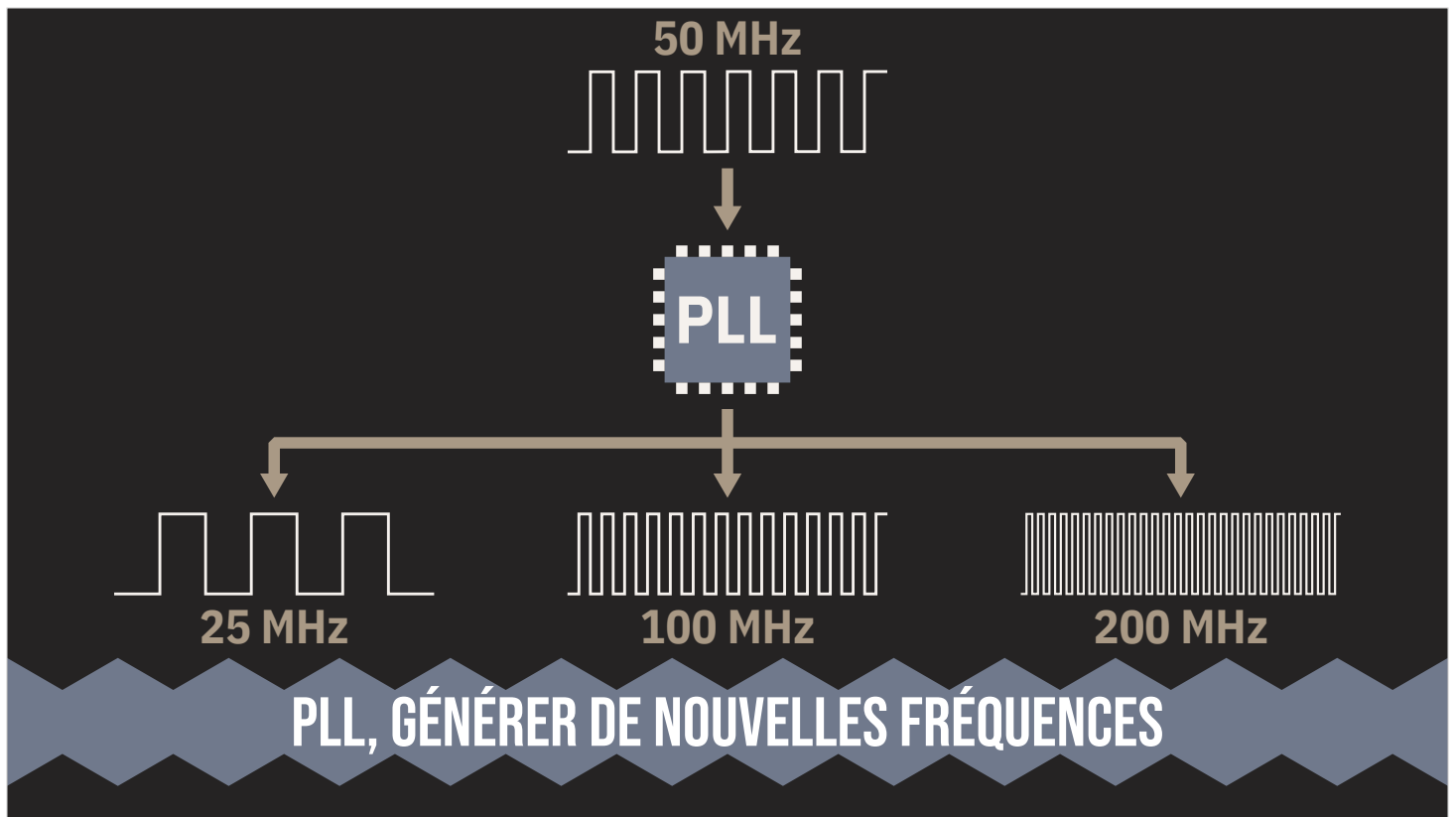
Au cœur du DE0-Nano-SoC se trouve le Cyclone V SoC.

Il s'agit d'un combo FPGA + ARM.

Il comporte sur la même puce : un FPGA de 15000 cellules, 330 kilooctets de RAM à 50 mégahertz et un ARM Cortex A9 à double cœur à 925 mégahertz.

Les deux peuvent communiquer directement et il est possible de désactiver l'un ou l'autre.



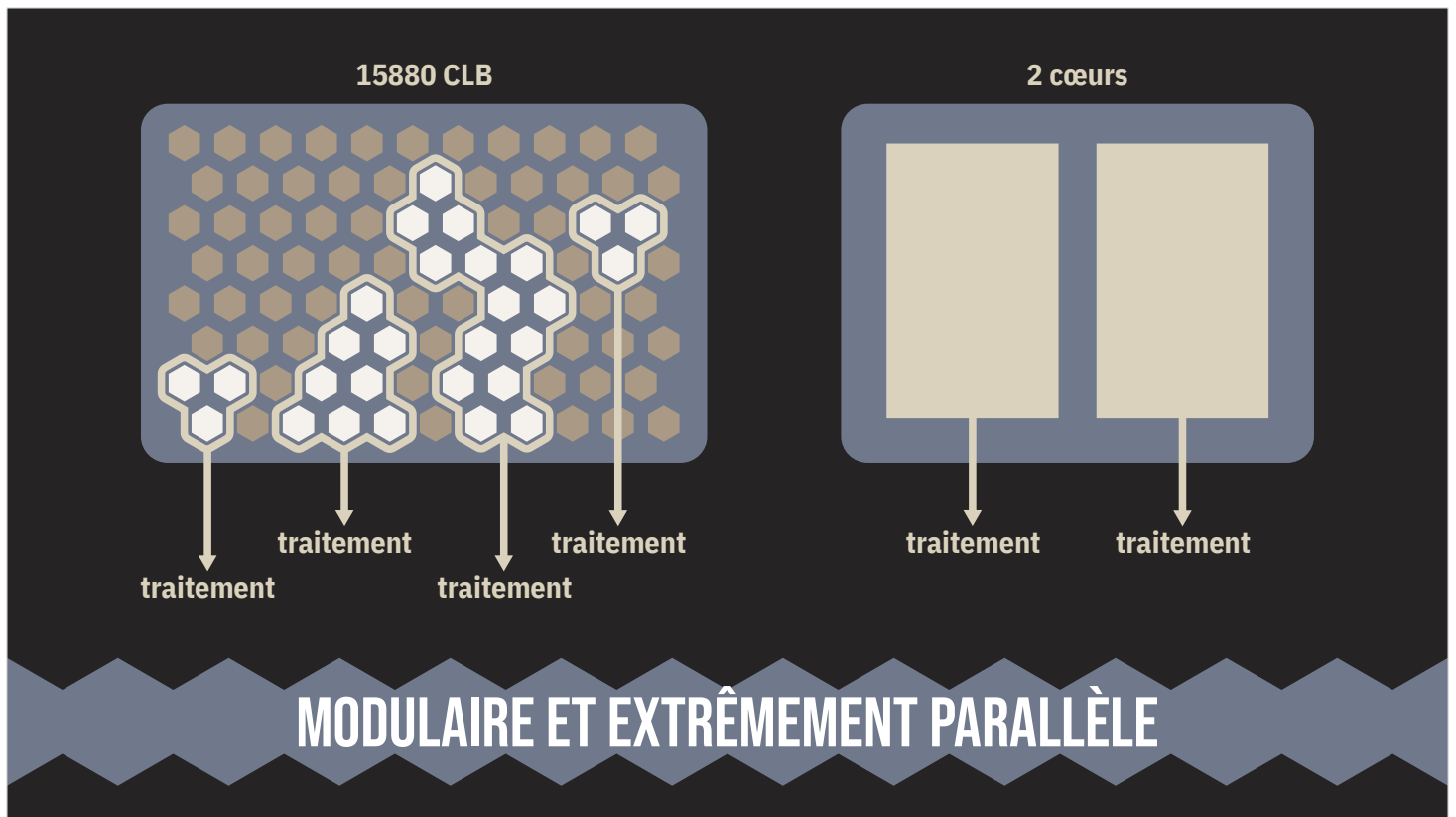


Attention aux idées reçues !

Certes les FPGA sont souvent cadencés à des fréquences moindres (ici 50 mégahertz).

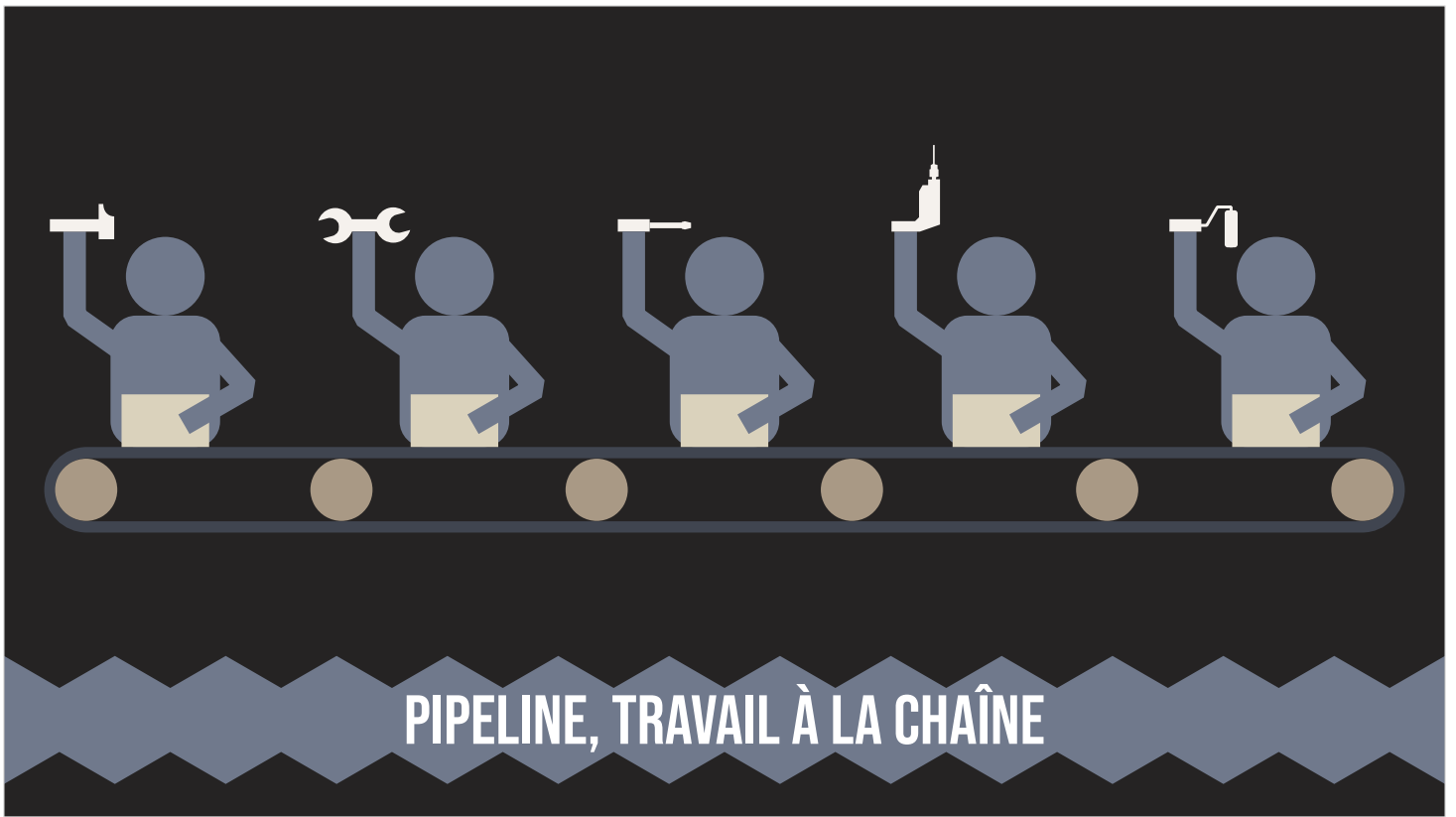
Mais les FPGA embarquent souvent un circuit appelé PLL qui permet de générer de nouvelles fréquences, qu'elles soient plus petites ou même plus grandes.

On peut passer de 50 à 200 mégahertz sans problème.



L'autre particularité qui va permettre au FPGA de faire la différence, est que son architecture est extrêmement modulaire et parallèle.

Les 15000 cellules peuvent être combinées en fonction du besoin et peuvent même fonctionner à des fréquences différentes les unes des autres.



Cette architecture parallèle bénéficie en plus de la possibilité de créer des pipelines spécialisés dans une tâche donnée.

Un pipeline ce n'est ni plus ni moins que le travail à la chaîne appliqué à l'électronique : chaque étape opère une tâche élémentaire et transmet le résultat à son successeur.

Pendant que le successeur commence à travailler, l'étape actuelle peut immédiatement effectuer une nouvelle itération.

# COMMENT PROGRAMMER UN FPGA ?

Comment programme-t-on un circuit avec une telle architecture ?

## LES OUTILS DISPONIBLES

- Chaque fabricant a ses propres outils
  - Xilinx → Vivado/ISE
  - Intel → Quartus Prime
  - Lattice → Diamond
  - etc.
- Ils sont **indispensables** pour générer l'image bitstream
  - ils sont gratuits pour les cartes d'initiation
  - les outils libres ne couvrent pas tout le flot de conception

Il faut d'abord savoir que chaque fabricant propose ses propres outils : Vivado ISE pour Xilinx, Quartus Prime pour Intel, Diamond pour Lattice, etc.

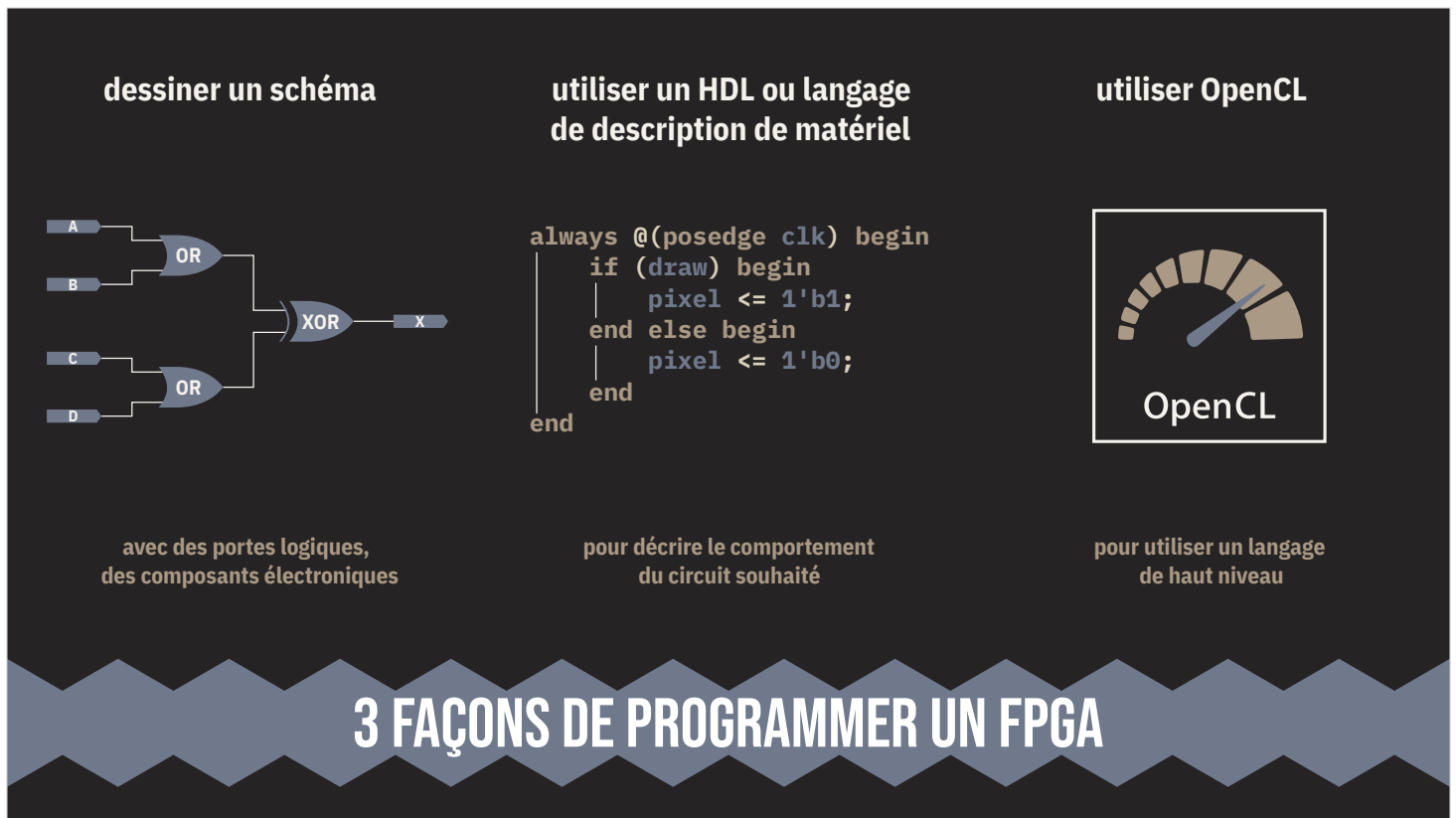
Le gros inconvénient est qu'il vous faudra travailler avec un OS supporté.

Si vous les téléchargez, soyez patients, vous êtes partis pour plusieurs gigaoctets. Et ils sont chers mais gratuits pour les plateformes d'initiation.

On est malheureusement contraint d'y recourir car ils sont indispensables pour la génération de l'image (ou bitstream).

Des préparatifs sont nécessaires pour Quartus Prime sous Linux :

<https://github.com/Zigazou/DE0-nano-soc-francais>



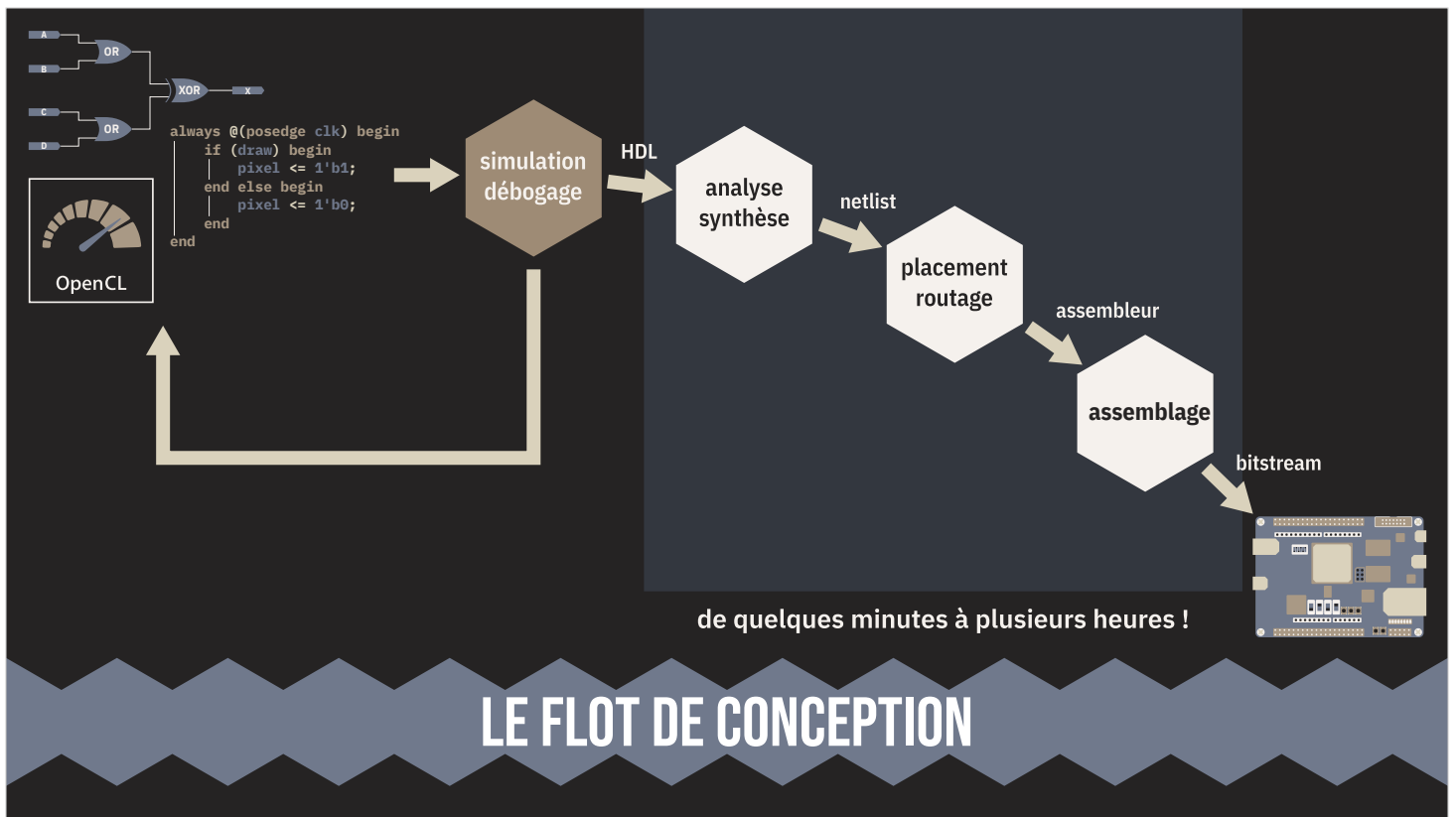
Il y a 3 façons de programmer un FPGA.

Dessiner un schéma permet de travailler graphiquement avec des portes logiques et des composants électroniques.

L'utilisation d'un langage de description de matériel (ou HDL), on peut décrire le comportement du circuit souhaité.

L'utilisation d'OpenCL est la façon de faire la plus éloignée du FPGA puisqu'on utilise une interface de haut niveau.

Cette présentation ne couvre que l'utilisation d'un HDL.



Le flot de conception a 3 phases : l'analyse synthèse, le placement routage et l'assemblage.

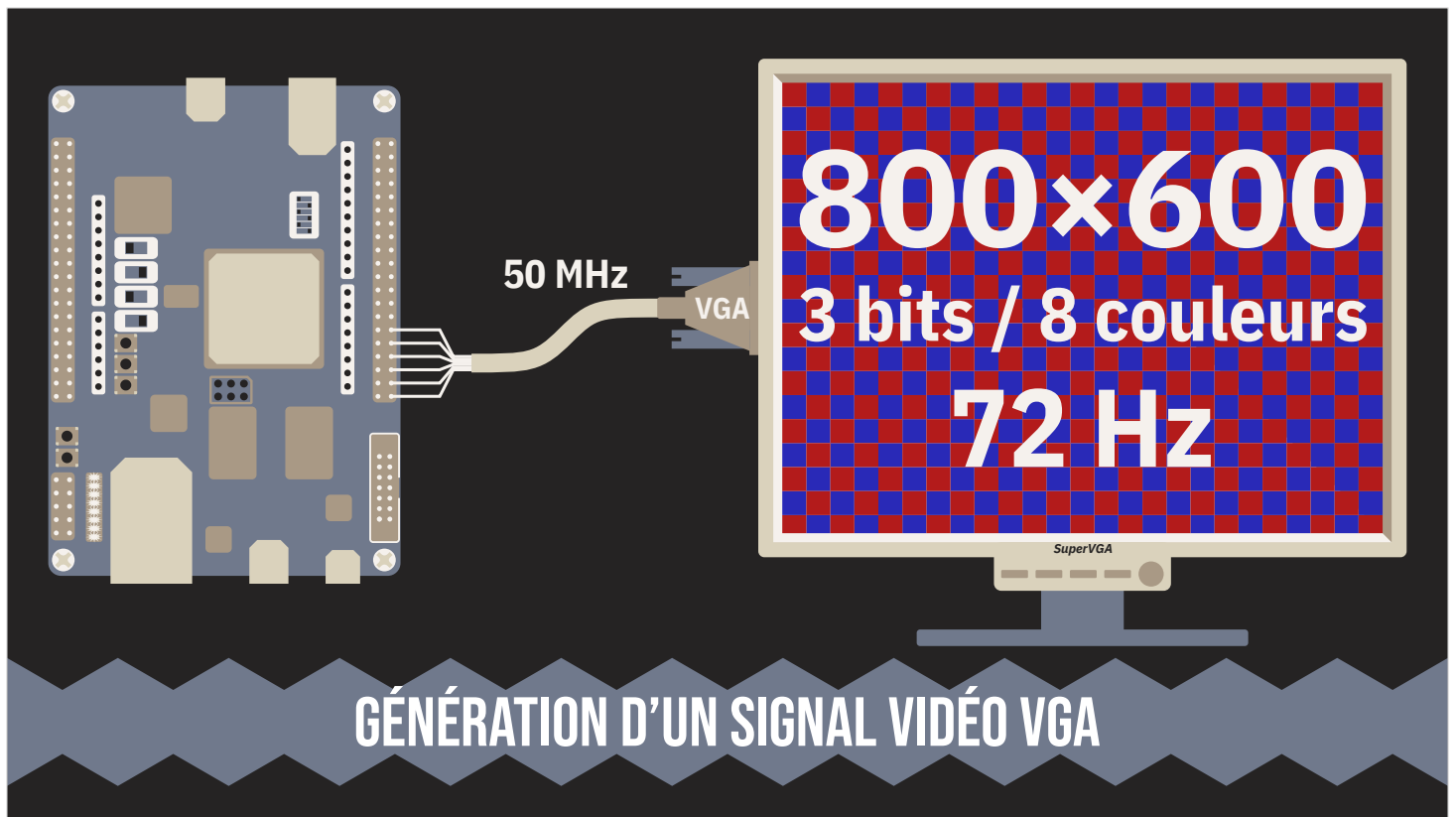
Plus on progresse dans le flot et moins d'outils libres sont disponibles.

La phase de simulation/débogage est indispensable car les phases suivantes peuvent prendre plusieurs heures !

# UN EXEMPLE : SIMPLVGA

L'analyse d'un petit projet va nous permettre de comprendre les bases de la programmation de FPGA.





Le DE0-Nano-SoC n'a aucun composant vidéo.

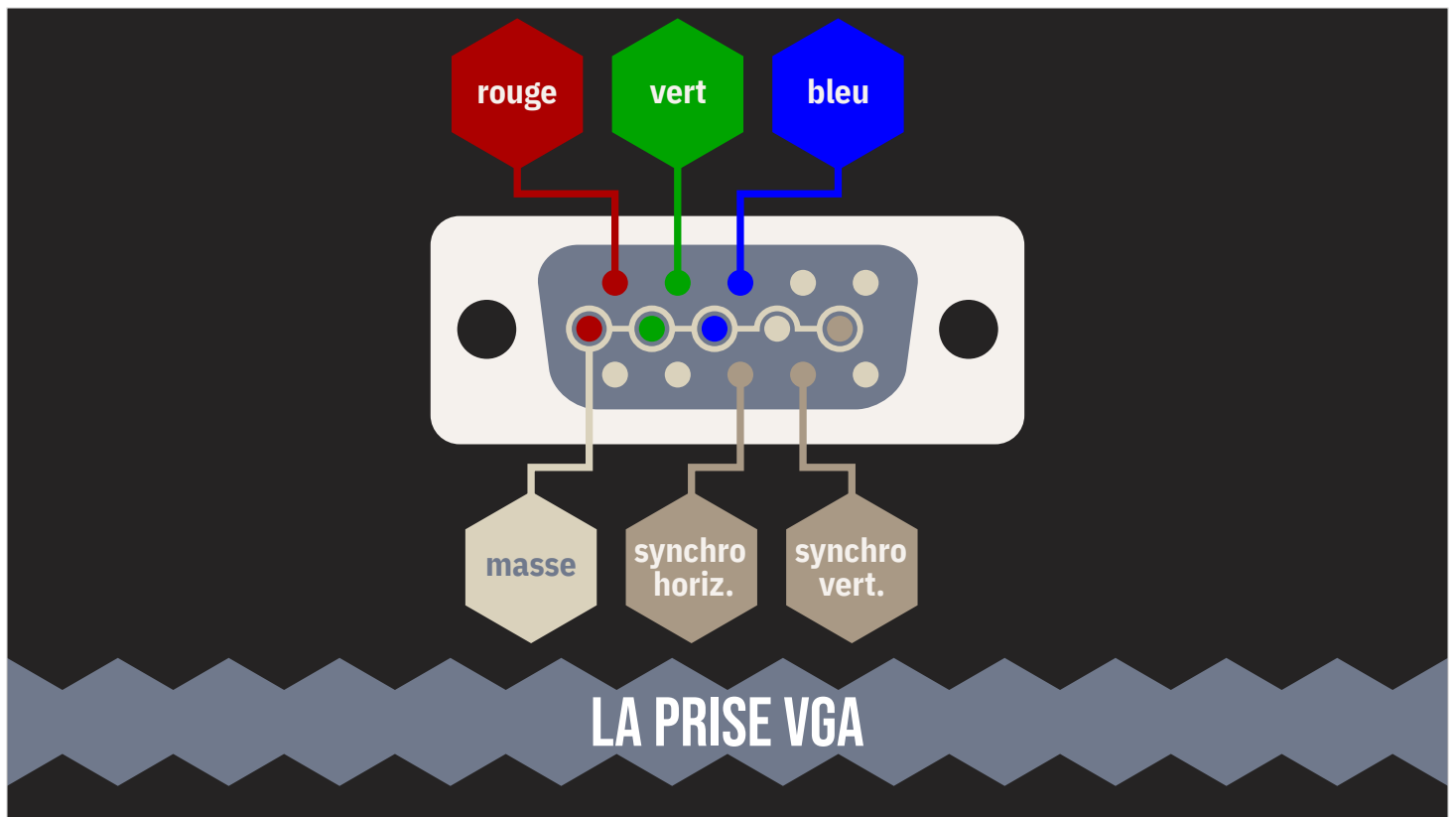
Nous allons générer un signal VGA qui affichera un damier rouge et bleu.

Sa résolution d'affichage sera de 800 par 600 pixels en 8 couleurs (3 bits).

Pour simplifier, nous utiliserons une fréquence de 72 hertz car elle permet d'avoir une horloge de point fonctionnant à 50 mégahertz, la fréquence de base du DE0-Nano-SoC.

Il n'y a pas de mémoire vidéo, l'image est entièrement générée à chaque trame.

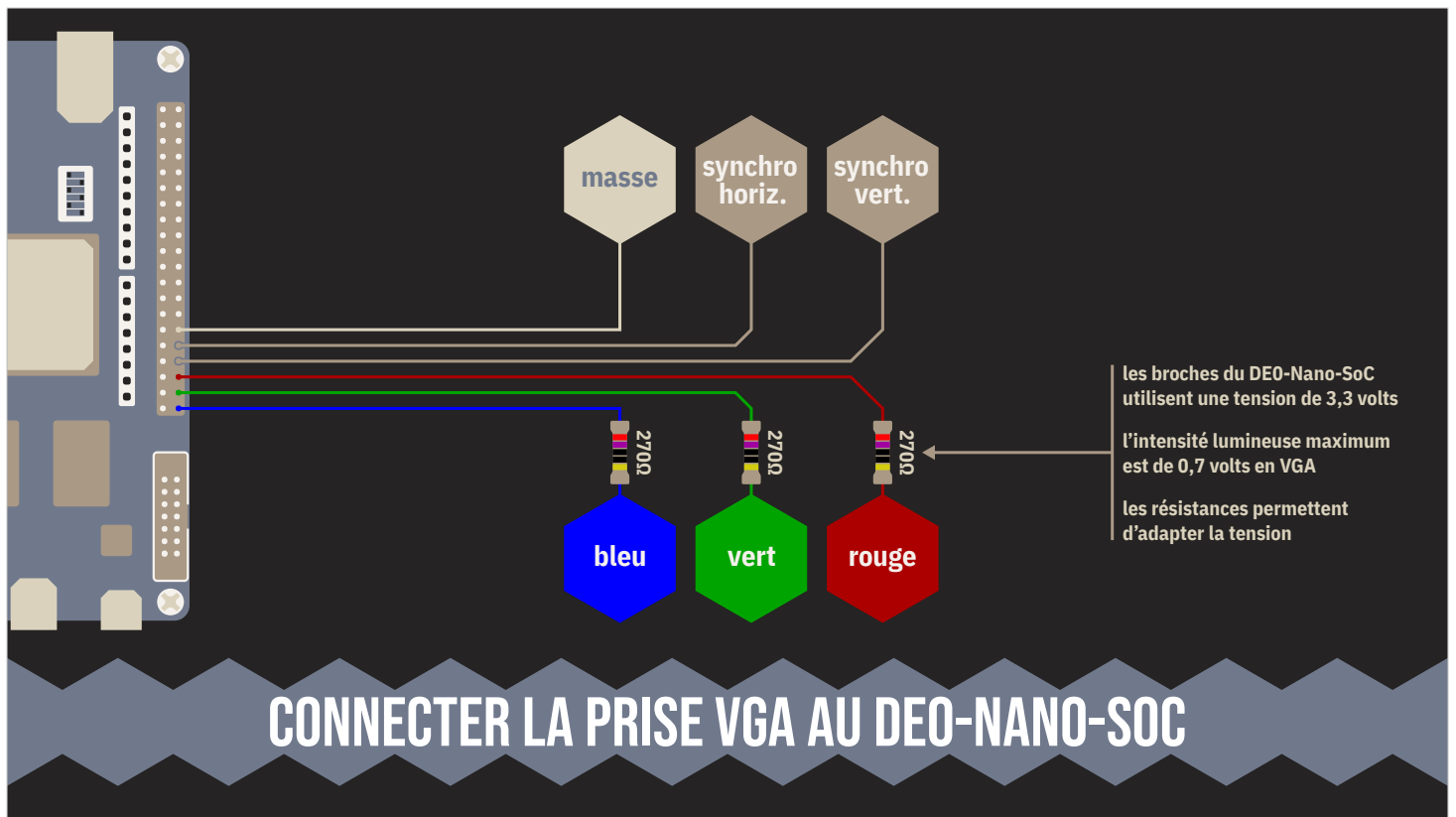
Et c'est notre FPGA qui va tout faire de A à Z.



On commence tout d'abord par étudier la partie physique.

La prise VGA comporte 15 broches mais nous n'en utiliserons que 6 : la masse, les 2 synchronisations horizontale et verticale et les trois couleurs rouge vert et bleu.

Les broches de synchronisation permettent au moniteur de savoir quand une nouvelle ligne, ou une nouvelle trame, débute.

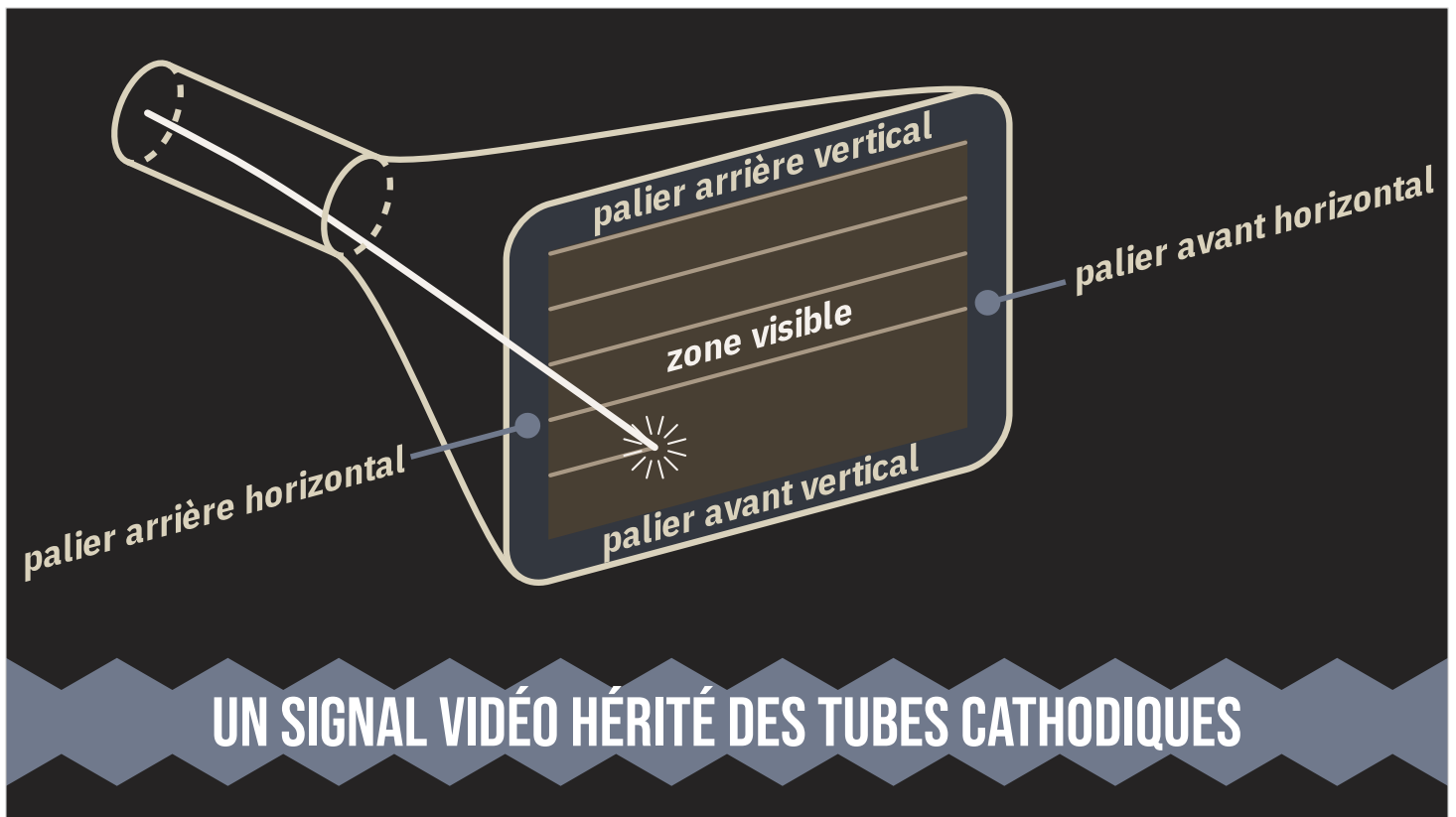


On relie ensuite les 6 broches au FPGA.

C'est une connexion directe à l'exception des 3 résistances sur les 3 couleurs.

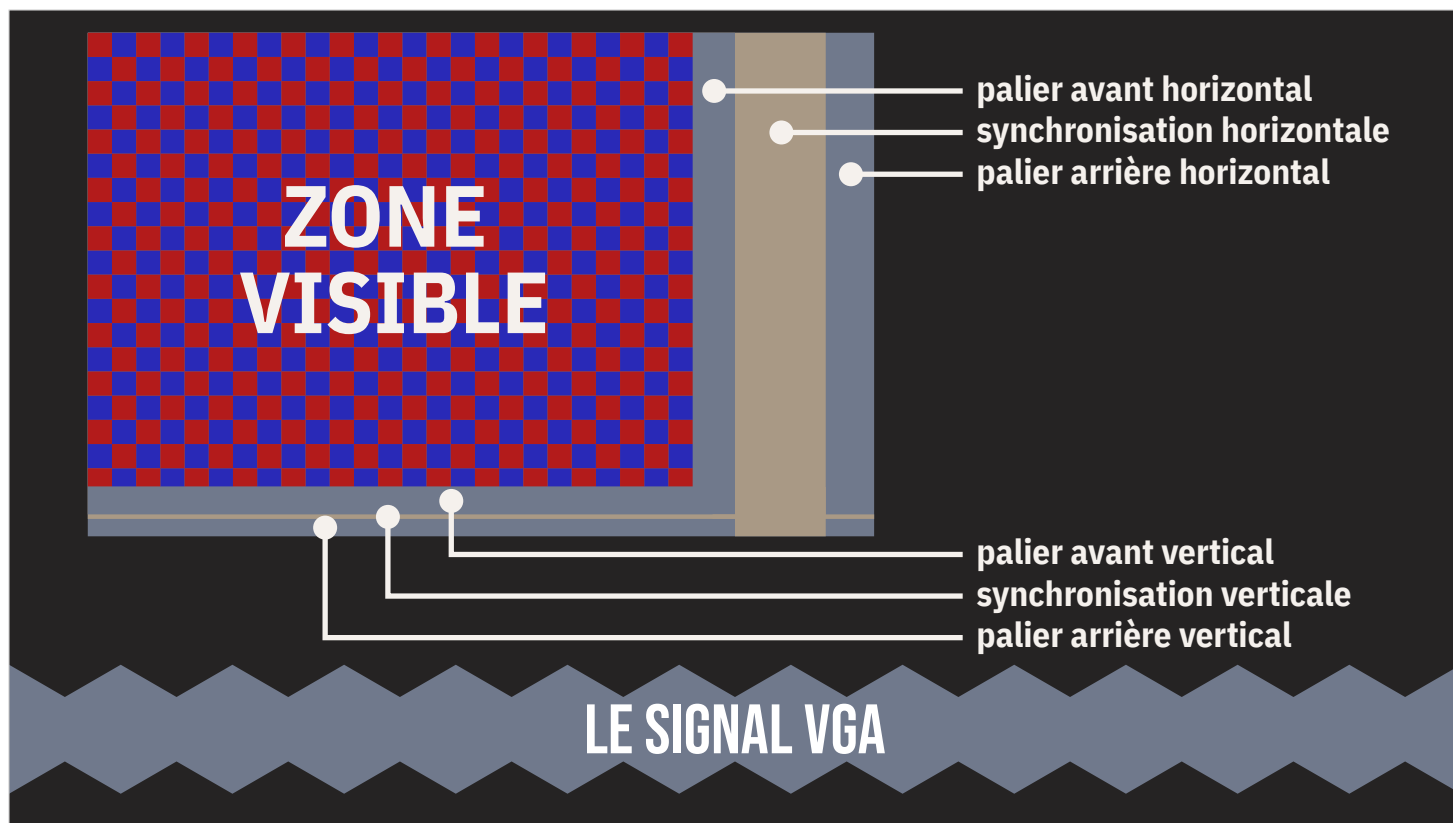
Elles sont nécessaires car notre FPGA fonctionne en 3,3 volts tandis que la norme VGA nécessite une tension maximum de 0,7 volts pour l'intensité maximum de chaque couleur.

C'est encore la norme VGA qui nous permet de placer uniquement 3 résistances en série, mais cela sort du cadre de cette présentation.



Le signal vidéo analogique est hérité de l'époque des écrans à tube cathodique.

Pour pouvoir accommoder les écarts de fréquences, les temps de déplacement du faisceau d'électrons et une zone d'affichage variable sans coins carrés, une partie du signal ne contient aucune information.



Cette partie du signal est divisée en trois temps : le palier avant (front porch), la synchronisation et le palier arrière (back porch).

Générer un signal vidéo va consister à compter des points à 50 mégahertz.

# ÉCRITURE DU CODE VERILOG

- 1) Déclaration du module
- 2) Synchronisation horizontale
- 3) Synchronisation verticale
- 4) Damier rouge et bleu

1

```
module SimpleVGA (  
    input wire clk,  
  
    output wire hsync,  
    output wire vsync,  
    output reg red,  
    output reg green,  
    output reg blue  
);
```

2

```
reg [10:0] xpos = 0;  
always @(posedge clk)  
    if (xpos == 1039) xpos <= 0;  
    else                xpos <= xpos + 1;  
  
assign hsync = xpos < 856 || xpos >= 976;
```

3

```
reg [9:0] ypos = 0;  
always @(posedge clk)  
    if (xpos == 1039) begin  
        if (ypos == 666) ypos <= 0;  
        else                ypos <= ypos + 1;  
    end  
  
assign vsync = ypos < 637 || ypos >= 643;
```

4

```
always @(posedge clk)  
    if (xpos < 800 && ypos < 600) begin  
        red  <= ~xpos[5] ^ ypos[5];  
        green <= 0;  
        blue  <= xpos[5] ^ ypos[5];  
    end else begin  
        red  <= 0;  
        green <= 0;  
        blue  <= 0;  
    end  
  
endmodule
```

Nous allons décrire notre circuit grâce au langage Verilog.

Il se décompose en quatre parties : déclaration du module, synchronisation horizontale, synchronisation verticale et damier rouge et bleu.



## DÉCLARATION DU MODULE

Commençons par la déclaration du module.



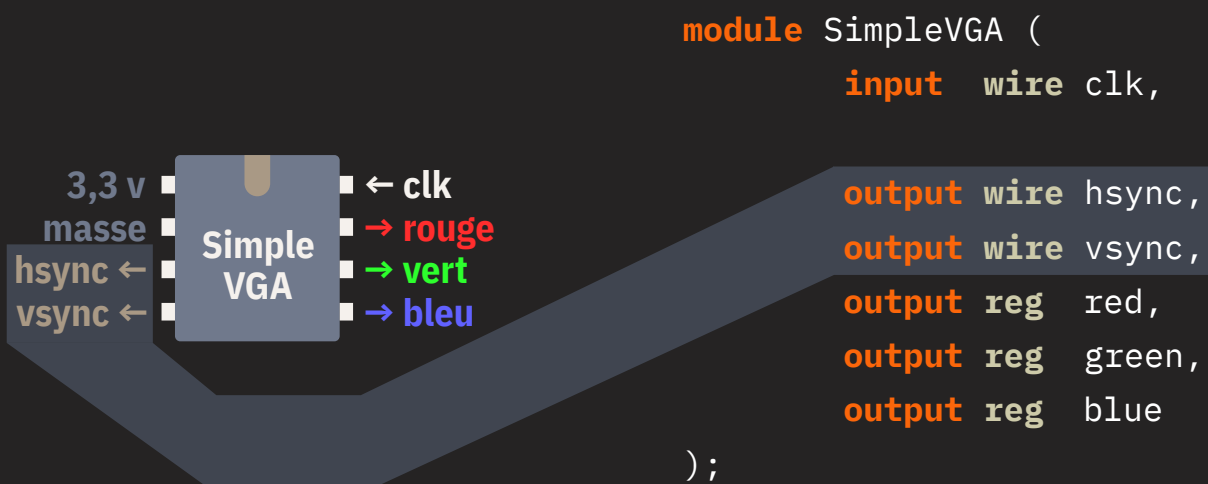
## CRÉATION DU CIRCUIT INTÉGRÉ, HORLOGE

Verilog permet de créer des circuits intégrés qu'on appelle module.

La tension d'alimentation et la masse ne sont jamais représentées car on ne s'intéresse qu'aux éléments logiques (0 ou 1).

En entrée (input), notre module a besoin d'une horloge fonctionnant à 50 mégahertz.



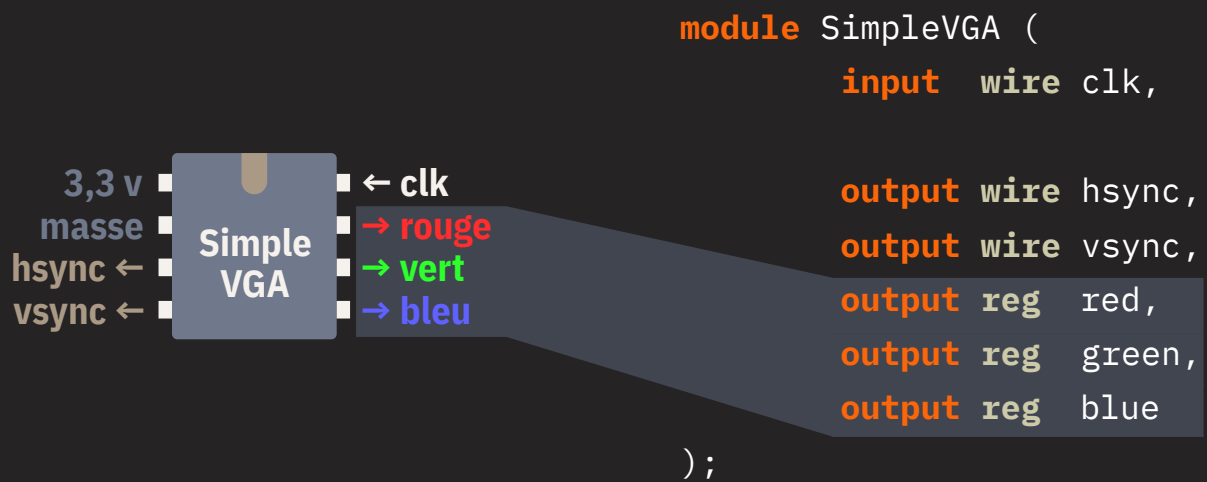


## CRÉATION DU CIRCUIT INTÉGRÉ, SYNCHRONISATION

En sortie (output), il produit les 5 signaux nécessaires au pilotage d'un moniteur VGA.

Ici, nous avons les synchronisations horizontale et verticale.

Le mot-clé wire désigne une valeur calculée en continue mais jamais stockée.



## CRÉATION DU CIRCUIT INTÉGRÉ, COULEURS

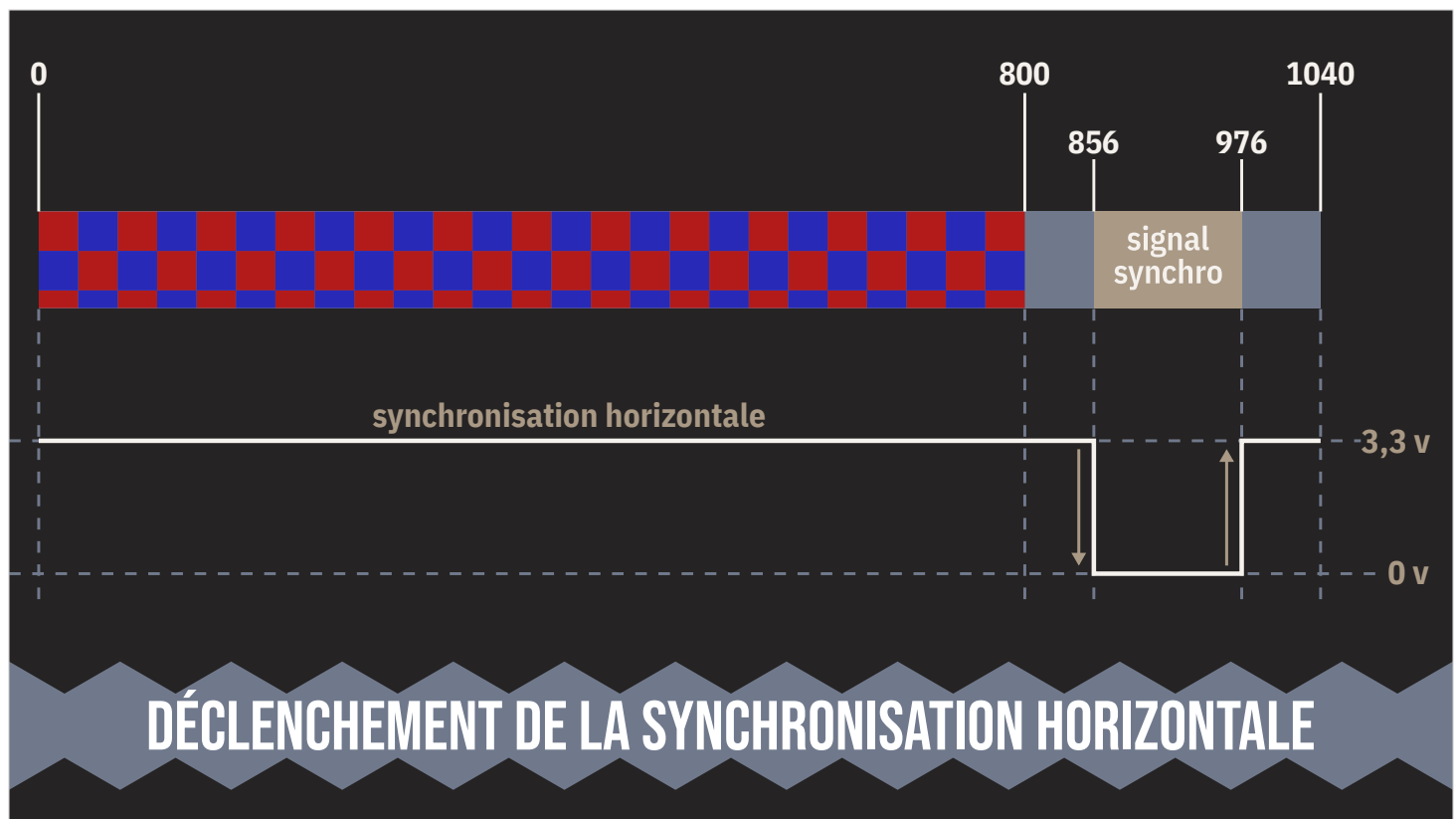
Et nous avons les 3 couleurs rouge, vert et bleu.

Le mot-clé reg désigne une mémoire dont la valeur sera calculée lors d'un événement.



## SYNCHRONISATION HORIZONTALE

Le module étant déclaré, le tout premier élément à calculer est la synchronisation horizontale.



Pour la génération du signal, on utilise les timings donnés par le projet TinyVGA → <http://www.tinyvga.com/vga-timing/800x600@72Hz>

Les 800 premiers points (16 microsecondes) sont dédiés à l'affichage de l'image.

Vient le palier avant pendant 56 points (1,12 microsecondes).

Il est suivi du signal de synchronisation horizontale qui dure 120 points (2,4 microsecondes). Il fonctionne à l'inverse : il est déclenché quand la ligne est à 0 volt.

Enfin arrive le palier arrière qui dure 64 points (1,28 microsecondes).

La ligne complète dure 1040 points (20,8 microsecondes).

```
reg [10:0] xpos = 0;  
always @(posedge clk)  
    if (xpos == 1039) xpos <= 0;  
    else                xpos <= xpos + 1;  
  
assign hsync = xpos < 856 || xpos >= 976;
```

## CODE GÉNÉRANT LA SYNCHRONISATION HORIZONTALE

Comment cela se traduit-il en Verilog ?

Voici la partie du code chargée de cette tâche.

registre de 11 bits = valeurs de 0 à 2047

```
reg [10:0] xpos = 0;  
always @(posedge clk)  
    if (xpos == 1039) xpos <= 0;  
    else                xpos <= xpos + 1;  
  
assign hsync = xpos < 856 || xpos >= 976;
```

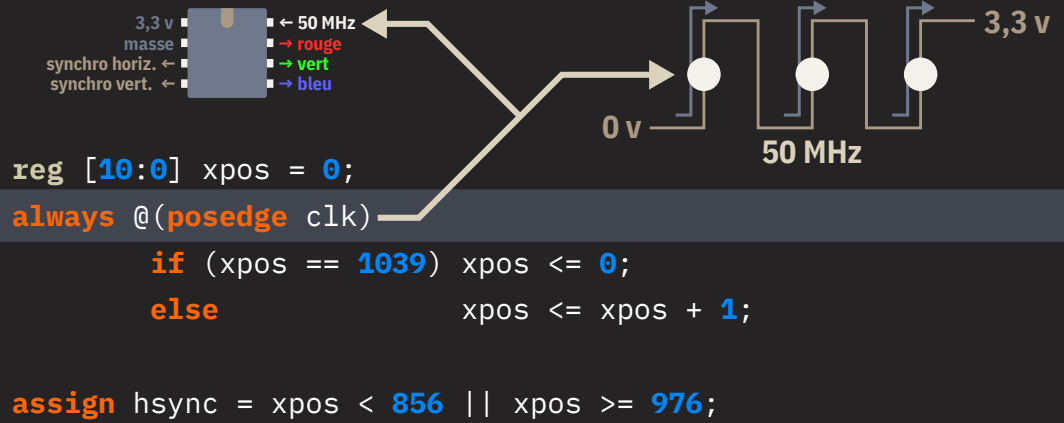
## DÉCLARATION D'UN REGISTRE DE 11 BITS

On commence par créer un registre de 11 bits initialisés à 0.

Générer la synchronisation horizontale nécessite de pouvoir calculer de 0 à 1040.

10 bits permettent de coder des nombres de 0 à 1023, ce qui n'est pas assez.

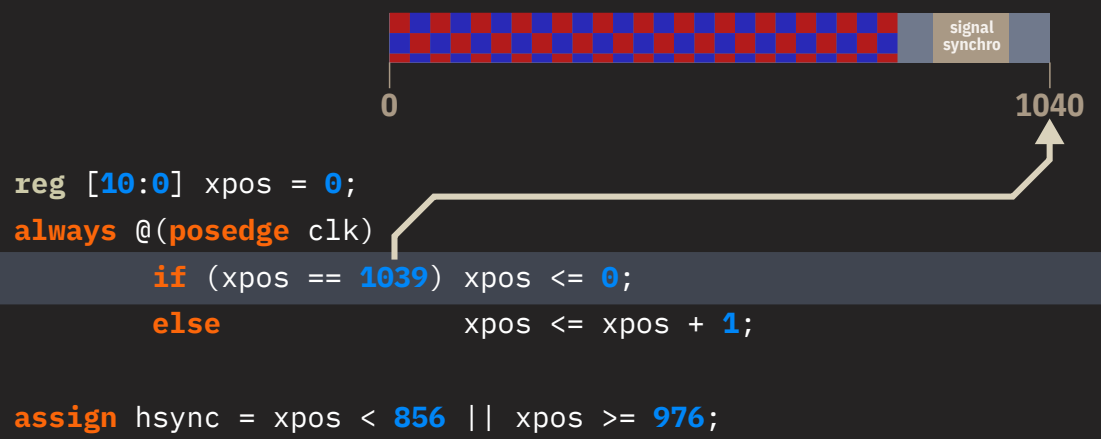
11 bits sont donc nécessaires car ils permettent de compter jusqu'à 2047.



## EXÉCUTION À CHAQUE FRONT MONTANT

Le bloc « always @(posedge clk) » va être exécuté à chaque front montant de l'horloge à 50 MHz.

C'est le passage de 0 volt à 3,3 volts qui va déclencher l'exécution.



REMISE À ZÉRO APRÈS 1040 POINTS

Si notre compteur arrive au dernier point, on le remet à zéro.

Une nouvelle ligne commence.



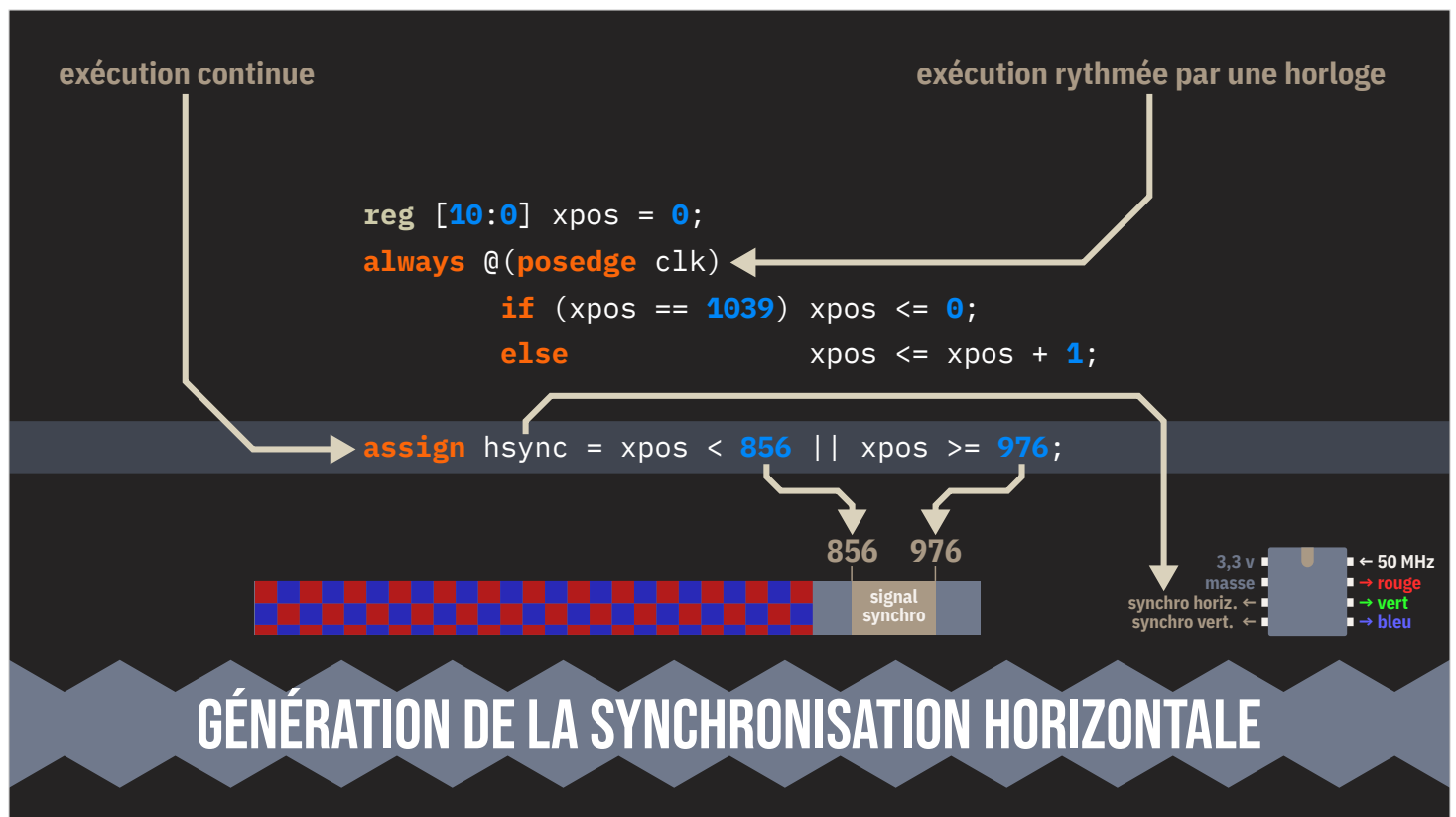


```
reg [10:0] xpos = 0;
always @(posedge clk)
    if (xpos == 1039) xpos <= 0;
    else                xpos <= xpos + 1;

assign hsync = xpos < 856 || xpos >= 976;
```

COMPTAGE DE 0 À 1039

Sinon, le compteur continue d'être incrémenté tous les 50 millionièmes de seconde.



La ligne qui génère le signal de synchronisation horizontale fonctionne différemment.

Elle est exécutée en continu, c'est-à-dire qu'elle n'est basée sur aucune horloge et fonctionne aussi rapidement que l'électronique le permet.

Le signal fonctionnant en inverse, il doit être à 0 volt pour indiquer un déclenchement. Le reste du temps, il doit être à 3,3 volts.



## SYNCHRONISATION VERTICALE

Le calcul de la synchronisation horizontale étant réglé, on s'occupe de la synchronisation verticale.

```
reg [9:0] ypos = 0;
always @(posedge clk)
    if (xpos == 1039) begin
        if (ypos == 665) ypos <= 0;
        else             ypos <= ypos + 1;
    end

assign vsync = ypos < 637 || ypos >= 643;
```

## CODE GÉNÉRANT LA SYNCHRONISATION VERTICALE

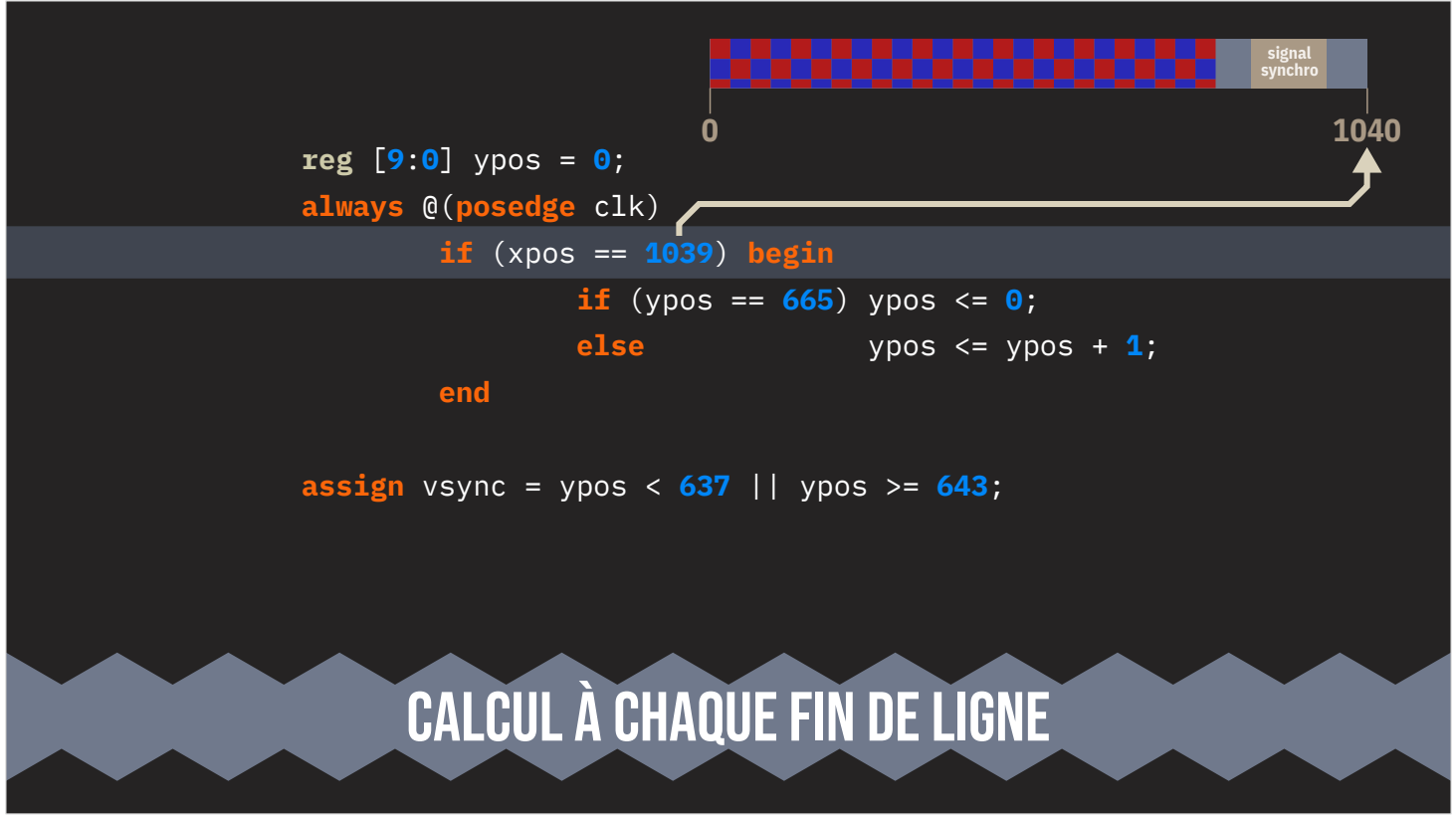
Le code générant la synchronisation verticale est très proche de celui générant la synchronisation horizontale.

registre de 10 bits = valeurs de 0 à 1023

```
reg [9:0] ypos = 0;  
always @(posedge clk)  
    if (xpos == 1039) begin  
        if (ypos == 665) ypos <= 0;  
        else ypos <= ypos + 1;  
    end  
  
assign vsync = ypos < 637 || ypos >= 643;
```

## DÉCLARATION D'UN REGISTRE DE 10 BITS

Comme une image est plus large que haute, seuls 10 bits sont nécessaires pour conserver le numéro de la ligne en cours.



```

reg [9:0] ypos = 0;
always @(posedge clk)
    if (xpos == 1039) begin
        if (ypos == 665) ypos <= 0;
        else
            ypos <= ypos + 1;
    end

assign vsync = ypos < 637 || ypos >= 643;

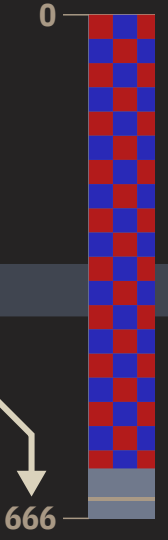
```

**CALCUL À CHAQUE FIN DE LIGNE**

On ne calcule le numéro de la ligne qu'à chaque fin de ligne.

```
reg [9:0] ypos = 0;
always @(posedge clk)
    if (xpos == 1039) begin
        if (ypos == 665) ypos <= 0;
        else
            ypos <= ypos + 1;
    end

assign vsync = ypos < 637 || ypos >= 643;
```



REMISE À ZÉRO APRÈS 666 LIGNES

Il y a 666 lignes à compter. Si le compte est atteint, on repart à zéro.

```
reg [9:0] ypos = 0;
always @(posedge clk)
    if (xpos == 1039) begin
        if (ypos == 665) ypos <= 0;
        else ypos <= ypos + 1;
    end

assign vsync = ypos < 637 || ypos >= 643;
```



COMPTAGE DE 0 À 665

Si le compte n'est pas atteint, on continue d'incrémenter le numéro de ligne.



```

reg [9:0] ypos = 0;
always @(posedge clk)
    if (xpos == 1039) begin
        if (ypos == 665) ypos <= 0;
        else
            ypos <= ypos + 1;
    end

```

```

assign vsync = ypos < 637 || ypos >= 643;

```

3,3 v  
 masse  
 synchro horiz. ←  
 synchro vert. ←

← 50 MHz  
 → rouge  
 → vert  
 → bleu

637  
643

## GÉNÉRATION DE LA SYNCHRONISATION VERTICALE

Le signal de synchronisation verticale est déclenché quand le numéro de ligne est compris entre 637 et 643.



## DAMIER ROUGE ET BLEU



Maintenant que les synchronisations horizontale et verticale sont générées, nous pouvons nous attaquer à la génération du damier rouge et bleu.

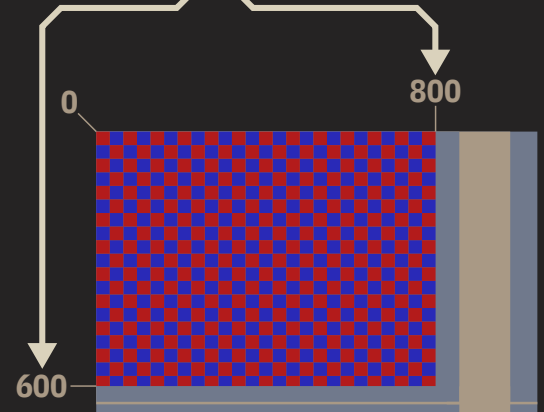
```
always @(posedge clk)
    if (xpos < 800 && ypos < 600) begin
        red    <= ~xpos[5] ^ ypos[5];
        green  <= 0;
        blue   <= xpos[5] ^ ypos[5];
    end else begin
        red    <= 0;
        green  <= 0;
        blue   <= 0;
    end
end
```

## CODE GÉNÉRANT LE DAMIER ROUGE ET BLEU

À nouveau, nous utilisons un bloc synchronisé sur l'horloge à 50 mégahertz.

Les trois blocs que nous avons créés fonctionnent tous parfaitement en parallèle.

```
always @(posedge clk)
  if (xpos < 800 && ypos < 600) begin
    red  <= ~xpos[5] ^ ypos[5];
    green <= 0;
    blue <= xpos[5] ^ ypos[5];
  end else begin
    red  <= 0;
    green <= 0;
    blue <= 0;
  end
end
```



## TEST DE LA ZONE VISIBLE

Le damier ne devant être affiché que dans la zone visible, il suffit de tester si la position X est inférieure à 800 et la position Y inférieure à 600.

```
always @(posedge clk)
    if (xpos < 800 && ypos < 600) begin
        red    <= ~xpos[5] ^ ypos[5];
        green  <= 0;
        blue   <= xpos[5] ^ ypos[5];
    end else begin
        red    <= 0;
        green  <= 0;
        blue   <= 0;
    end
end
```

les 3 calculs et affectations  
sont exécutés en parallèle

## UN FPGA TRAVAILLE EN PARALLÈLE

Encore une fois, un FPGA est extrêmement parallèle.

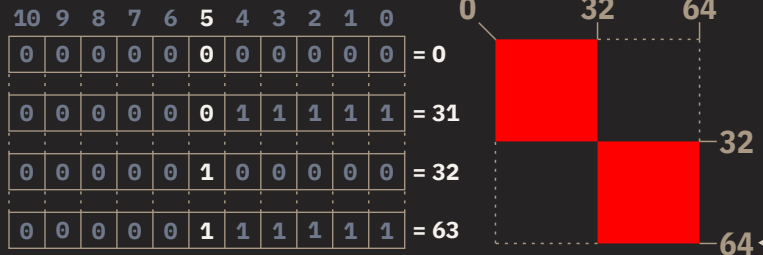
Les calculs et affectations de cette partie du bloc seront exécutées en même temps.

Cela veut dire qu'en un cycle d'horloge, les composantes rouge, vert et bleu seront calculées et affectées à leurs registres respectifs.

```

always @(posedge clk)
    if (xpos < 800 && ypos < 600) begin
        red    <= ~xpos[5] ^ ypos[5];
        green  <= 0;
        blue   <= xpos[5] ^ ypos[5];
    end else begin
        red    <= 0;
        green  <= 0;
        blue   <= 0;
    end
end

```



## ALTERNER LES CARRÉS ROUGES

Pour pouvoir alterner les carrés rouges du damier, on recourt à une astuce qui consiste à utiliser le sixième bit de la position X et de la position Y.

Avec 6 bits, on peut écrire des nombres de 0 à 63 (2 puissance 6 égale 64).

Quand on compte de 0 à 63, le sixième bit sera à 0 de 0 à 31 et à 1 de 32 à 63. Cela nous permet de créer des carrés de 32 pixels de large.

En utilisant un ou exclusif (^) avec le sixième bit de la position Y, on peut alterner les carrés rouges toutes les 32 lignes.

En utilisant un non logique (~), cela permet au tout premier carré d'être rouge.

```
always @(posedge clk)
    if (xpos < 800 && ypos < 600) begin
        red    <= ~xpos[5] ^ ypos[5];
        green  <= 0;
        blue   <= xpos[5] ^ ypos[5];
    end else begin
        red    <= 0;
        green  <= 0;
        blue   <= 0;
    end
end
```

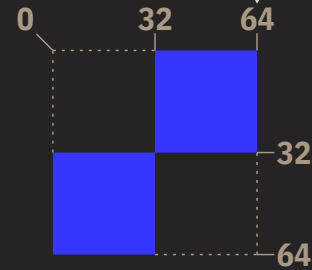
LE VERT N'EST JAMAIS UTILISÉ

Le vert n'étant jamais utilisé, on le force à zéro.

```

always @(posedge clk)
    if (xpos < 800 && ypos < 600) begin
        red    <= ~xpos[5] ^ ypos[5];
        green  <= 0;
        blue   <= xpos[5] ^ ypos[5];
    end else begin
        red    <= 0;
        green  <= 0;
        blue   <= 0;
    end
end

```



## ALTERNER LES CARRÉS BLEUS

Pour alterner les carrés bleus, on procède exactement comme pour les carrés rouges.

On évite de mélanger le rouge et le bleu en omettant l'inversion sur la position X, forçant le carré bleu à commencer à partir de la 32<sup>e</sup> colonne.



```
always @(posedge clk)
    if (xpos < 800 && ypos < 600) begin
        red    <= ~xpos[5] ^ ypos[5];
        green  <= 0;
        blue   <= xpos[5] ^ ypos[5];
    end else begin
        red    <= 0;
        green  <= 0;
        blue   <= 0;
    end
end
```

**AUCUNE COULEUR EN DEHORS DE LA ZONE VISIBLE**

Pour toute coordonnée en dehors de la zone visible, les composantes sont forcées à zéro.

## REMARQUES ET PIÈGES

- Verilog n'est pas un langage procédural
- Un bit peut avoir 4 états
  - 0, 1, X et Z → faux, vrai, inconnu, impédance haute
- Vous êtes responsables des délais de propagation
- Générer une image « bitstream » est long...
- Tous les codes HDL ne sont pas synthétisables

Quelques remarques et pièges pour les débutants.

On est facilement tenté de considérer Verilog comme un langage procédural, au même titre que le C ou le PHP. Mais il n'en est rien ! Il s'agit bien d'un langage de description de matériel.

Il faut également penser à bien initialiser les registres. Un bit peut avoir 4 états, 0, 1, X pour inconnu et Z pour impédance haute.

Tenter d'incrémenter un registre dont l'état est inconnu, donnera un état inconnu.

Vous êtes responsable des délais de propagation. Chaque élément utilisé pour un calcul nécessite un délai pour changer d'état. Plus on enchaîne d'éléments, plus le délai s'allonge.

Générer une image « bitstream » est un processus long, très long. N' imaginez pas un cycle du genre « coder, compiler, tester, coder... ».

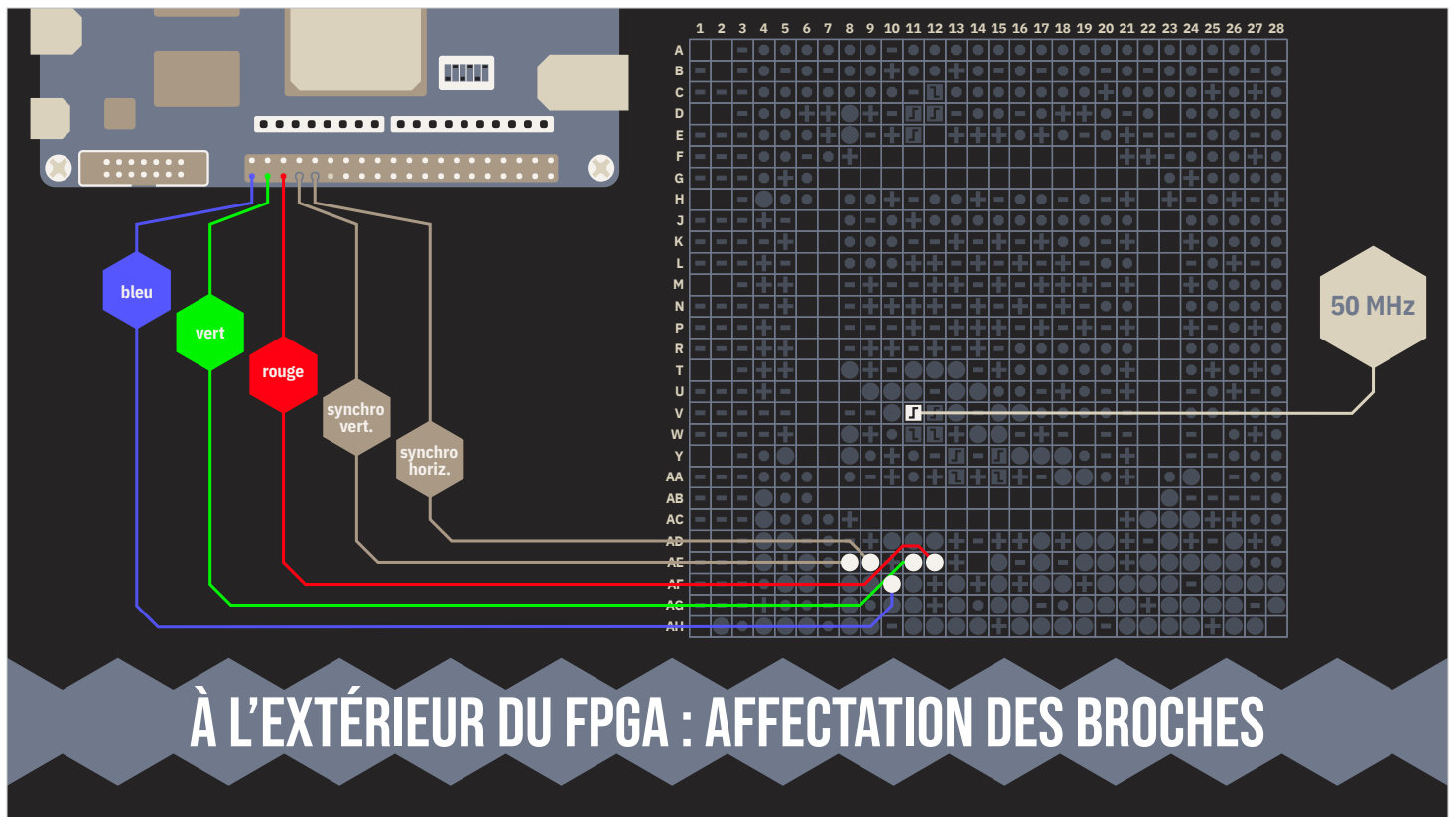
Et tous les codes HDL que vous pouvez écrire ne sont pas synthétisables.



## GÉNÉRATION DE L'IMAGE « BITSTREAM »



Le code de notre circuit étant écrit, il faut maintenant générer l'image « bitstream » qui correspond à la configuration du réseau interne du FPGA.



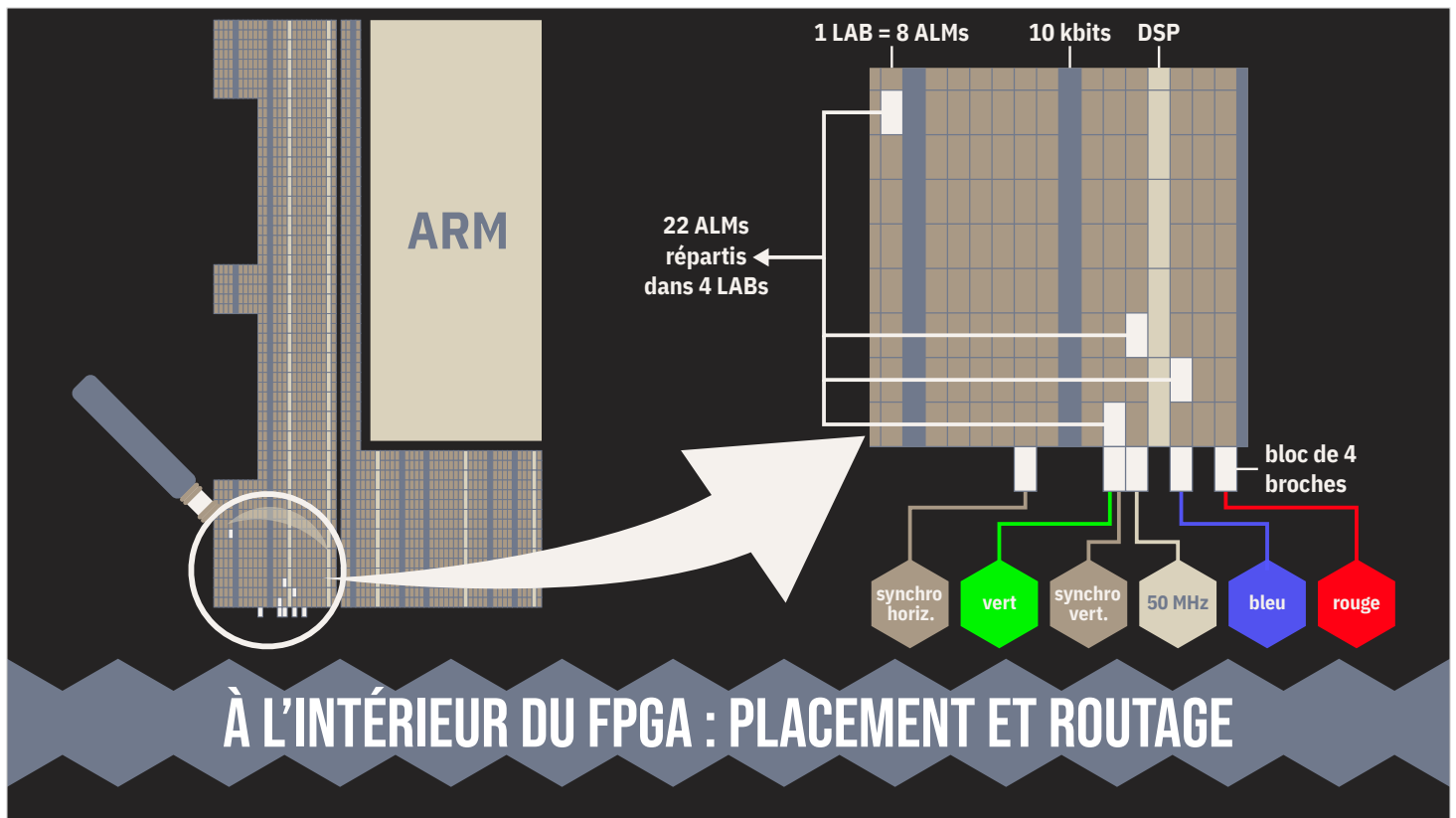
Mais avant cela, il faut indiquer à Quartus Prime les broches que nous utiliserons.

La grille représente la position physique des broches sous le FPGA.

Il n'est pas toujours aisé de s'y retrouver étant donné le grand nombre de broches disponibles et la correspondance broche du FPGA et broches du DE0-Nano-SoC (Quartus Prime ne connaît que les broches du FPGA).

Certaines broches ont une fonction dédiée, d'autres peuvent être configurées.

L'affectation des broches est importantes pour deux raisons : d'une part elle a des implications quant à l'utilisation finale du FPGA, d'autre part elle va déterminer les ALMs qui seront utilisés pour réaliser notre circuit.

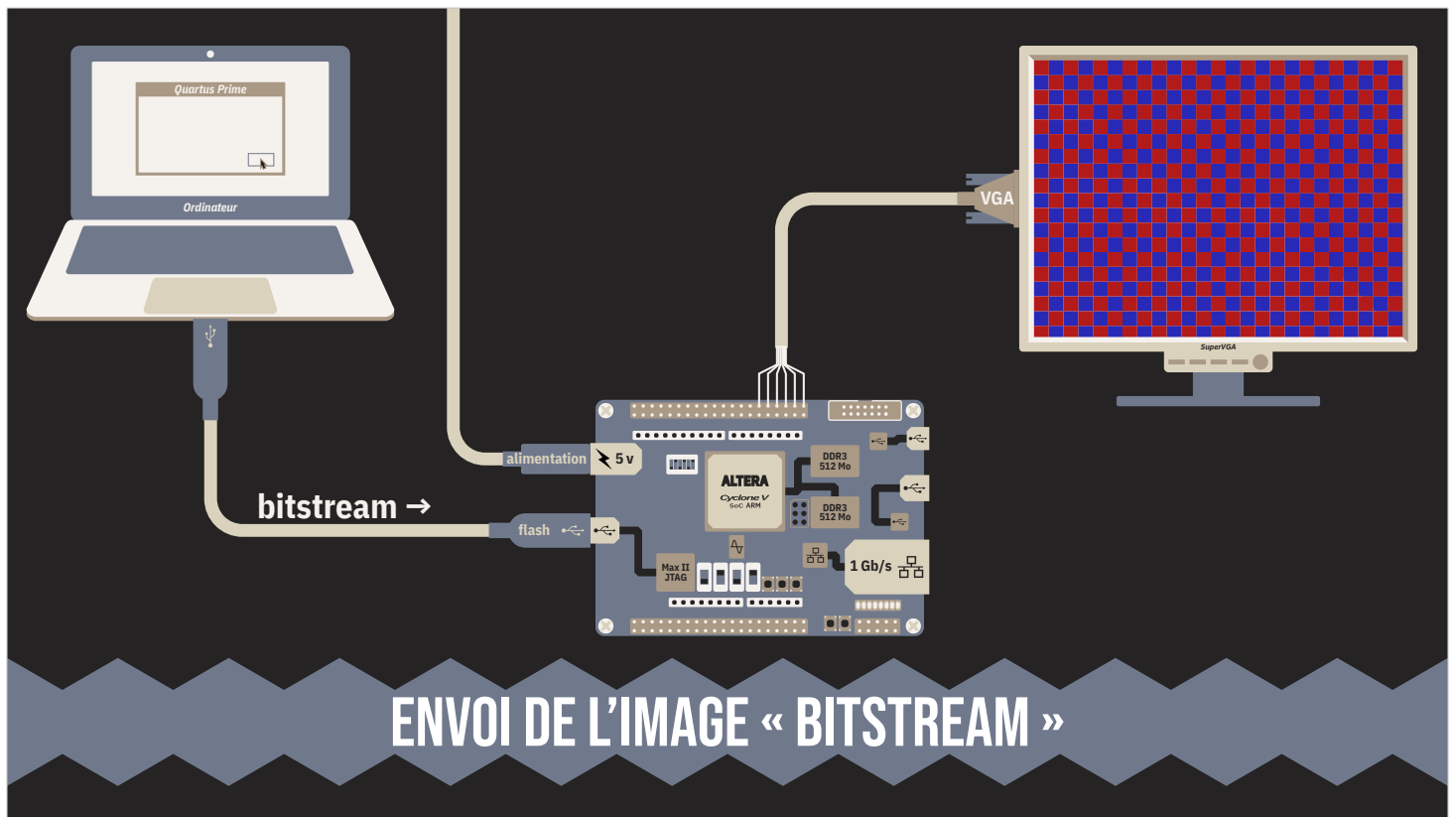


Une fois les broches sélectionnées et la netlist créée, l'étape de placement et routage a lieu.

Elle va consister pour Quartus Prime à sélectionner les ALMs répondant au mieux aux besoins de notre design et aux contraintes du FPGA.

L'étape prend en compte la position des broches, les besoins en ressources dédiées (mémoire, DSP), le réseau à mettre en place ou des contraintes imposées comme la température de fonctionnement (le FPGA recevra-t-il un système de refroidissement ?).

Quartus Prime a évalué que le circuit SimpleVGA nécessitait 22 ALMs qu'il répartit dans 4 LABs (un LAB est un bloc de 8 ALMs). Le circuit ne nécessite aucune mémoire ni de DSP (Quartus Prime peut recourir à des DSP notamment pour effectuer des multiplications rapides).



L'image « bitstream » est envoyée via la prise USB dédiée (d'autres FPGA ont une prise spécifique et requiert un matériel additionnel).

Il est possible d'envoyer une image temporaire qui sera oubliée à l'extinction.

On peut aussi envoyer une image permanente qui sera utilisée à chaque allumage du FPGA.

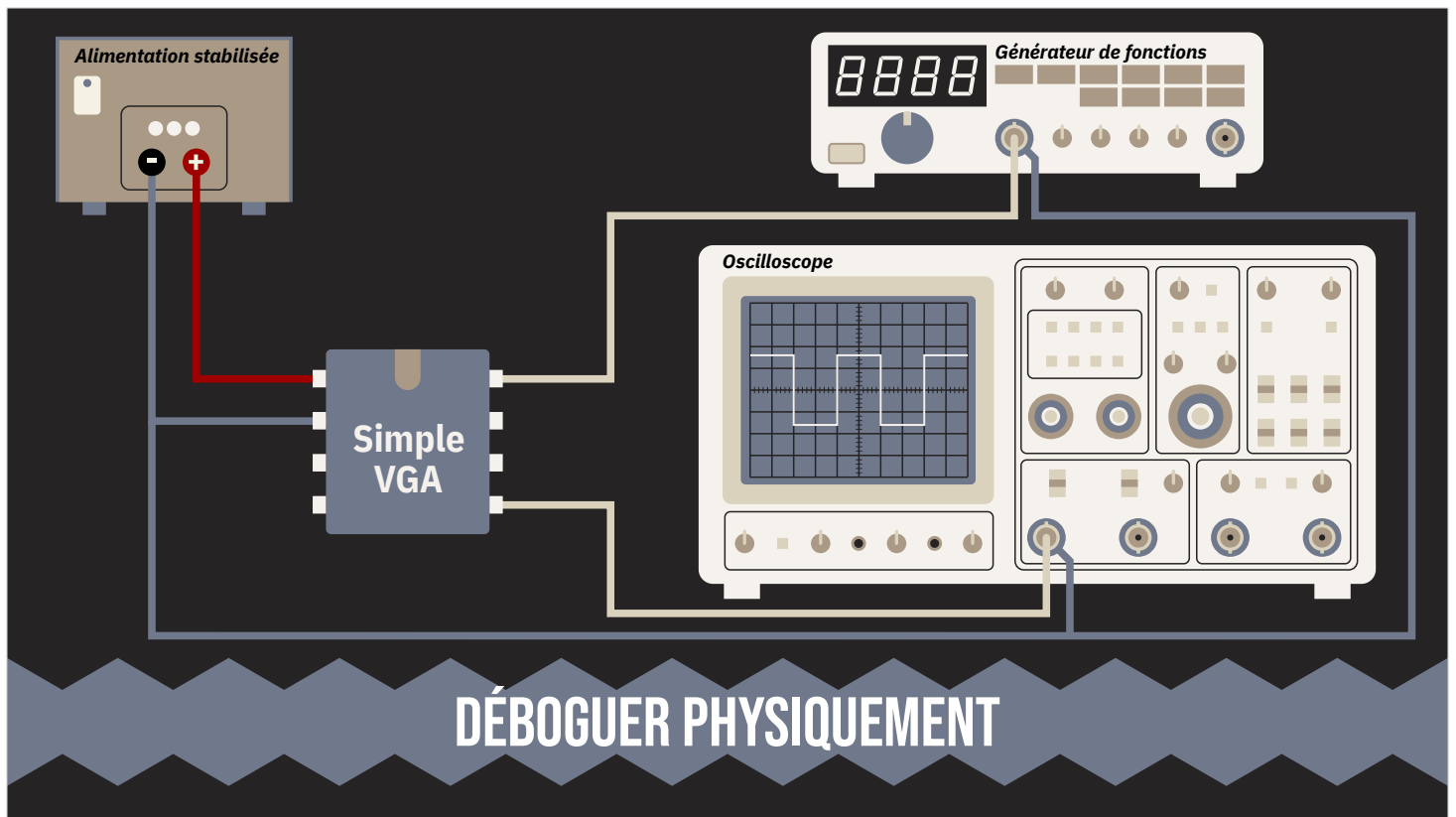
Cette dernière option permet d'obtenir un circuit complètement autonome, sans système d'exploitation, avec un temps d'initialisation très court.

# DÉBOGUER UN CIRCUIT

Le code Verilog est comme tous les autres codes : il est rare que tout fonctionne du premier coup.

À la différence d'un programme exécuté par un processeur, il n'est pas possible d'interrompre le fonctionnement d'un circuit.

Mais alors, comment débogue-t-on un FPGA ?



Pour déboguer un circuit physique, des appareils spécialisés sont nécessaires comme une alimentation stabilisée, un générateur de fonctions, un oscilloscope etc.

Mais générer une image « bitstream » prend du temps.

Il est plus intéressant de travailler avec des bancs d'essai ou test bench.



```
Icarus Verilog

module SimpleVGA (
    input wire clk,
    output wire hsync,
    output wire vsync,
    output reg red,
    output reg green,
    output reg blue
);

reg [10:0] xpos = 0;
always @(posedge clk)
    if (xpos == 1023) xpos <= 0;
    else
        xpos <= xpos + 1;

assign hsync = xpos < 856 || xpos >= 976;

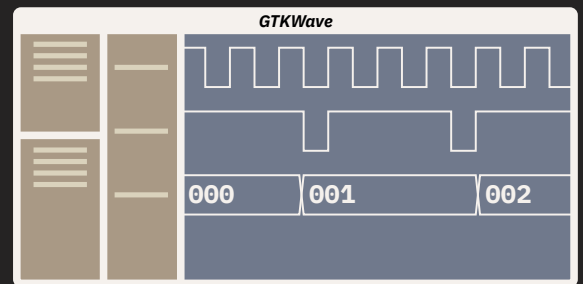
reg [10:0] ypos = 0;
always @(posedge clk)
    if (ypos == 645) ypos <= 0;
    else
        ypos <= ypos + 1;

assign vsync = ypos < 637 || ypos >= 643;

always @(posedge clk)
    if (xpos < 800 && ypos < 400) begin
        red <= xpos[1] ^ ypos[1];
        green <= 0;
        blue <= ~xpos[0] ^ ypos[0];
    end else begin
        red <= 0;
        green <= 0;
        blue <= 0;
    end
end

endmodule
```

**SimpleVGA + Banc d'essai**



## SIMULER POUR DÉBOGUEUR

Un banc d'essai est un programme lui aussi écrit en Vérilog.

Le logiciel Icarus Verilog est en mesure de compiler le module et son banc d'essai et de simuler le fonctionnement du circuit sur temps donné.

Il peut même générer un enregistrement de tous les changements de signaux dans le circuit.

Cet enregistrement peut alors être lu par GTKWave afin d'analyser le comportement de chaque signal (il joue le rôle de l'oscilloscope).

Quartus Prime Lite permet d'utiliser ModelSim ASE mais ce logiciel est plus complexe et donc moins évident à présenter ici.

# ORGANISATION DU BANC D'ESSAI

- 1) Génération d'une horloge
- 2) Instanciation de SimpleVGA
- 3) Lancement de la simulation
- 4) Création d'un compteur

1

```
timescale 10ns / 10ns
module SimpleVGA_tb;
  reg clk = 0;
  always #1 clk = !clk;

  wire hsync;
```

2

```
SimpleVGA testbench(
  .clk (clk),
  .hsync (hsync)
);
```

3

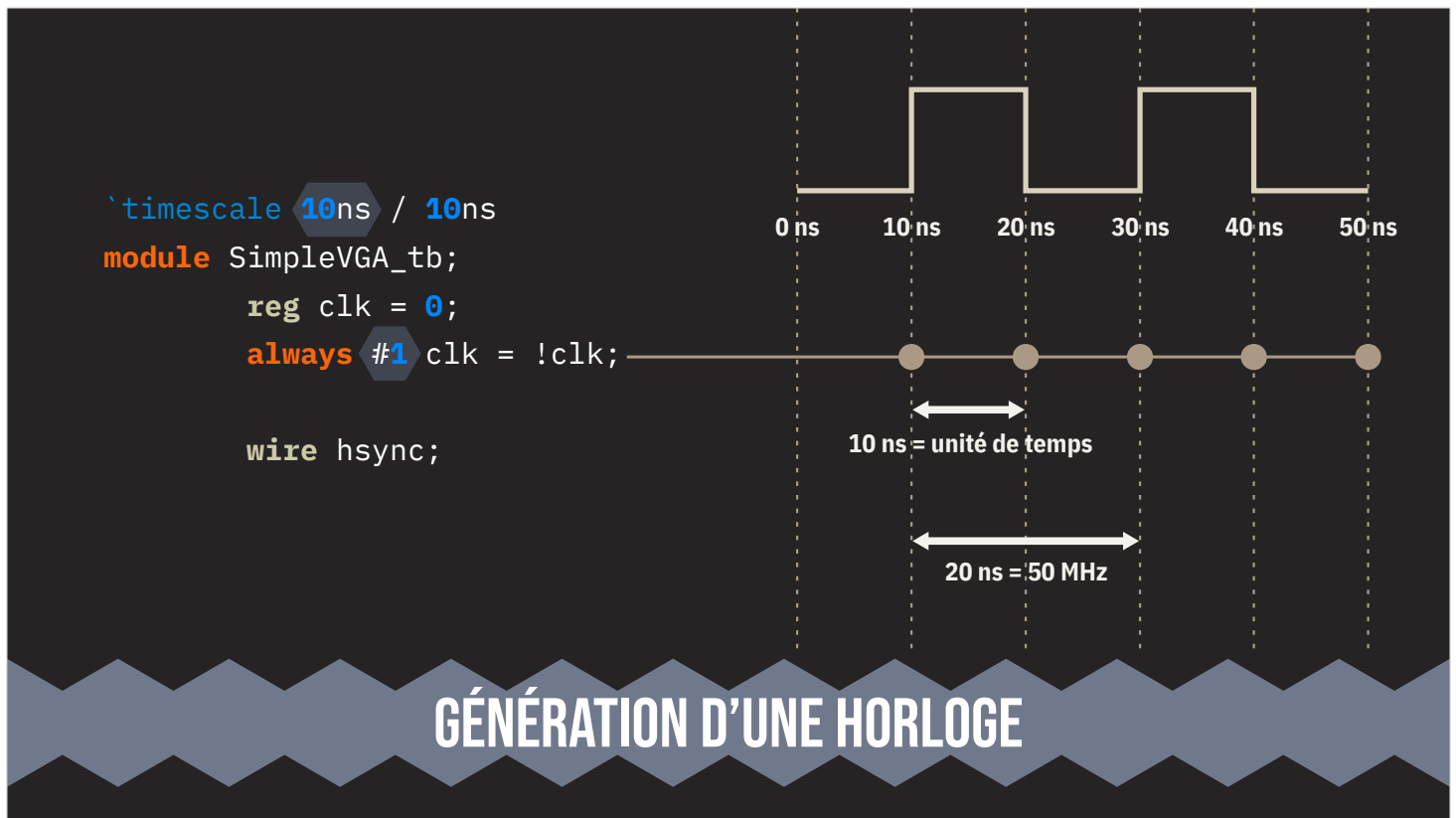
```
initial begin
  $dumpfile("SimpleVGA_tb.vcd");
  $dumpvars(0, testbench);

  #1385280 $finish;
end
```

4

```
reg [24:0] counter = 0;
always @(negedge hsync) begin
  counter = counter + 1;
  $display("HSYNC = %0d @ %0t ns", counter, $time);
end
endmodule
```

Le banc d'essai présenté comporte 4 parties : la génération d'une horloge, l'instanciation du circuit SimpleVGA, le lancement de la simulation et la création d'un compteur.



Le banc d'essai doit générer une horloge qu'on fournira au circuit à tester.

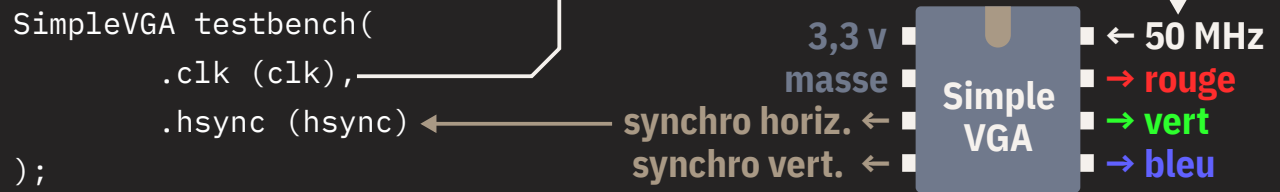
Il recourt à une syntaxe particulière de Verilog.

La directive « `timescale » indique l'unité de temps, ici 10 nanosecondes (1 nanoseconde = 1 milliardième de seconde).

Le dièse suivi d'un nombre indique un délai (multiple de l'unité de temps) avant d'exécuter ce qui suit.

« #1 clk = !clk » va donc faire alterner le registre clk toutes les 10 nanosecondes, ce qui donne un cycle de 20 nanosecondes ou une fréquence de 50 mégahertz.

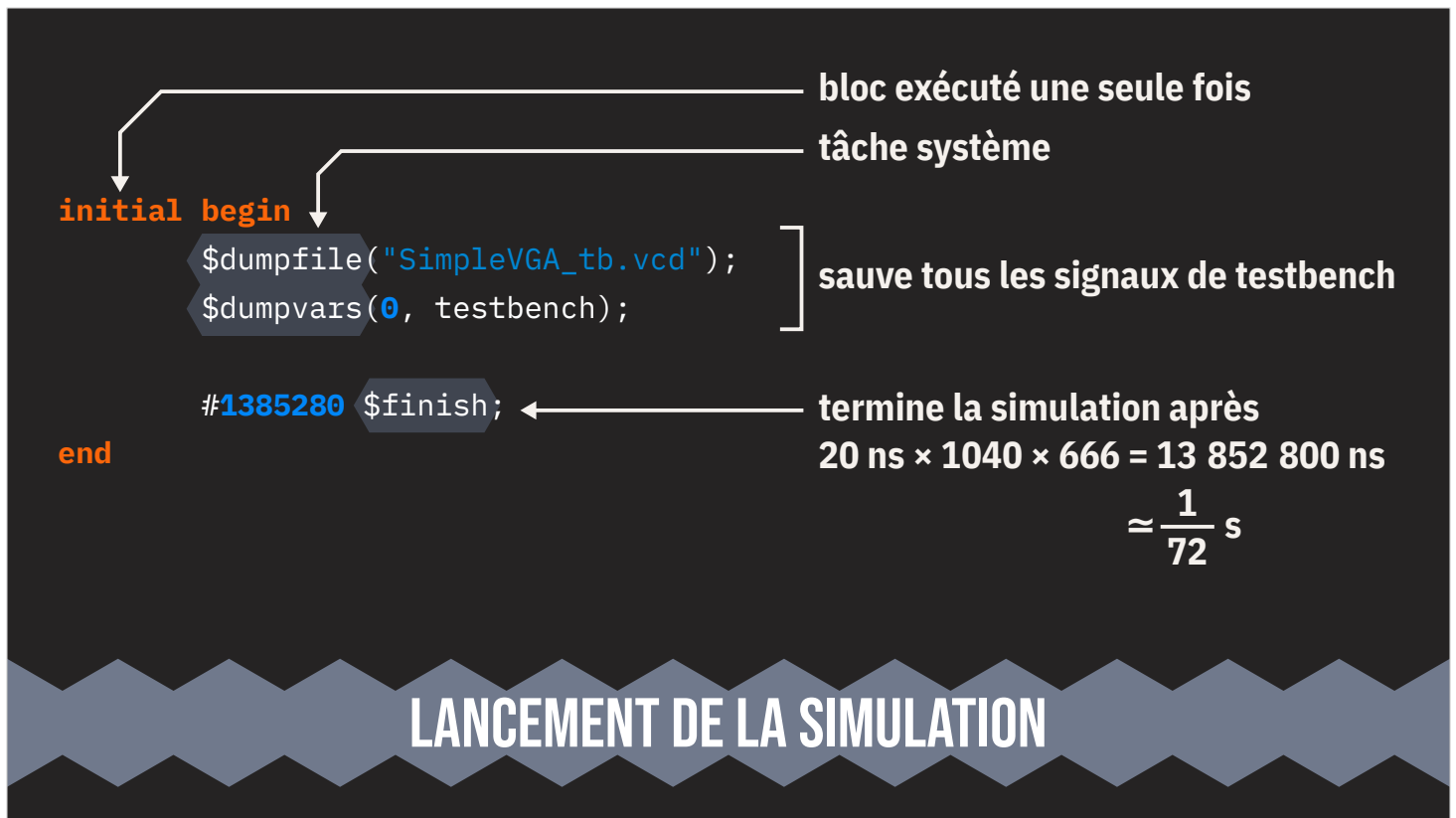
Pour ce banc d'essai, on déclare une ligne hsync utilisée pour notre test.



## INSTANCIATION DE SIMPLEVGA

Le banc d'essai utilise une instance du module SimpleVGA.

Seules les broches nécessaires sont reliées.



Pour le lancement de la simulation, on utilise un bloc « initial » qui ne sera exécuté qu'une seule fois.

Les commandes commençant par un « \$ » sont des tâches système.

« \$dumpfile » et « \$dumpvars » demande au simulateur d'enregistrer l'évolution des signaux du module « testbench » dans le fichier « SimpleVGA\_tb.vcd ».

Enfin, au bout de 13,8 millisecondes, on stoppe la simulation.

```
reg [24:0] counter = 0;
always @(negedge hsync) begin
    counter = counter + 1;
    $display("HSYNC = %0d @ %0t ns", counter, $time);
end
```

↑ affichage du compteur

## CRÉATION D'UN COMPTEUR

Un registre de 25 bits est utilisé pour compter le nombre de signaux de synchronisation horizontale déclenchés pendant la simulation.

Étant donné que la simulation est prévue pour tourner le temps d'une seule trame, la dernière valeur affichée devrait être 666.

*Ligne de commande*

```
$ iverilog -o SimpleVGA_tb SimpleVGA_tb.v SimpleVGA.v
```

## COMPILATION DU BANC D'ESSAI

On utilise Icarus Verilog pour compiler le banc d'essai.

Ligne de commande

```
$ iverilog -o SimpleVGA_tb SimpleVGA_tb.v SimpleVGA.v  
$ ./SimpleVGA_tb  
VCD info: dumpfile SimpleVGA_tb.vcd opened for output.  
HSYNC = 1 @ 1711 ns  
HSYNC = 2 @ 3791 ns  
HSYNC = 3 @ 5871 ns  
# ...  
HSYNC = 664 @ 1380751 ns  
HSYNC = 665 @ 1382831 ns  
HSYNC = 666 @ 1384911 ns
```

## EXÉCUTION DE LA SIMULATION

On peut alors exécuter le banc d'essai.

Celui-ci nous informe d'abord qu'il a bien pris en compte notre demande d'enregistrement des signaux.

Il exécute enfin la simulation et nous affiche les messages programmés avec « \$display ».

On constate alors que nos 666 signaux de synchronisation horizontale ont bel et bien été générés.

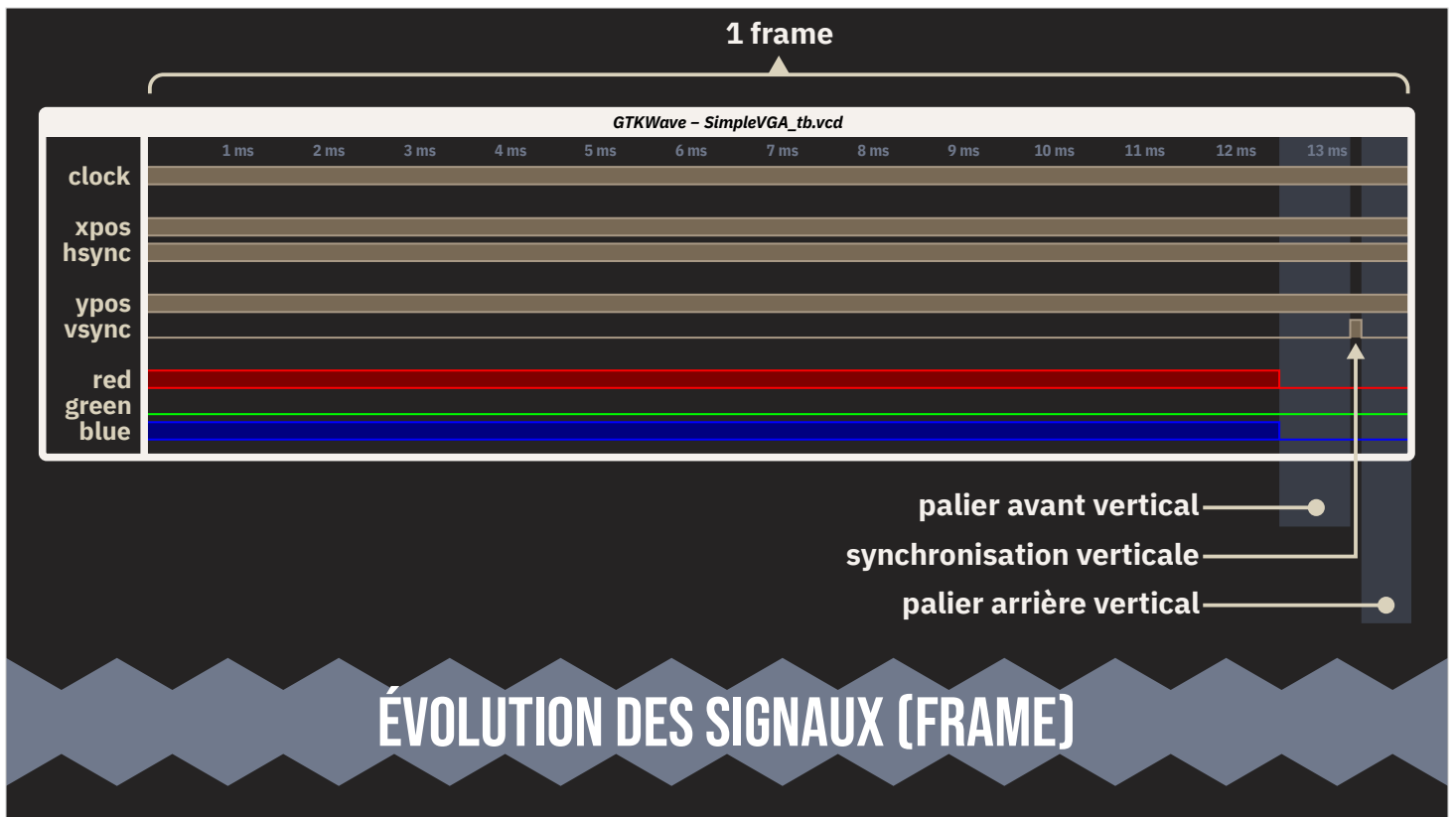


*Ligne de commande*

```
$ iverilog -o SimpleVGA_tb SimpleVGA_tb.v SimpleVGA.v  
$ ./SimpleVGA_tb  
VCD info: dumpfile SimpleVGA_tb.vcd opened for output.  
HSYNC = 1 @ 1711 ns  
HSYNC = 2 @ 3791 ns  
HSYNC = 3 @ 5871 ns  
# ...  
HSYNC = 664 @ 1380751 ns  
HSYNC = 665 @ 1382831 ns  
HSYNC = 666 @ 1384911 ns  
$ gtkwave SimpleVGA_tb.vcd
```

## LANCEMENT DE GTKWAVE

Il ne reste plus qu'à lancer GTKWave pour analyser les signaux.



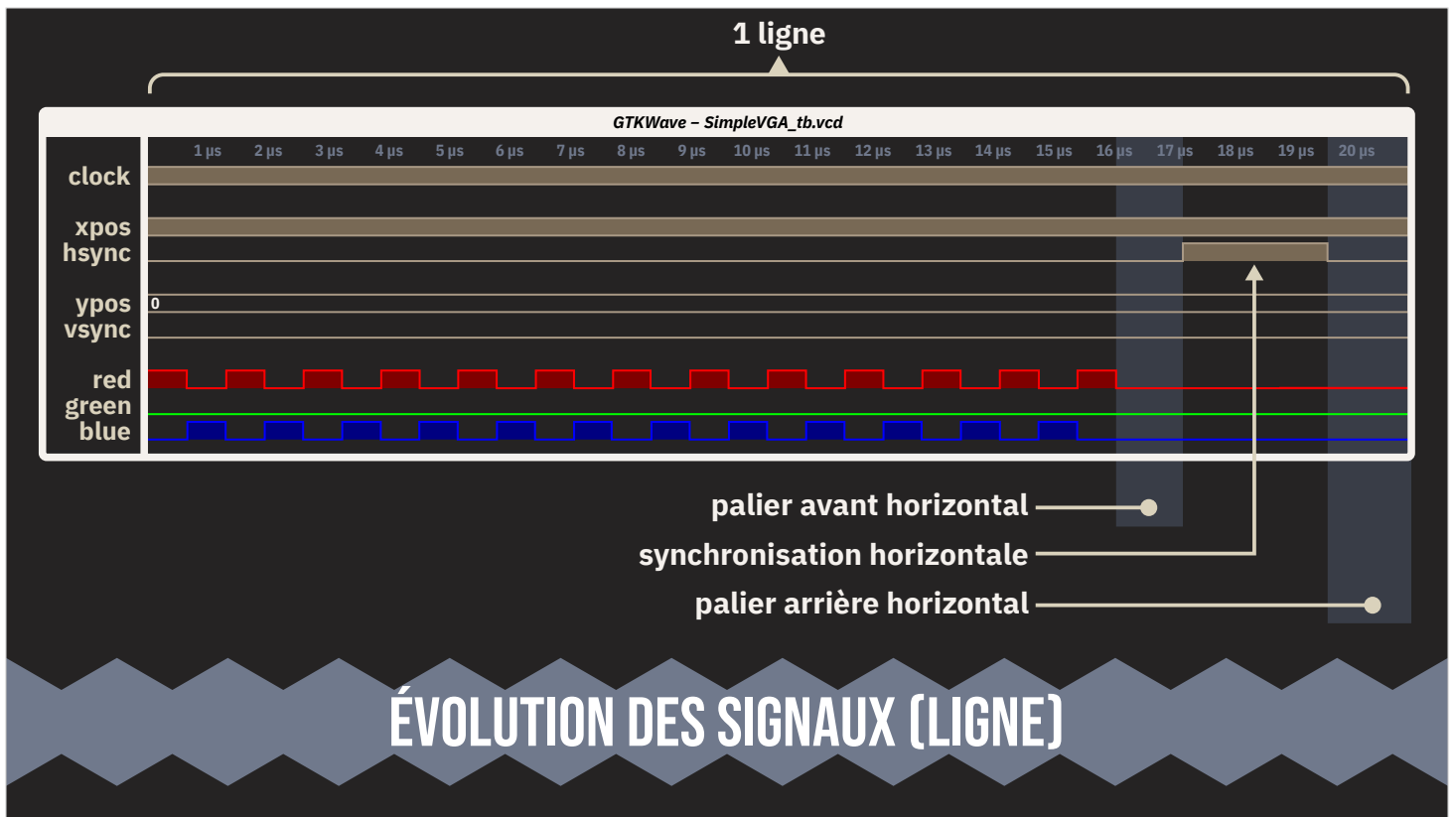
GTKWave permet de sélectionner les signaux à afficher.

Étant donnée la quantité d'informations, les signaux sont étudiés en variant le niveau de zoom.

Au niveau de zoom le plus large, on peut seulement étudier le signal de synchronisation verticale.

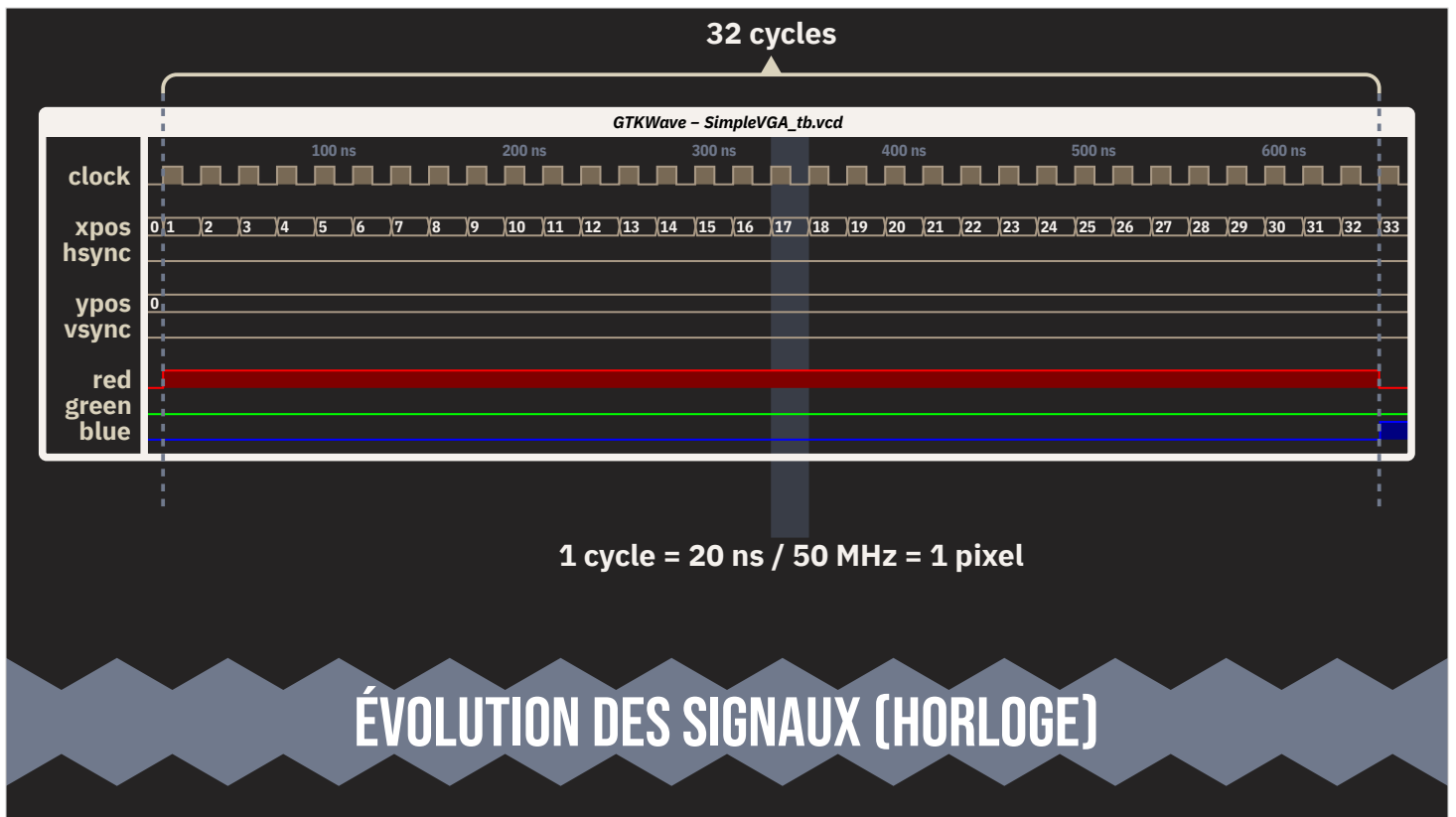
GTKWave permet de configurer l'affichage.

En signalant « vsync » comme un signal inversé, GTKWave l'affiche comme un signal classique.



En zoomant sur une ligne, on peut étudier plus de signaux.

À ce niveau de zoom, on identifie le signal de synchronisation horizontale ainsi que l'alternance entre les carrés rouges et les carrés bleus.



En zoomant encore, on arrive au niveau de l'horloge.

On voit alors l'évolution de l'horloge et de la position X.

Il ne s'agit toutefois que d'une simulation !

Elle fait ressortir les erreurs de logiques mais pas les problèmes de propagation du signal.

Mais cela sort du cadre de cette présentation.



**ET MAINTENANT ?**

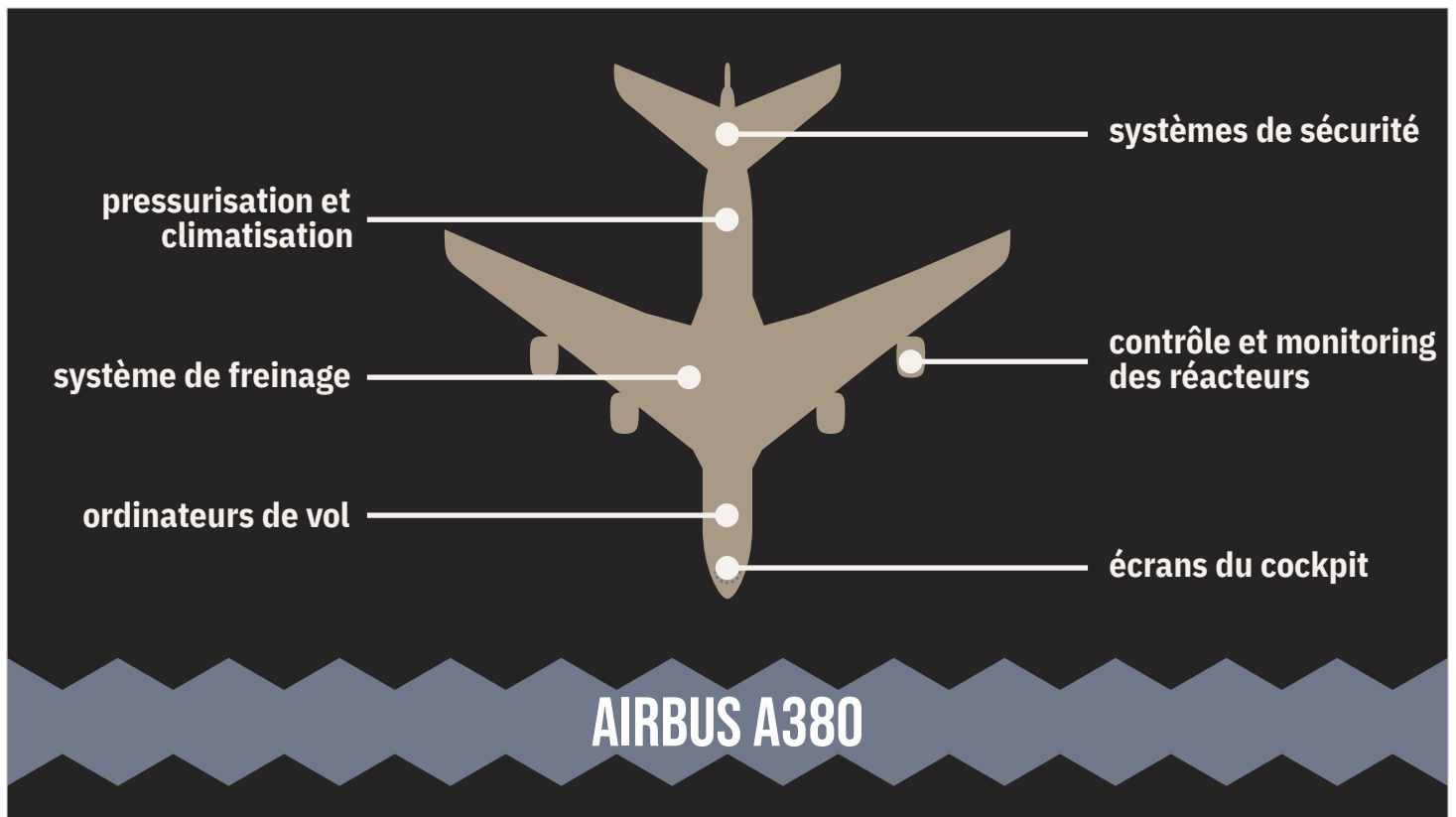
Fort de ces informations, que faire maintenant ?

**85/101**

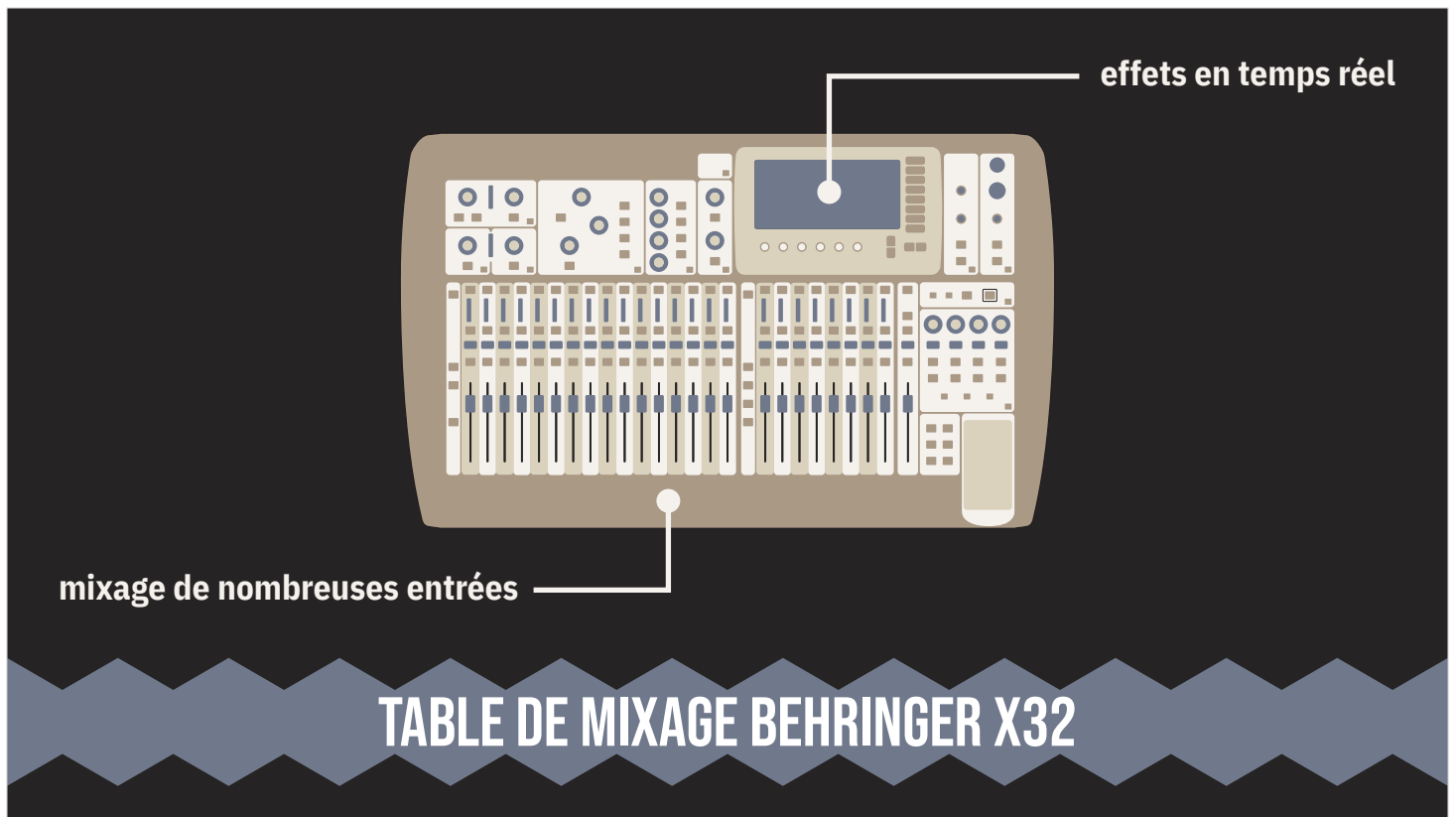


## QUELQUES EXEMPLES D'UTILISATION

Voici quelques exemples d'utilisation de FPGA dans le monde réel.



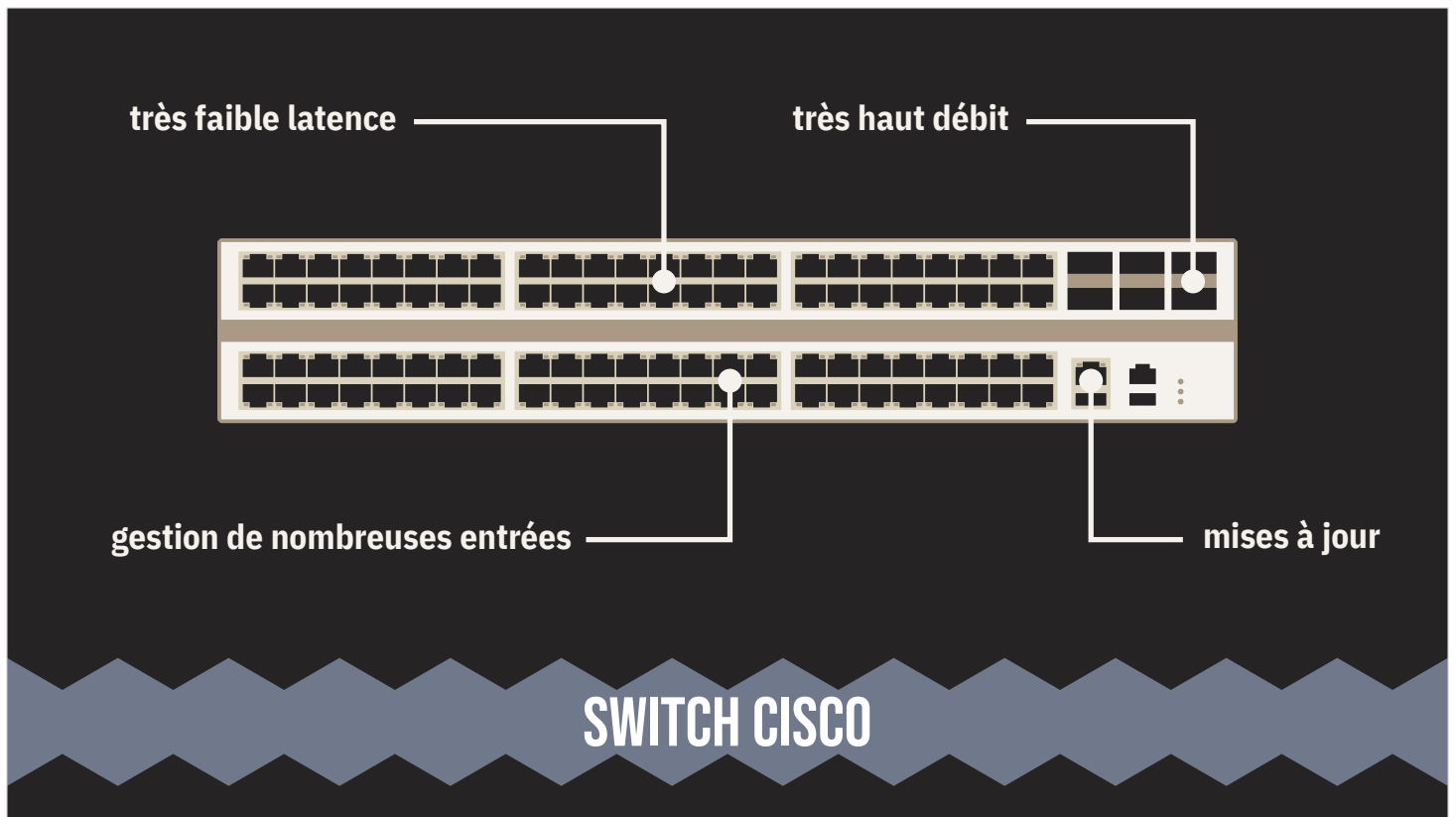
On retrouve de nombreux FPGA dans les Airbus A380 que ce soit pour les ordinateurs de vol, le système de freinage, la pressurisation, les réacteurs etc.



## TABLE DE MIXAGE BEHRINGER X32

On en trouve aussi dans des tables de mixage numérique professionnelle pour gérer un grand nombre d'entrée et d'effets en temps réel.





Il y en a aussi dans les switches Cisco pour obtenir des temps de latence très faibles, un très haut débit tout en gérant de nombreuses entrées. Et la technologie des FPGA permet également de les mettre à jour en cas de faille.

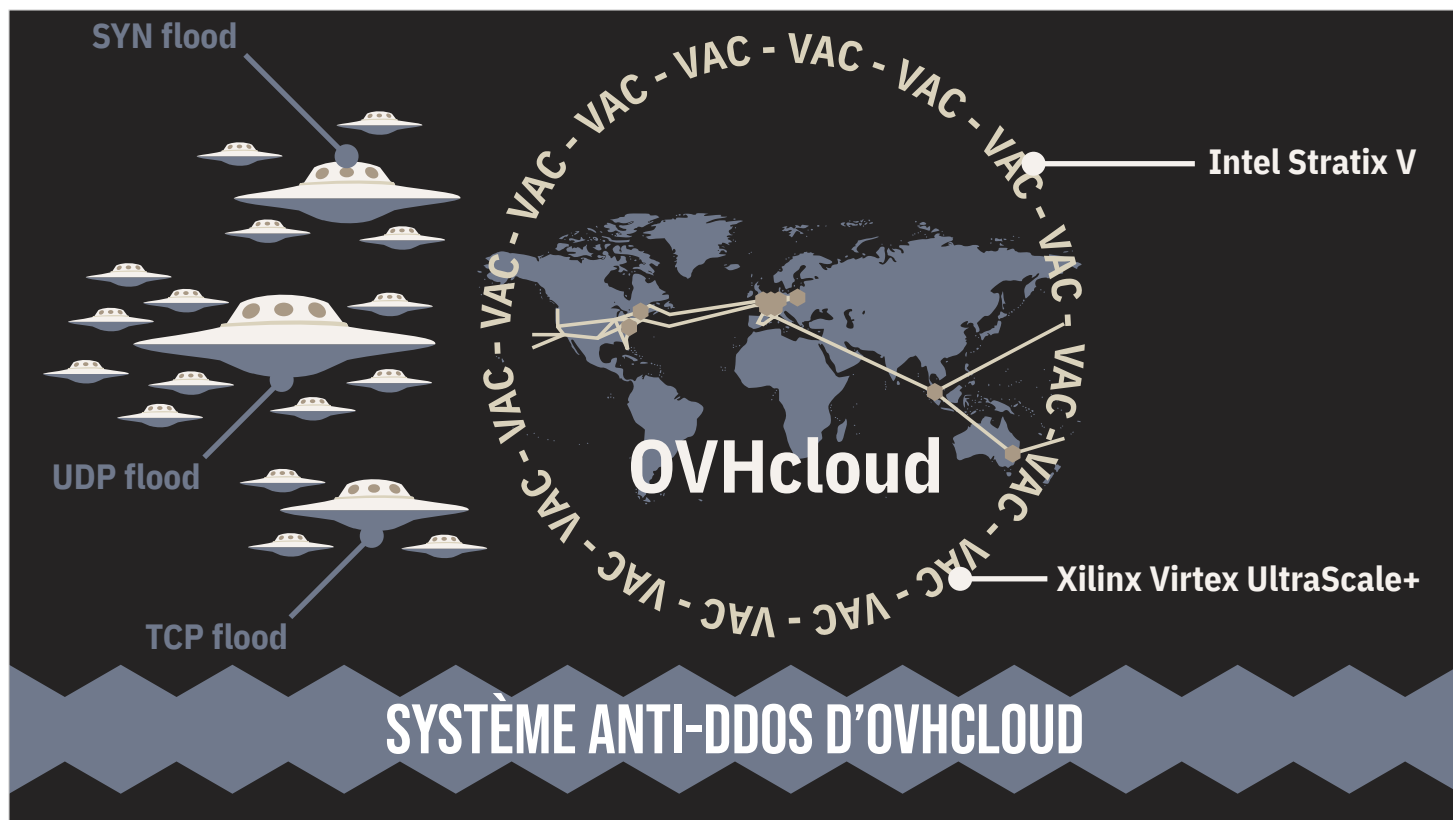
traitement d'image  
après acquisition



traitement des  
données brutes

## IMAGERIE MÉDICALE : IRM

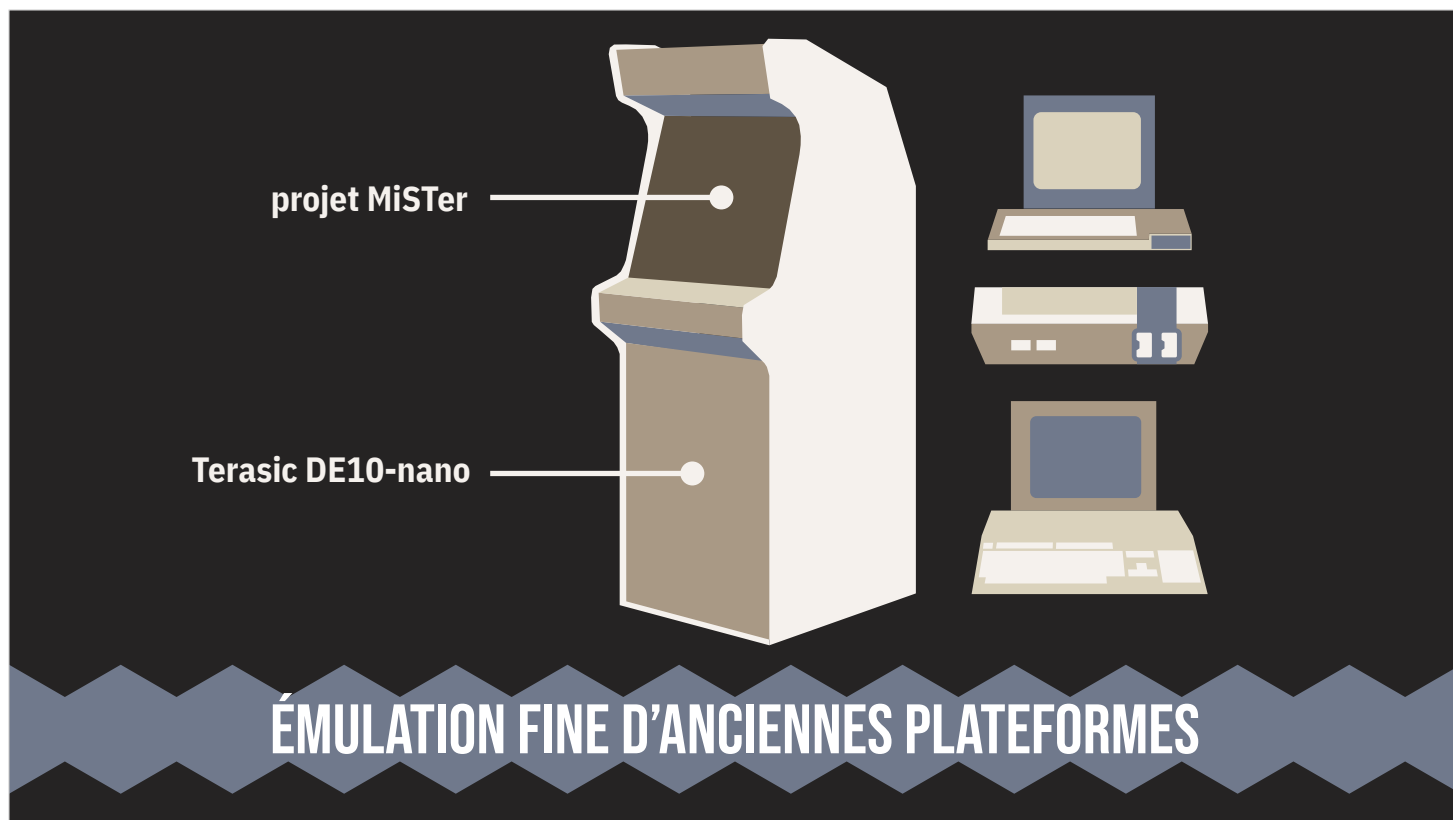
Ils sont aussi souvent utilisés dans l'imagerie médicale que ce soit dans le cadre des IRM, des scanner ou des ultrasons.



OVH a développé son propre système anti-DDOS.

Ils utilisent des FPGA de chez Intel et Xilinx.

Cela leur permet d'encaisser des téraoctets/seconde d'attaque en en mitigeant l'impact.



De façon plus ludique, le projet MiSTer permet d'émuler de façon très fine d'anciens micro-ordinateurs (Atari ST, Amiga, Commodore 64...) ou consoles (NES, SNES...).

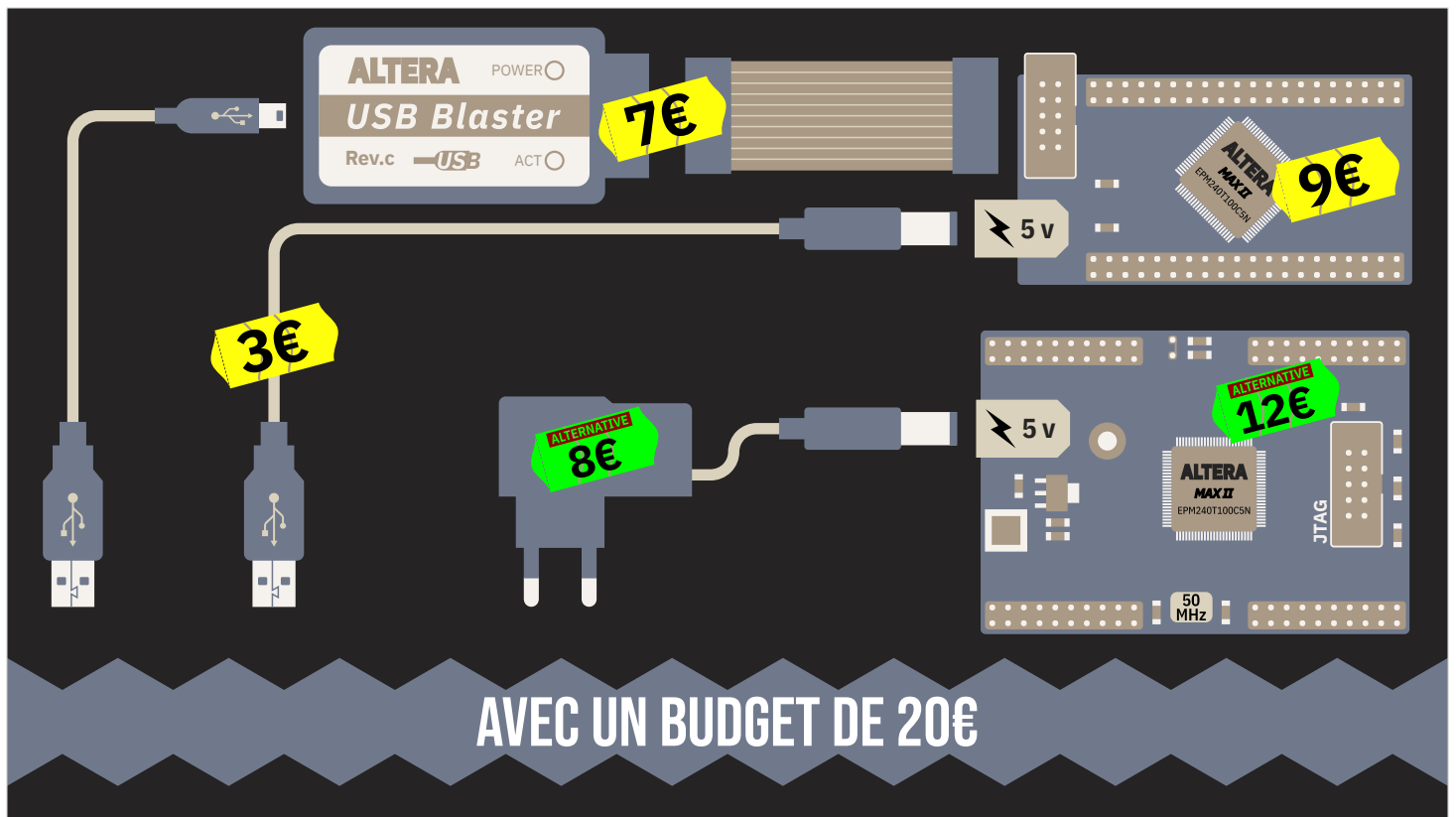
Il utilise le DE10-Nano qui est le grand frère du DE0-Nano. Il est accessible pour environ 130€.

L'émulation permet de coller au plus près du fonctionnement réel des circuits originaux.



## PAR OÙ COMMENCER ?

Si vous êtes curieux·ses mais n'avez pas envie d'investir pour tester cette technologie...



... un budget de 20 € permet une bonne initiation.

Un boîtier USB Blaster pour la programmation du FPGA et une carte Altera Max II EPM240 sont tout ce dont vous avez besoin.

Les prix indiqués sont ceux pratiqués chez Amazon ou AliExpress.

L'environnement de développement Quartus Prime Lite est disponible gratuitement.

On trouve aussi des circuits dans cet ordre de prix chez les concurrents comme Xilinx ou Lattice.

Attention toutefois à ne pas vous faire une idée générale des FPGA à partir de ces cartes : elles sont basiques et non sont programmables qu'une centaine de fois.



## AVANTAGES/INCONVÉNIENTS DES FPGA

Il faut tout d'abord connaître les avantages et les inconvénients des FPGA afin de décider si la technologie peut répondre à votre problématique.

# INCONVÉNIENTS DES FPGA

- Autre façon de programmer
  - paradigme
  - temps de compilation
  - débogage
- Positionnement
  - ASIC
  - GPU
  - microcontrôleur
- Ticket d'entrée
  - coût des logiciels
  - coût des circuits
- Écosystème limité
  - open source peu présent
  - disponibilité des compétences
  - OpenCL

Développer sur FPGA est très différent du développement traditionnel, que ce soit au niveau du paradigme, des temps de compilation ou du débogage.

Son positionnement est loin d'être évident tant les alternatives sont présentes (ASIC, GPU, microcontrôleur...).

Le ticket d'entrée est élevé que ce soit au niveau des logiciels, des circuits mais aussi de la courbe d'apprentissage.

Et même s'il se développe, l'écosystème reste limité. Il n'est par exemple pas possible d'imaginer un workflow complètement open-source sans s'imposer des contraintes énormes.



## AVANTAGES DES FPGA

- Plus proche de l'électron
  - flexibilité des entrées/sorties
  - moins de composants
  - consommation électrique
- Puissance de calcul
- Précision du signal
  - comportement déterministe
  - faible latence
- Reconfigurable
- Sécurité
  - pas de dépassement
  - offuscation ?

Les FPGA ont cependant de sérieux avantages.

Il est difficile de trouver à quoi ils ne pourraient pas se connecter. Il existe des FPGA se connectant directement sur un port mémoire, comme une barrette de RAM, afin d'aller encore plus vite qu'avec un port PCI-Express.

Sa consommation électrique est faible et sa configurabilité réduit le nombre de composants d'un circuit.

Leur architecture donne une puissance de calcul conséquente à faible fréquence d'horloge.

Elle permet de traiter le signal de façon déterministe et avec une latence très faible.

Au niveau sécurité, il n'y a pas de débordement de pile ou de tas et l'ingénierie inverse est difficile.



**Merci de votre attention !**

Merci à

Virginie Férey Rochefeuille, Rémi Passerieu, David Glaude, Thoma Hauc,  
Tristan Groléat (OVHcloud), Francis Trautmann, Sébastien Dupire,  
Loïc “Iooner”, Pascale Lambert-Charreteur,  
Échelle Inconnue et l'équipe du Devfest Nantes

Merci de votre attention !

Une bibliographie vous attend après les diapositives sur les licences et logiciels utilisés.

# LICENCES

- Présentation sous Licence CC-BY 4.0
  - textes, images, gabarits... sauf mention contraire
  - <https://creativecommons.org/licenses/by/4.0/legalcode.fr>
- Les polices suivantes ont été utilisées
  - IBM Plex™, licence OFL
  - Bebas Neue, licence OFL


Cette présentation est proposée sous licence Creative Commons Attribution CC-BY 4.0.

Cela inclut les textes, images, schémas, gabarits, palette de couleurs...









Les polices IBM Plex et Bebas Neue ont été utilisées sous licence OFL.

# LOGICIELS

- Logiciels

- Intel® Quartus® Prime Lite
- ModelSim ASE
- Icarus Verilog 
- GTKWave 

- Présentation

- Inkscape 
- LibreOffice Impress 
- LibreOffice Writer 
- LibreOffice Calc 
- JabRef 
- hilite.me 
- Colors.co 
- TinyVGA 

Lors de l'élaboration des exemples, j'ai utilisé les logiciels Quartus Prime, Icarus Verilog et GTKWave.

Pour réaliser cette présentation, j'ai utilisé les logiciels Inkscape LibreOffice, JabRef, hilite.me, colors.co et TinyVGA.

# BIBLIOGRAPHIE

Pour réaliser cette présentation sur les FPGA, j'ai compulsé quelques documents dont voici la liste...

**101/101**