

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université de Carthage

Ecole Nationale d'Ingénieurs de Carthage



المدرسة الوطنية للمهندسين بقرطاج

Ecole Nationale d'Ingénieurs de Carthage

وزارة التعليم العالي و البحث العلمي

جامعة قرطاج

المدرسة الوطنية للمهندسين بقرطاج

Systèmes embarqués

Niveau: 2 ING INFO

Année Universitaire
2020/2021



Plan du module



1

Introduction aux systèmes embarqués

2

Cible logicielle: Les microcontrôleurs STM32

3

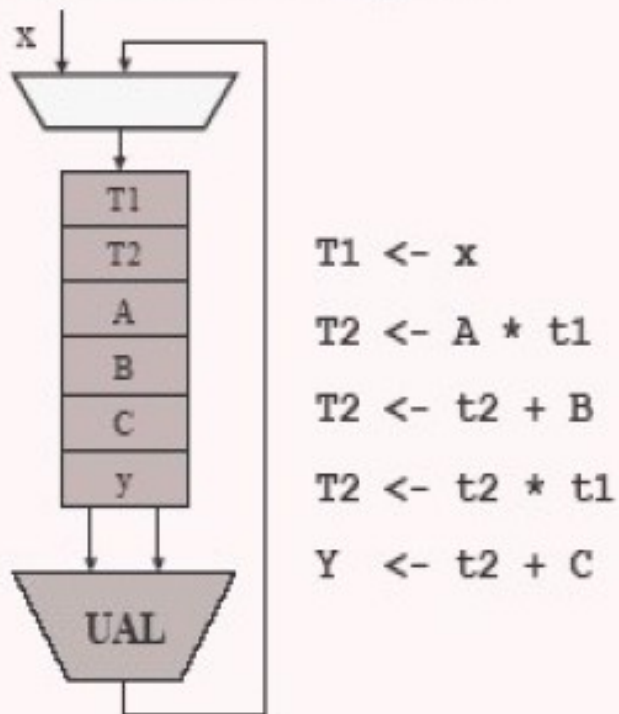
Cible matérielle: Langage VHDL et carte FPGA

Exemple introductif:

$$A.x^2 + B.x + C = (A.x + B).x + C$$

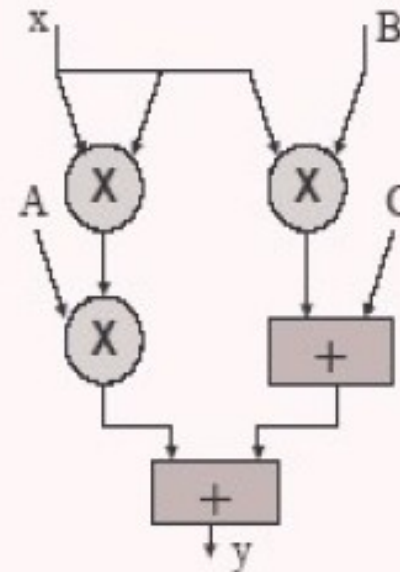
Solution logicielle

Implémentation Temporelle

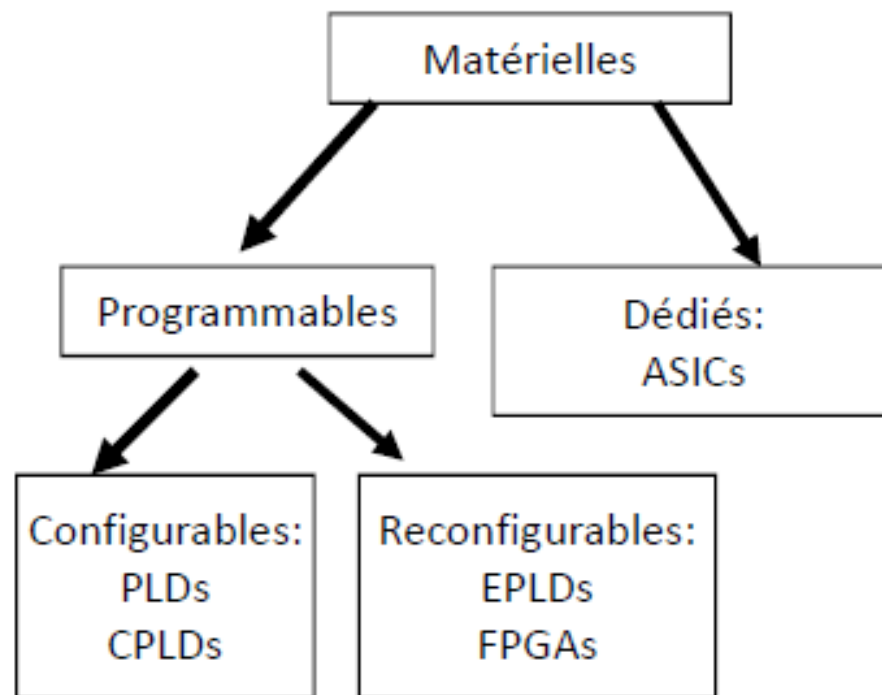


Solution matérielle

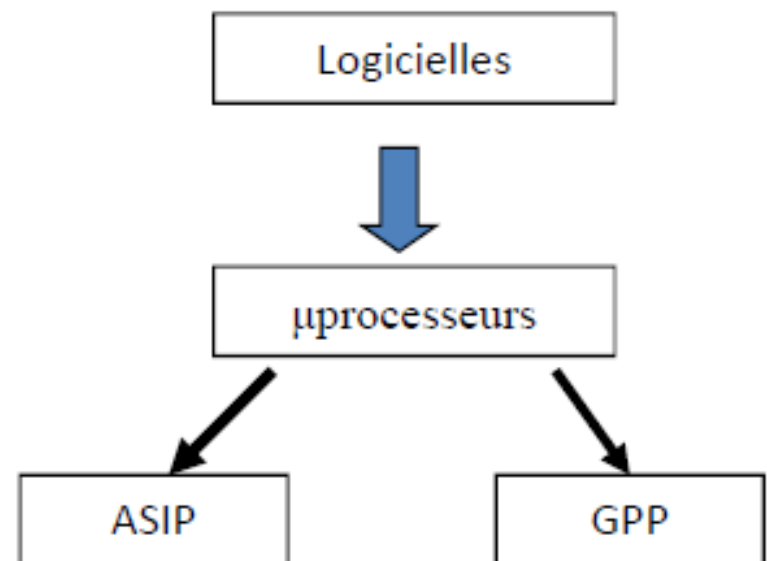
Implémentation Spatiale



Traitement parallèle



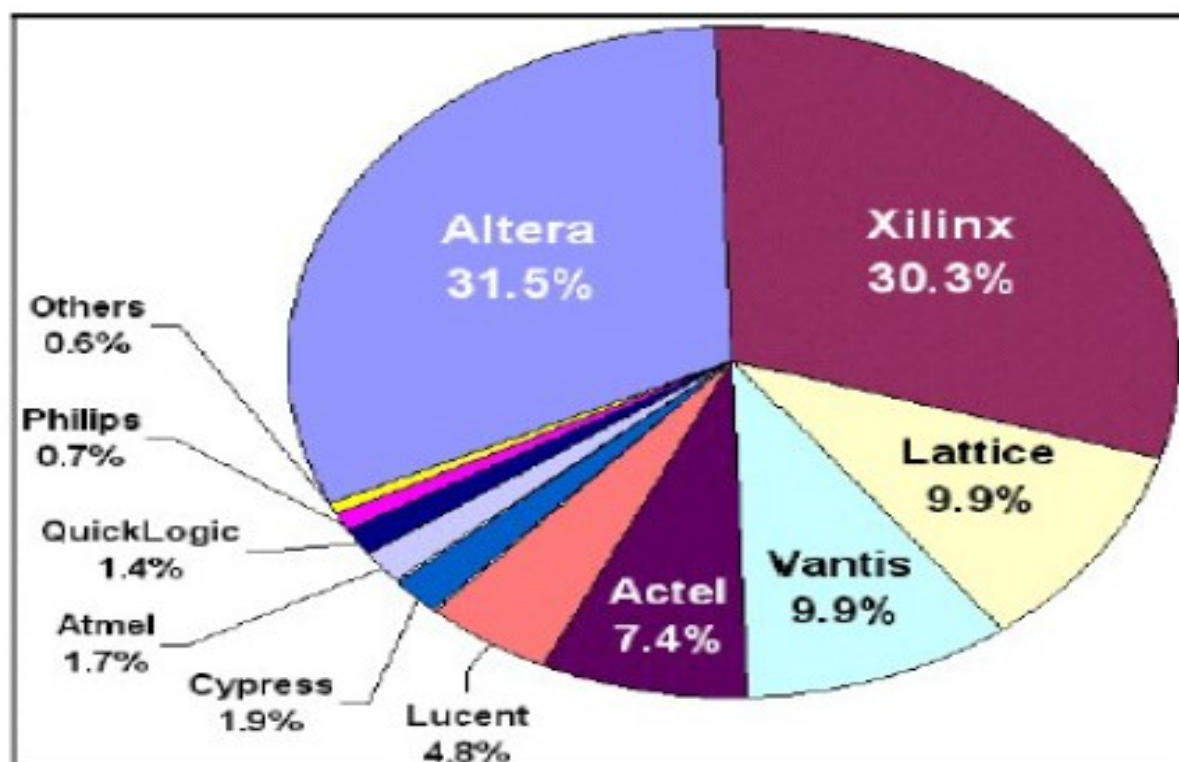
Traitement séquentiel



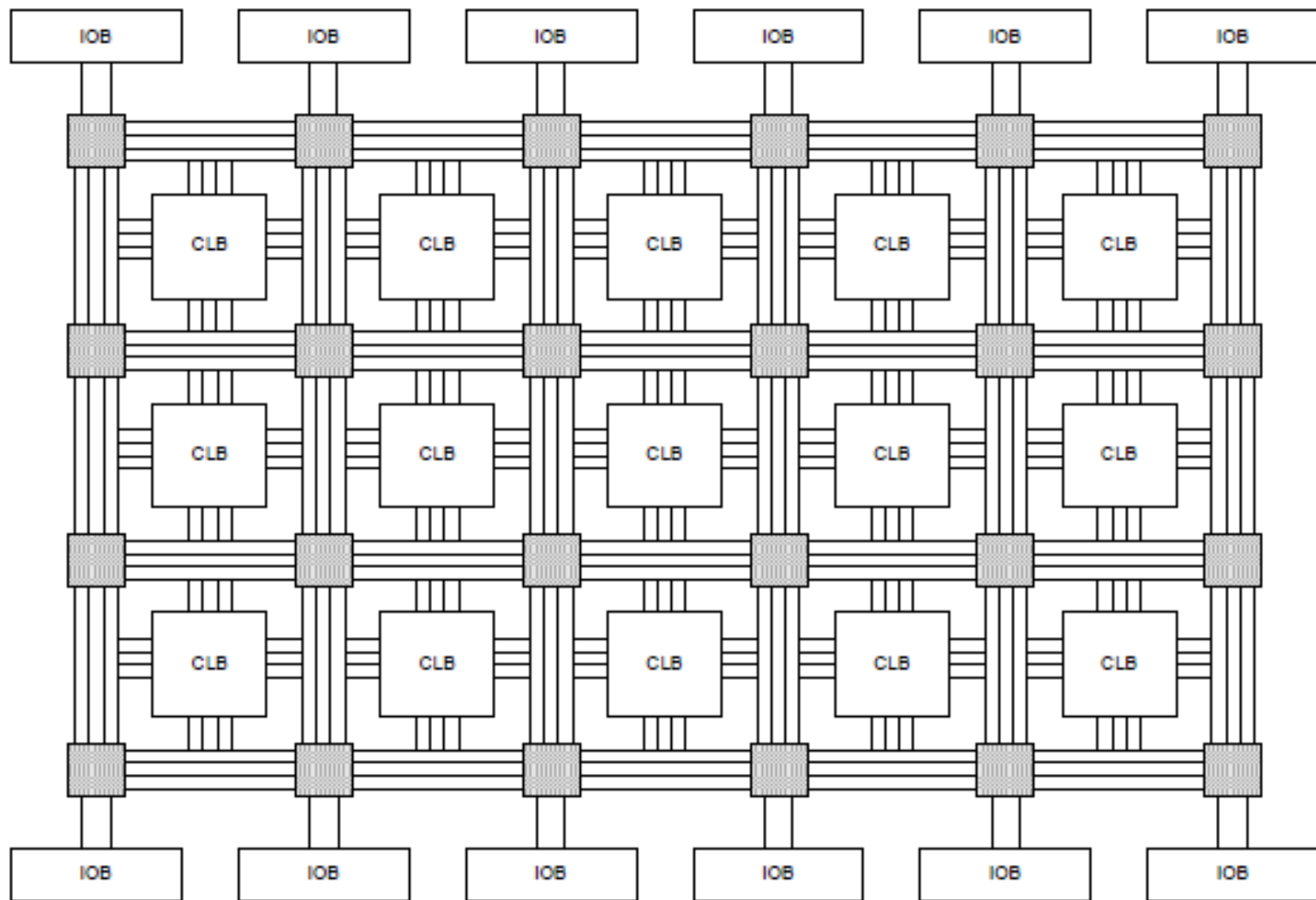
Introduction aux cartes FPGA

- Les **FPGA** (Field Programmable Gate Array) se sont imposés sur le marché des circuits logiques depuis la fin des années 1980 pour remplacer quelques circuits programmables de façon plus efficace
- Ils concurrencent de nos jours certains microprocesseurs et microcontrôleurs en complexité et en performance.
- Un FPGA est composé essentiellement de :
 - ✓ un réseau de blocs de logique programmable (Configurable Logic Block **CLB**), chaque bloc pouvant réaliser des fonctions complexes de plusieurs variables, et comportant des éléments à mémoire;
- un réseau d'interconnexions reconfigurables (**PSM**: Programmable Switch Matrix)
- ✓ des blocs spéciaux d'entrée et de sortie avec le monde extérieur (Input/Output Block – **IOB**).

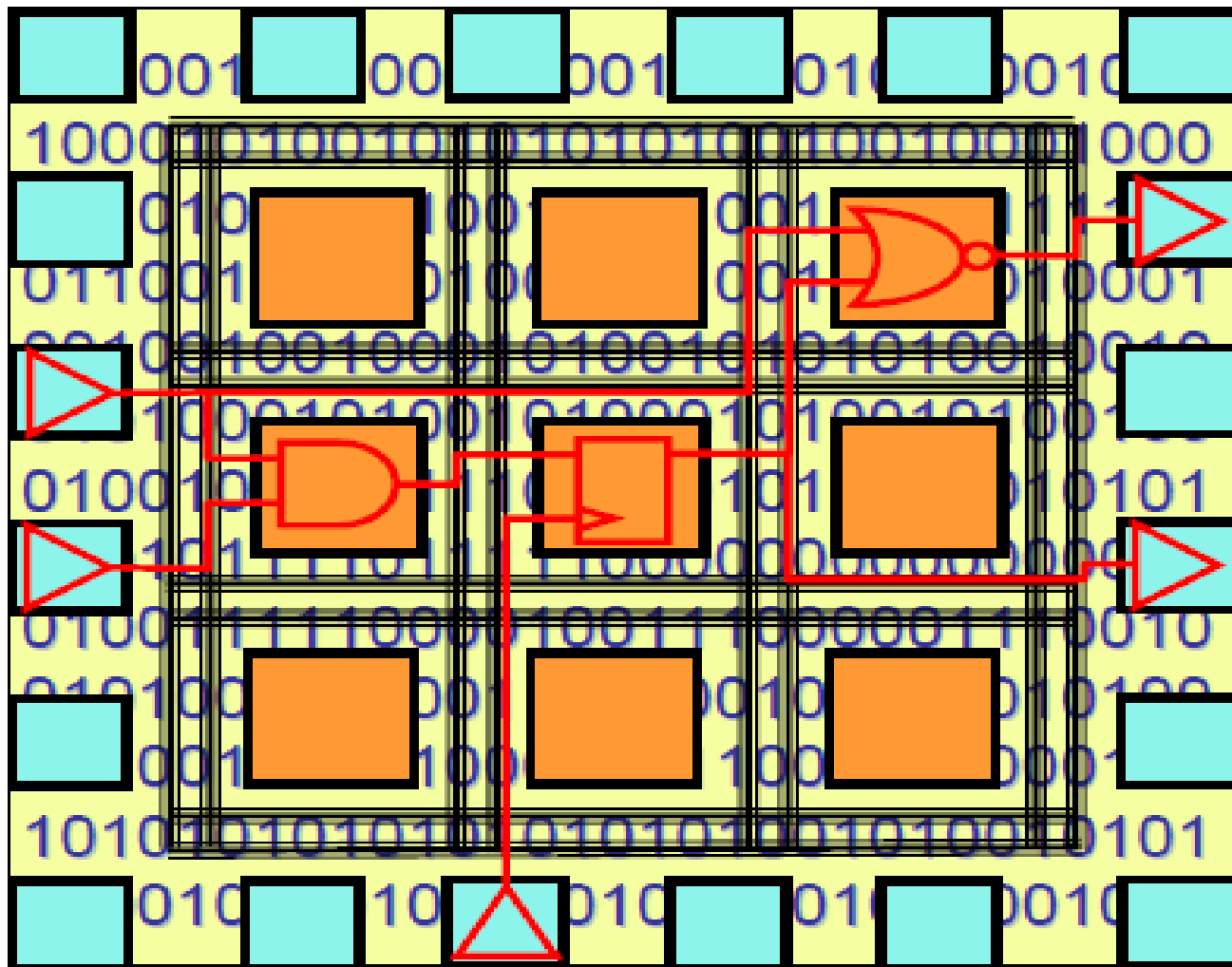
Parts du marché des fabricants du FPGA



Sources : <http://www.optimagic.com/market.html>



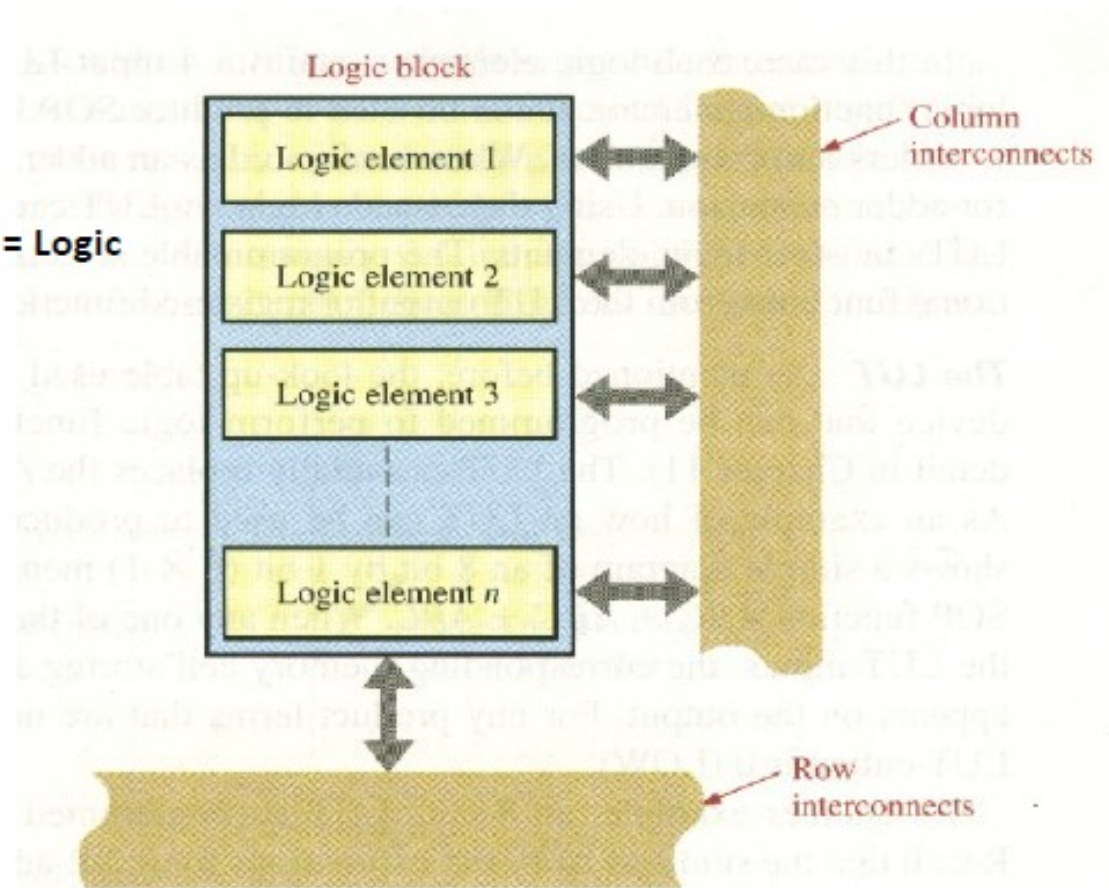
Dans quel objectif?



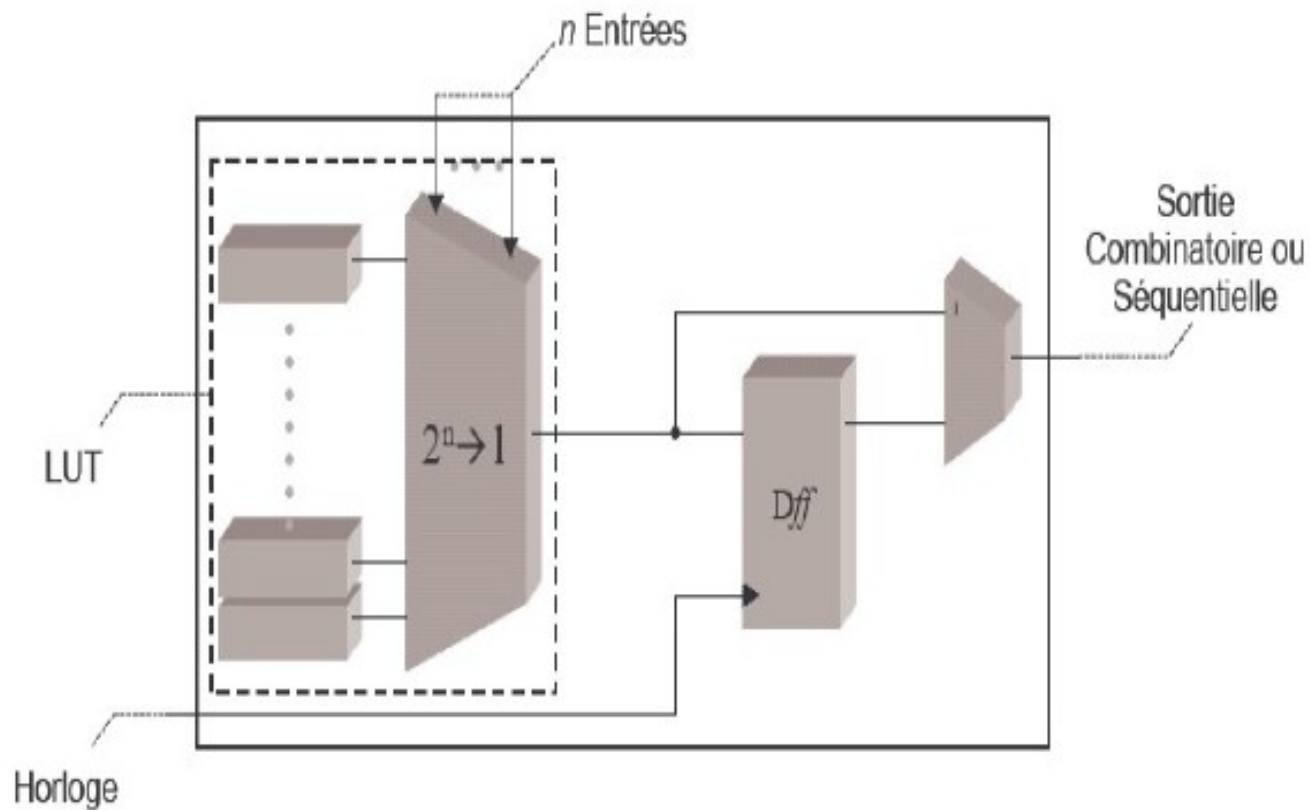
FPGA OPERATION

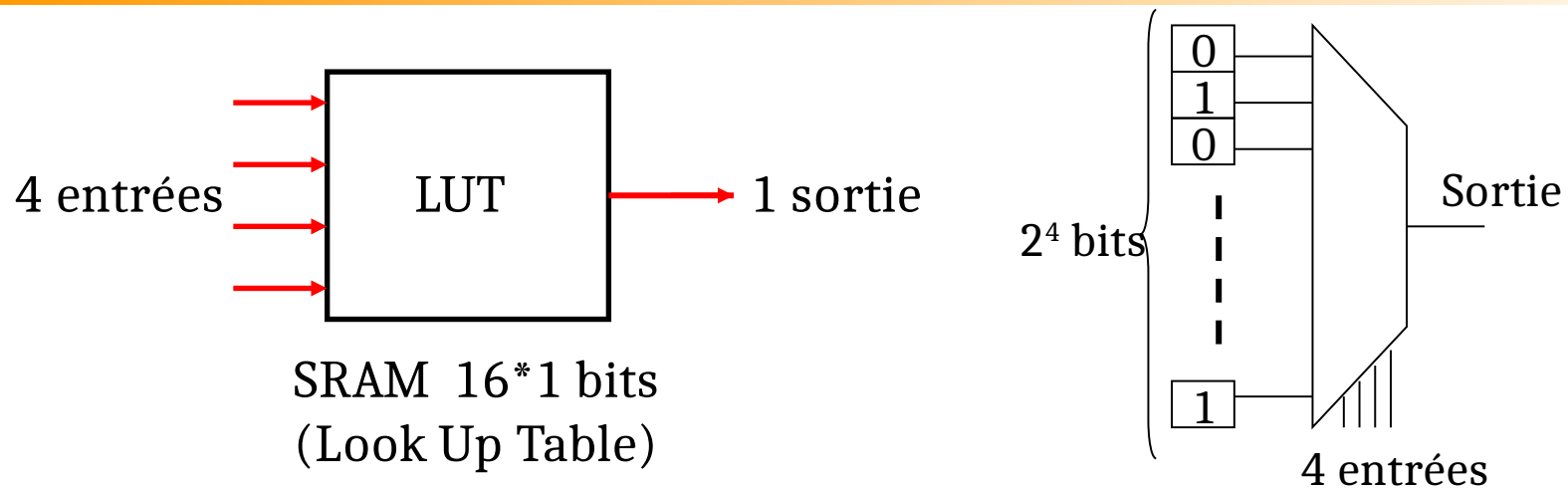
Bloc Logique configurable (CLB)

Logic Element = Logic cell

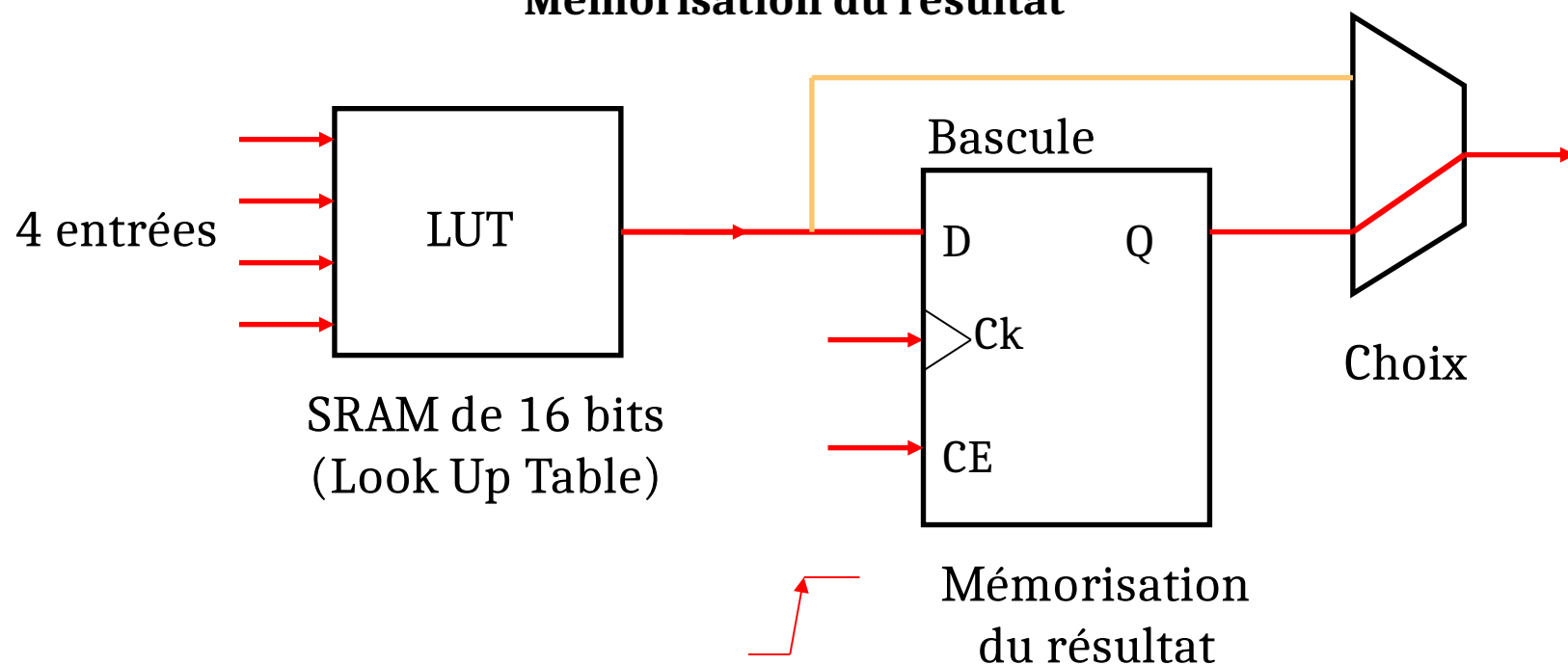


Élément Logique (Logic Element)

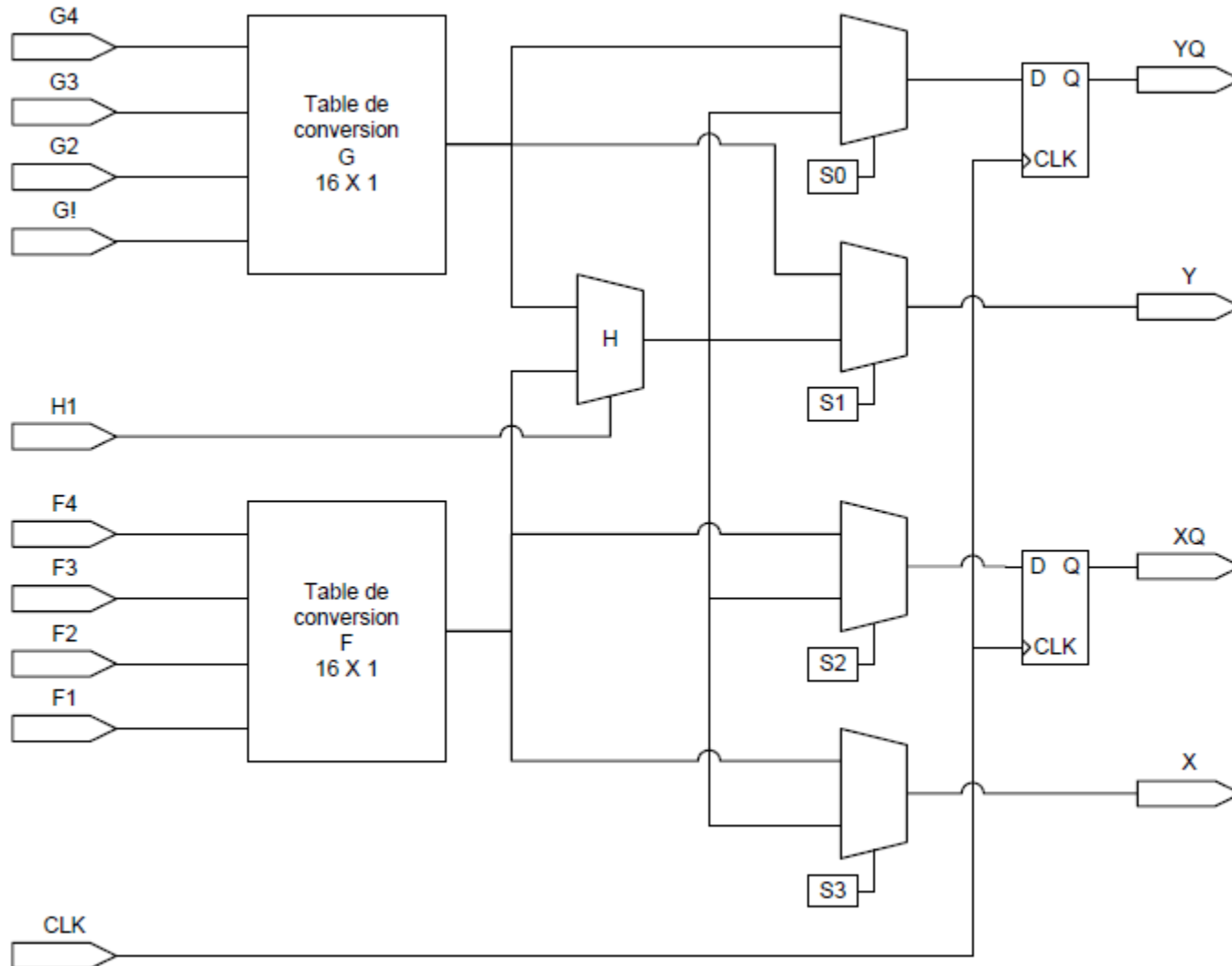




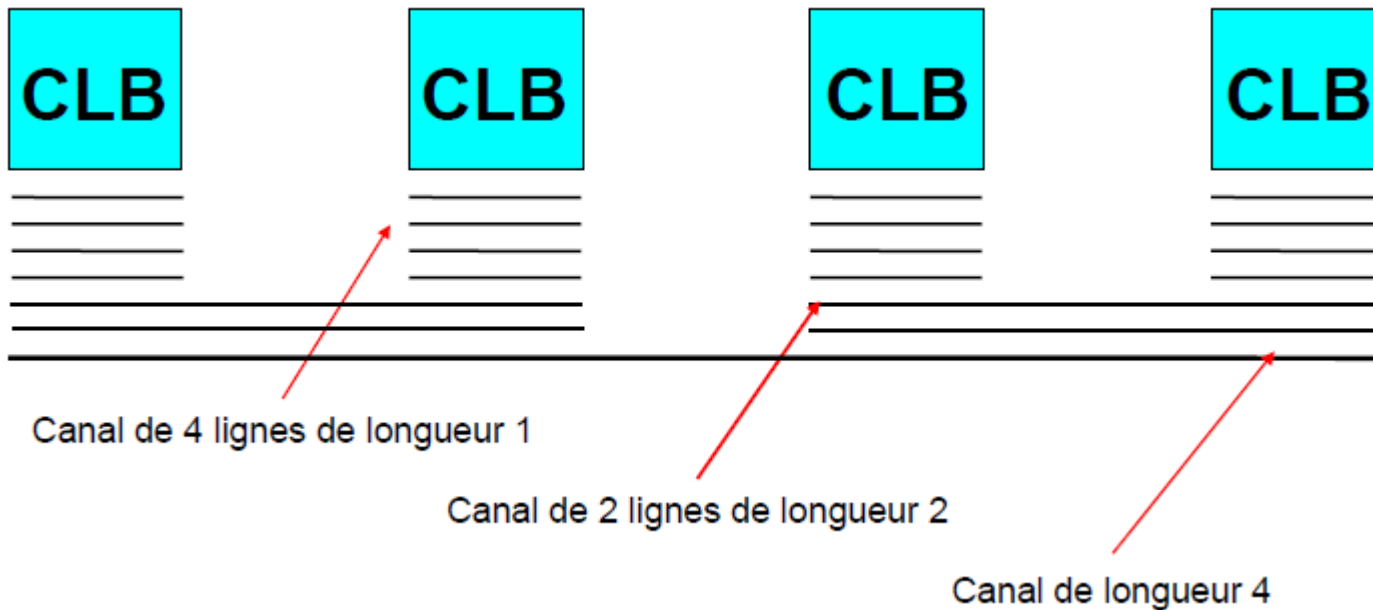
Mémorisation du résultat



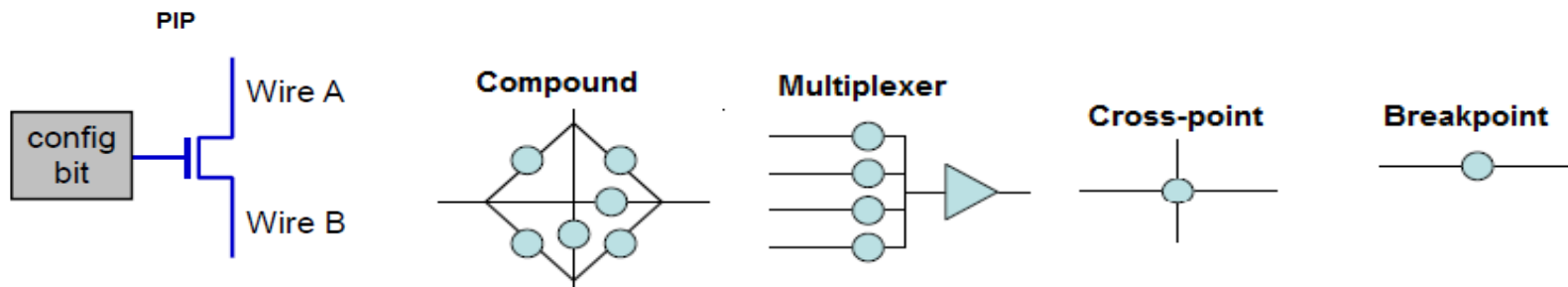
Exemple de CLB (XILINX)



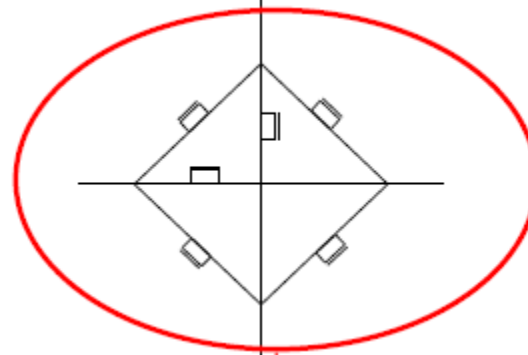
Interconnexions dans les FPGA



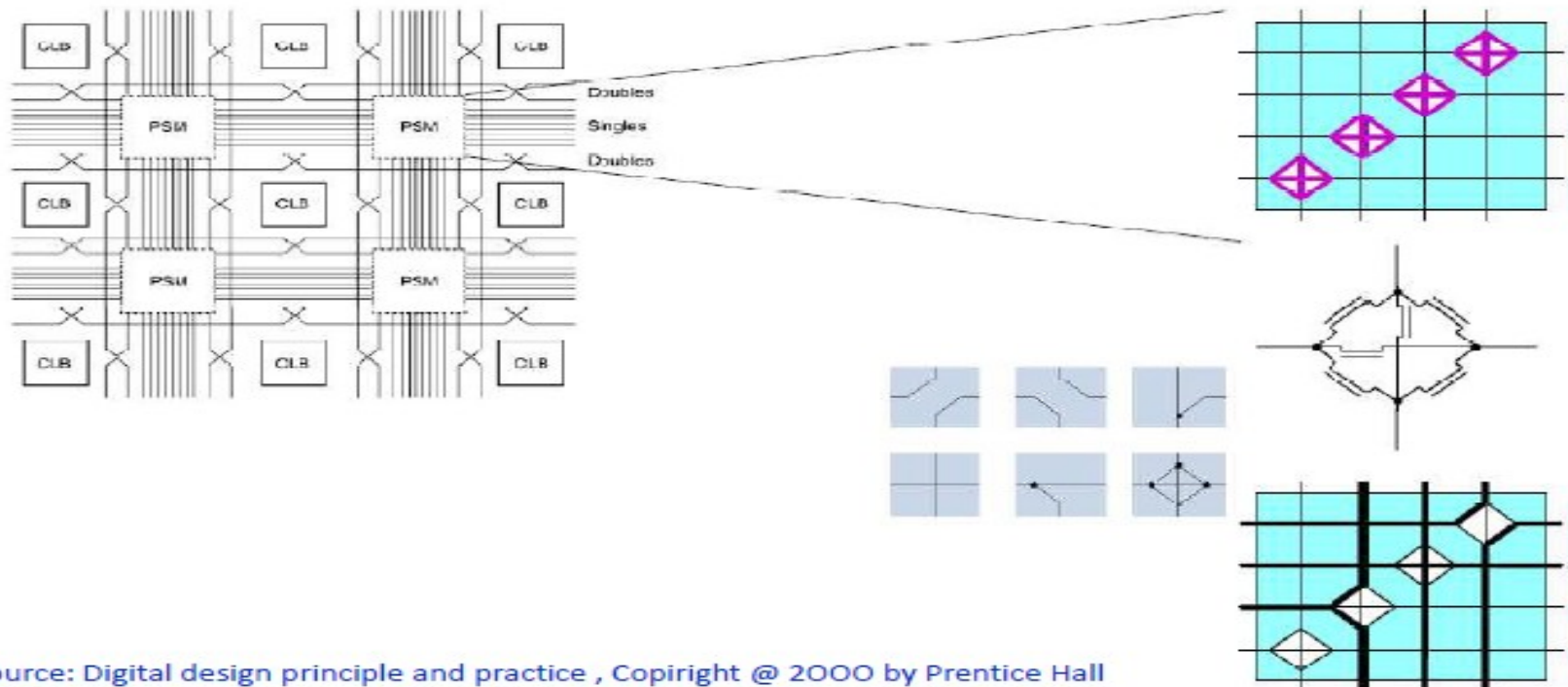
Un canal faisant toute la largeur sert pour les signaux globaux (horloge, reset...)

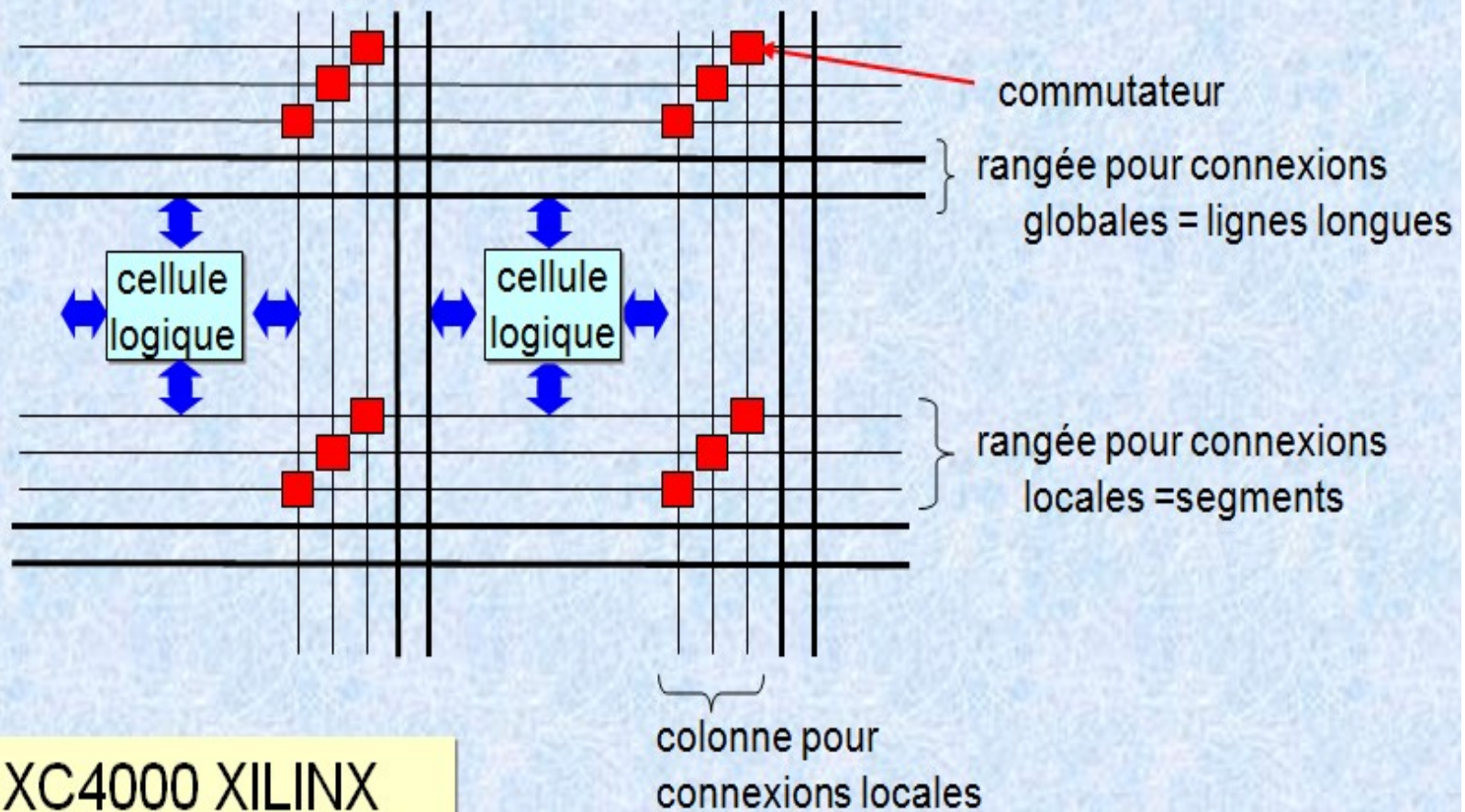


PIP: Programmable Interconnect Point



La matrice d'interconnexion se constitue d'un ensemble de transistors qui permettent la liaison entre les lignes.

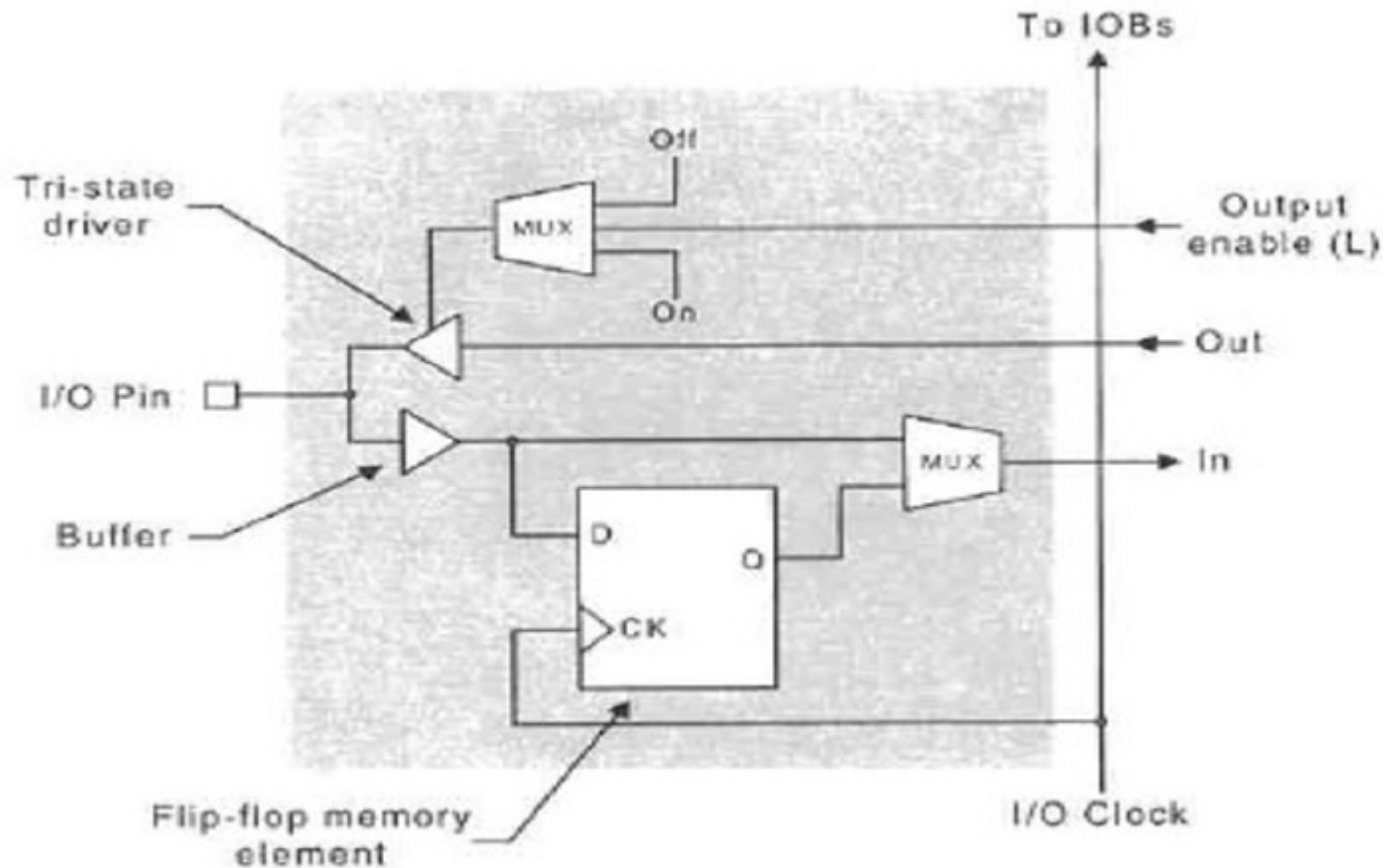




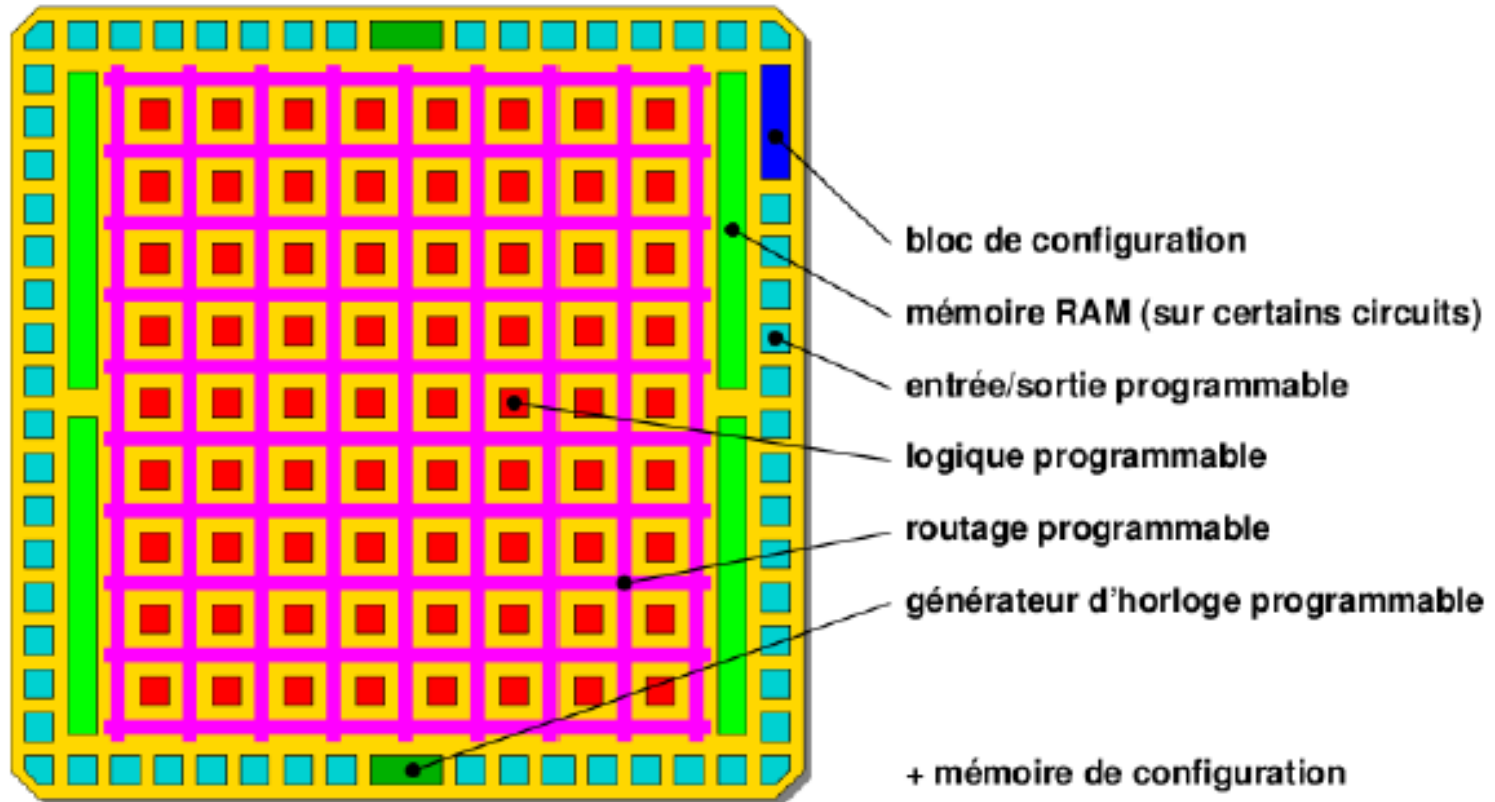
exemple : XC4000 XILINX

IOB

- ✓ Contrôle une broche du composant
- ✓ Peut être programmé: en entrée, en sortie, en E/S (bidirectionnelle)



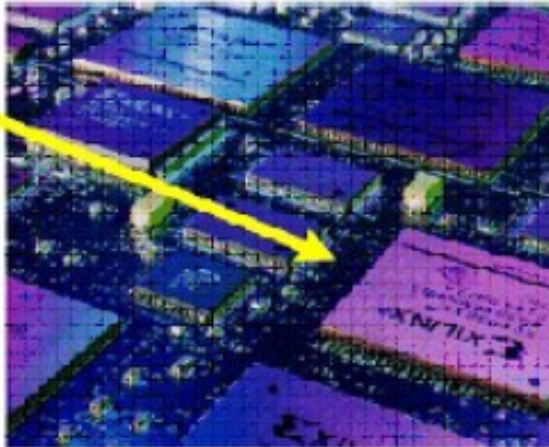
AUTRES ÉLÉMENTS (EXEMPLE DE LA STRUCTURE INTERNE FPGA- XILINX)



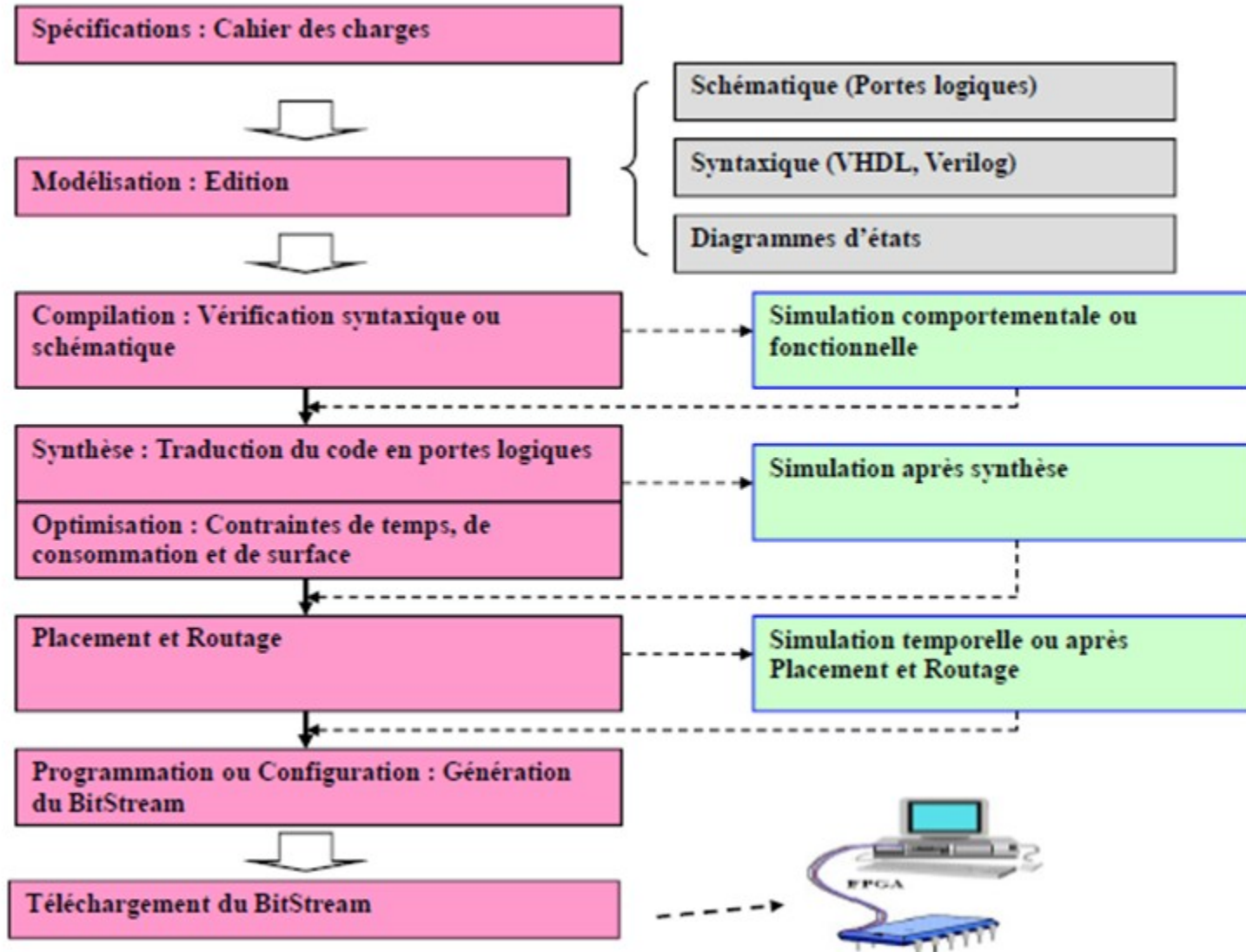
LE BITSTREAM

- ❑ Le bitstream décrit la configuration de tous les éléments configurables du circuit.
- ❑ Un transfert de bitstream est nécessaire lors de la mise sous tension et à chaque reconfiguration

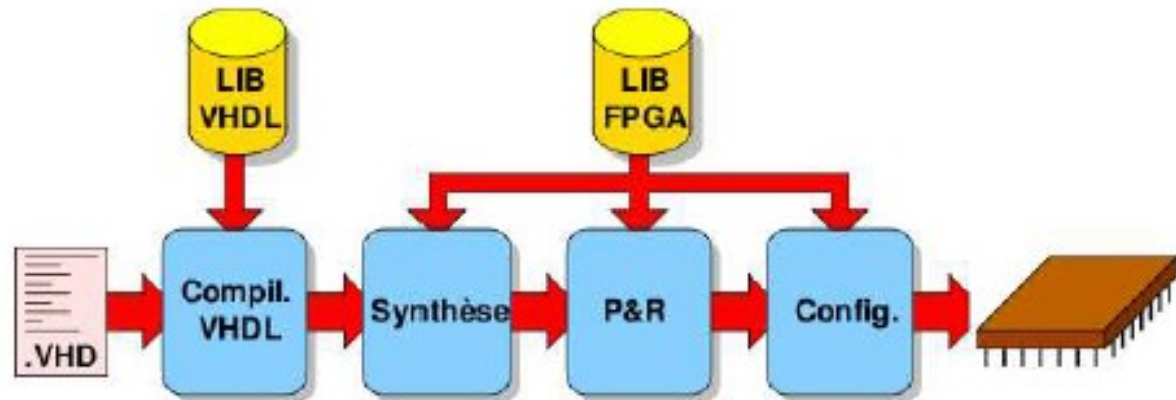
10010010011110010110



Flot de conception d'un design sur FPGA



Développement sur les FPGA: VHDL



Déclaration des ressources

- ✓ Permet d'inclure les librairies de types prédéfinis ou fonctions
- ✓ On retrouve en entête du fichier source *.vhd les instructions suivantes:

Library ieee;

Use ieee.std_logic_1164.all

Le mot clé: **library**
rend les packages dans
IEEE visibles

Le mot clé: **all** rend
tous les éléments du
package sont disponibles

Le mot clé: **use** identifie ce qui pourrait être
utilisé de la bibliothèque IEEE

ENTITE

- Décrit l'interface sans décrire le comportement interne du modèle
- C'est une description des ports d'entrée/sortie avec sa direction (in, out, inout) et son type
- **Exemple 1: porte logique deux entrées, une sortie:**

```
entity GATE is  
  port (  
    A, B: in bit  
    S: out bit);  
end GATE;
```

- **Exemple 2 : écrire l'entité d'un demi-additionneur**



Architecture

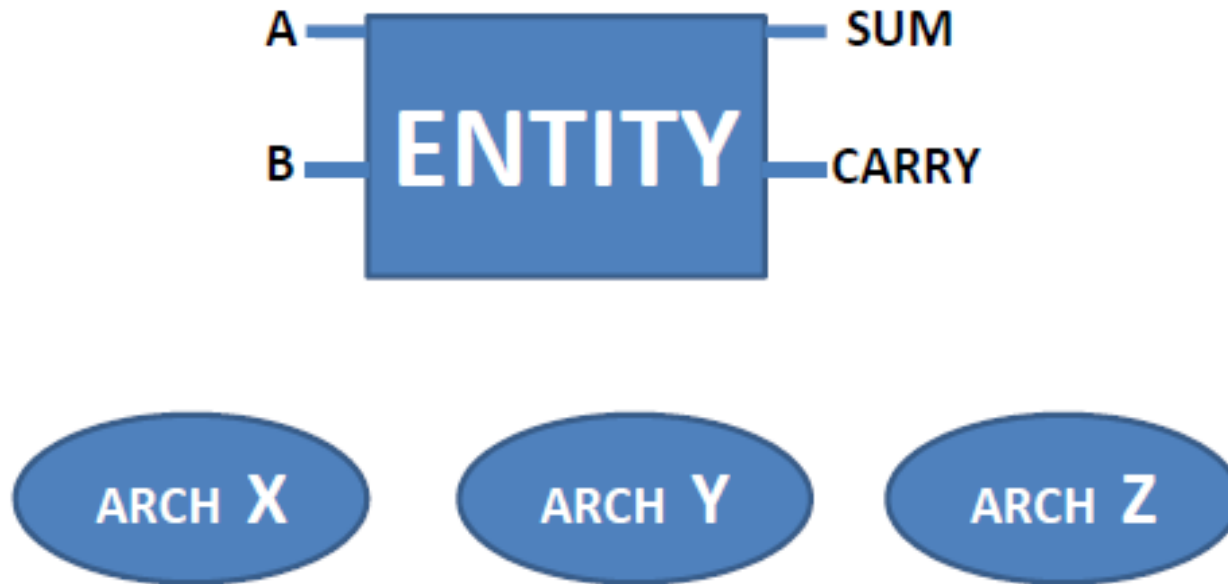
- Reliée à une entité. Description du fonctionnement du système: comportementale + structurelle
- Le mot clé **architecture** définit l'architecture interne de l'entité à laquelle elle est associée

Exemples:

```
Architecture ORGATE of GATE is  
begin  
    S <= A or B;  
end ORGATE;
```

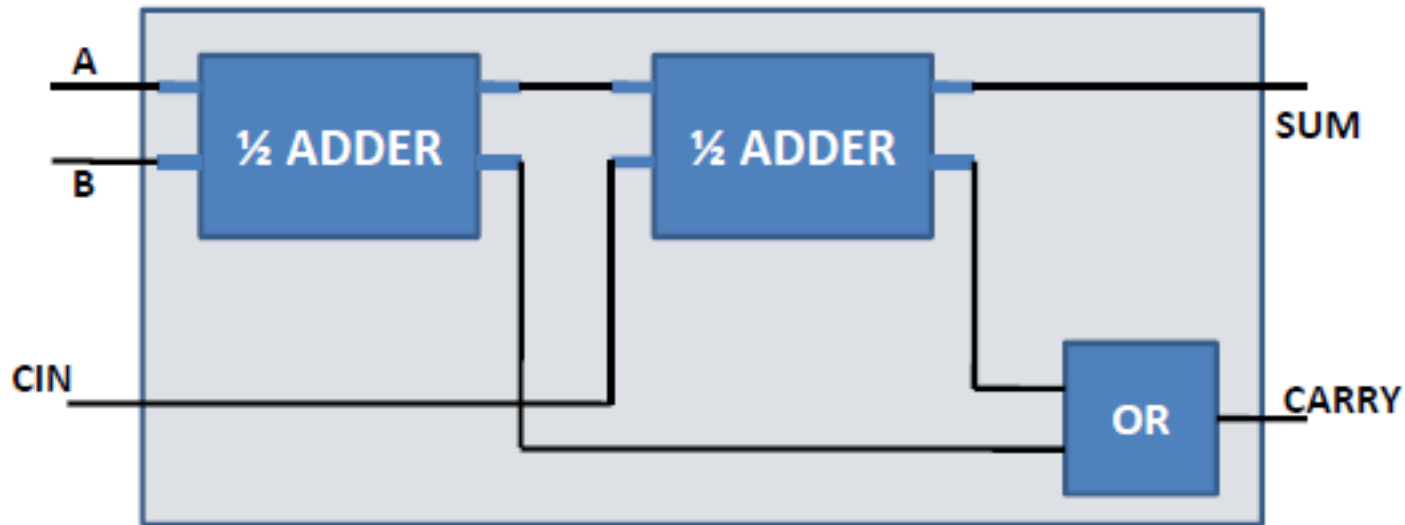
```
Architecture BEHAVE of HALFADD is  
begin  
    SUM    <= A xor B;  
    CARRY  <= A and B;  
end BEHAVE;
```

- Toute architecture doit être associée à une entité
- Le nom d'un fichier VHDL doit être celui de l'entité qu'il contient
- Après le begin de l'architecture les signaux (ou ports) sont affectés.
- Une entité peut avoir plusieurs architectures.



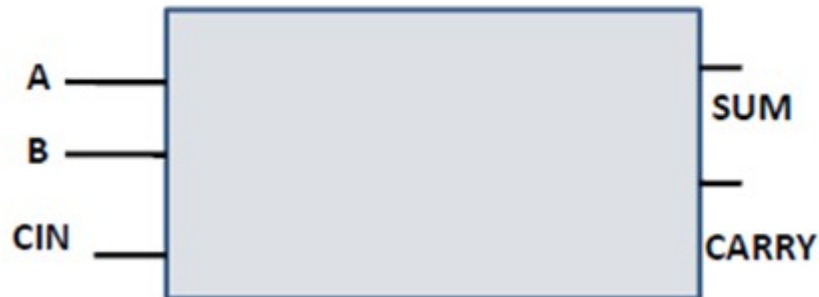
Hiérarchie

- L'entité et l'architecture sont les principaux blocs autour desquels un modèle hiérarchique est construit.
- **L'exemple** d'un additionneur complet, décrit par deux demi additionneurs et une porte OR, illustre ce principe.



On commence par décrire l'entité de l'additionneur complet

```
entity FULLADD is  
port (  
    A,    B, CIN : in bit ;  
    SUM,  CARRY : out bit);  
end FULLADD;
```



Nous disposons déjà des entités et architectures des éléments internes de l'additionneur complet

```
----- HALFADD -----  
entity HALFADD is  
port (  
    A,      B      : in bit ;  
    SUM,    CARRY  : out bit);  
end HALFADD;  
  
Architecture BEHAVE of HALFADD is  
begin  
    SUM    <= A xor B ;  
    CARRY  <= A and B ;  
end BEHAVE ,  
  
----- ORGATE -----  
entity ORGATE is  
port (  
    A,      B      : in bit ;  
    Z       : out bit);  
end ORGATE ;  
  
architecture arch of ORGATE is  
begin  
    Z <= A or B ;  
end arch;
```

Implémentation de l'architecture de l'additionneur complet:

▫ L'avant begin:

```
architecture struct of FULLADD is
    signal I1, I2, I3: bit;
    component HALFADD
    port (
        A,      B      : in bit ;
        SUM,    CARRY   : out bit);
    end component;

    component ORGATE
    port (
        A,      B      : in bit ;
        Z       : out bit);
    end component;

begin
```

- ✓ Les signaux locaux sont toujours déclarés à l'intérieur de l'architecture
- ✓ Avant de faire une instance d'une entité, les composants à utiliser doivent être déclarés.

```
architecture struct of FULLADD is
    signal I1, I2, I3: bit;
```

→ Déclaration des composants ici

```
begin
    U1: HALFADD port map (A, B, I1, I2);
    U2: HALFADD port map (I1, CIN, SUM, I3);
    U3: ORGATE  port map (I3, I2, CARRY);
end struct;
```

- U1, U2 sont deux instances du modèle HALFADD et U3 est une instance de la porte OR
- I1, I2 et I3 sont trois signaux intermédiaires utilisés pour l'interconnexion
- l'interconnexion est assurée par le mot clé **port map**

Le code complet de l'exemple

```
----- HALFADD -----
entity HALFADD is
port (
    A,    B      : in bit ;
    SUM,   CARRY  : out bit);
end HALFADD;

Architecture BEHAVE of HALFADD is
begin
    SUM    <= A xor B;
    CARRY  <= A and B;
end BEHAVE;

----- ORGATE -----
entity ORGATE is
port (
    A,    B      : in bit ;
    Z      : out bit);
end ORGATE;

architecture arch of ORGATE is
begin
    Z <= A or B;
end arch;
```

```
----- FULLADD -----
entity FULLADD is
port (
    A,    B, CIN : in bit ;
    SUM,   CARRY : out bit);
end FULLADD;

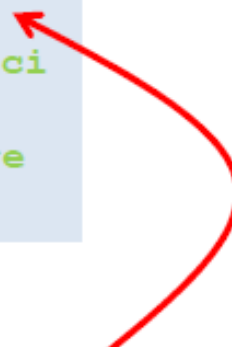
architecture struct of FULLADD is
    signal I1, I2, I3: bit;
    component HALFADD
    port (
        A,    B      : in bit ;
        SUM,   CARRY  : out bit);
    end component;

    component ORGATE
    port (
        A,    B      : in bit ;
        Z      : out bit);
    end component;

begin
    U1: HALFADD port map (A, B, I1,I2);
    U2: HALFADD port map (I1, CIN, SUM, I3);
    U3: ORGATE port map (I3, I2, CARRY);
end struct;
```

Types de données

```
architecture STRUCT of FULLADD
is
    signal    N_SUMM: bit;
    -- autres déclarations ici
Begin
    -- code de l'architecture
end STRUCT;
```



➤ Les valeurs possibles pour le type bit sont {0, 1}

- Le langage VHDL exige que chaque signal doit avoir un type de données lorsqu'il est déclaré.
- Le type définit un ensemble de valeurs.
- Une affectation à un signal ne peut avoir qu'une valeur définie par cet ensemble.

FALSE

TRUE

Boolean

'0'

'1'

bit

10 ns

200 fs

2.5 ps

Time

"0000"

"111"

"010101010"

bit_vector

'a' '1'

'A'

'Z' '0'

character

256 -45

0

1 137

integer

"Hello"

"VHDL"

"Esperanto"

string

1.02

1.0

-37.4

real

- Une donnée de type `std_logic` possède une valeur parmi neuf possibles:

'U' uninitialized

'X' forcing unknown

'0' forcing 0

'1' forcing 1

'Z' high impedance

'W' weak unknown

'L' weak 0 (pull-down)

'H' weak 1 (pull-up)

'-' don't care

- Pour une affectation, les valeurs utilisées sont: **'X', '0', '1', 'Z'**

- Un vecteur est un rassemblement d'objets de même type.
- Le langage définit deux types standards de vecteurs qui sont les vecteurs de bits et les strings.

Objets multiples de même type



'1'	'0'	'1'	'0'
-----	-----	-----	-----

- L'intervalle d'un vecteur est défini en déclaration.
- En VHDL l'intervalle est défini en utilisant les opérateurs « **to** » ou « **downto** ».

Déclaration correctes

```
signal Z_BUS:bit_vector(3 downto 0);  
signal C_BUS:bit_vector(1 to 4);
```

Déclaration incorrectes !!!

```
signal Z_BUS:bit_vector(0 downto 3);  
signal C_BUS:bit_vector(3 to 0);
```

```
signal Z_BUS:bit_vector(3 downto 0);  
signal C_BUS:bit_vector(1 to 4);
```

```
Z_BUS <= C_BUS;
```

Est la même que:

```
Z_BUS(3) <= C_BUS(1);  
Z_BUS(2) <= C_BUS(2);  
Z_BUS(1) <= C_BUS(3);  
Z_BUS(0) <= C_BUS(4);
```

```
signal Z_BUS:bit_vector(3 downto 0);  
signal C_BUS:bit_vector(1 to 4);
```

Legal:

```
Z_BUS(3 downto 2) <= "00";  
C_BUS(2 to 4) <= Z_BUS(3 downto 1);
```

Illegal:

```
Z_BUS(0 to 1) <= "11";
```

```
signal Z_BUS : bit_vector(3 downto 0);  
signal A, B, C, D : bit;  
signal BYTE : bit_vector(7 downto 0);  
signal A_BUS : bit_vector(3 downto 0);  
signal B_BUS : bit_vector(3 downto 0);
```

```
Z_BUS <= A & B & C & D;
```



concatenation
operator



```
BYTE <= A_BUS & B_BUS;
```

```
signal Z_BUS : bit_vector(3 downto 0);  
signal A, B, C, D : bit;  
signal BYTE : bit_vector(7 downto 0);
```

```
Z_BUS <= (A, B, C, D);
```

agrégation



Équivaut à

```
Z_BUS(3) <= A;  
Z_BUS(2) <= B;  
Z_BUS(1) <= C;  
Z_BUS(0) <= D;
```

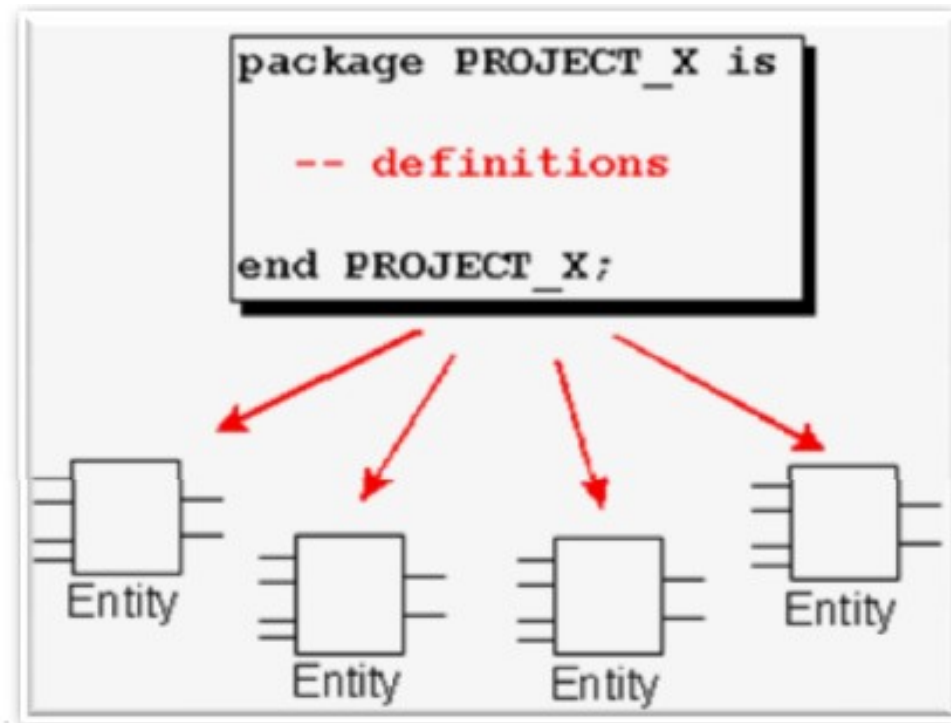
Définition des types utilisateurs

- Les types utilisateurs sont des types énumérés
- La définition se fait dans des parenthèses dont les éléments sont séparés par virgule.
- **Exemple:**

```
type MY_STATE is
    (RESET, IDLE, RW_CYCLE, INT_CYCLE);
...
signal STATE : MY_STATE;
signal TWO_BIT : bit_vector (0 to 1);
...
    STATE <= RESET;      ✓
    STATE <= "00";       ✗
    STATE <= TWO_BIT;    ✗
```

Package

- Le package contient une collection de définitions qui pourraient être référencées par plusieurs entités en même temps.
- Placer des définitions en commun à plusieurs designs en une seule location est plus adéquat pour les concepteurs



- **Exemple:** Les types standards en VHDL sont prédéfinies dans le package standard comme suit:

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  type BIT is ('0', '1');
  type CHARACTER is (-- ascii set );
  type INTEGER is range
                    implementation_defined;
  type REAL is range
                implementation_defined;
  -- BIT_VECTOR, STRING, TIME
end STANDARD;
```


Processus


- Les processus constituent la région du code VHDL où les instructions s'exécutent séquentiellement. Ils sont déclarés à l'intérieur des architectures.
- Les processus interagissent ensemble en concurrence.

```
entity ORGATE is
  port (A,B : in bit;
        Z : out bit);
end ORGATE;

architecture BEHAVE of ORGATE is
begin
  OR_FUNC: process (A,B)
  begin
    if (A='1' or B='1') then
      Z <= '1';
    else
      Z <= '0';
    end if;
  end process OR_FUNC;
```

Variables

```
process (A, B, C)
    variable M, N : integer;
begin
    M := A;
    N := B;
    Z <= M + N;
    M := C;
    Y <= M + N;
end process;
```



- ❑ Les variables ne peuvent être utilisées qu'à l'intérieur des processus.
- ❑ Les variables s'affectent avec :=

L'instruction IF

```
if CONDITION then
    -- sequential statements
end if;
```


```
if CONDITION then
    -- sequential statements
else
    -- sequential statements
end if;
```




```
if CONDITION then
    -- sequential statements
elsif CONDITION then
    -- sequential statements
elsif CONDITION then
    -- sequential statements
else
    -- sequential statements
end if;
```

Executes first true branch

- La première valeur qui est vraie sera exécutée

L'instruction CASE

object 

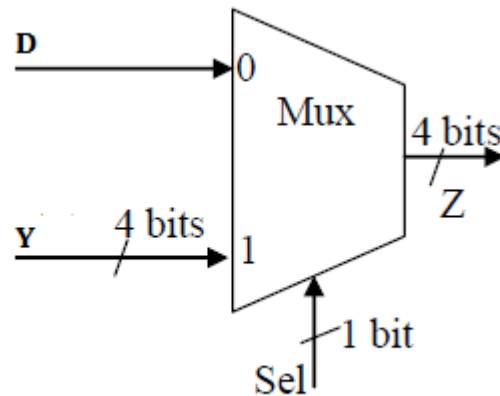
possible 
values 


```
case OBJECT is
  when VALUE_1 =>
    -- statements
  when VALUE_2 =>
    -- statements
  when VALUE_3 =>
    --statements

    --etc...
end case;
```

Les objets peuvent être des expressions et les valeurs peuvent être des constantes

Exemple du multiplexeur



```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
    port
    (
        Sel : in std_logic;
        D,Y : in std_logic_vector(3 downto 0);
        Z: out std_logic_vector(3 downto 0)
    );
end mux;
```

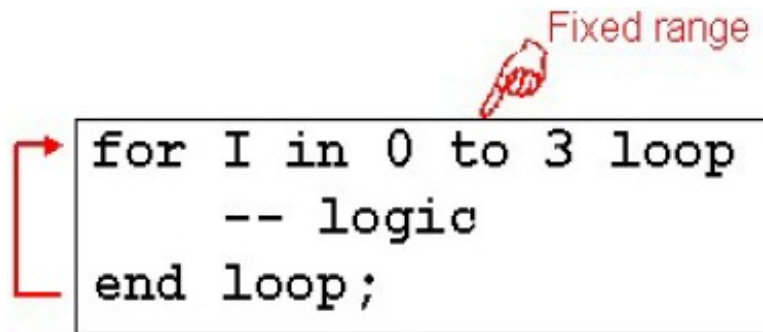
architecture comport of mux is
begin

```
    P1: process(Sel,D,Y)
    begin
        if(Sel = '1') then
            Z <= Y;
        else
            Z <= D;
        end if;
```

```
    end process P1;
```

```
end comport;
```

L'instruction FOR



Fixed range

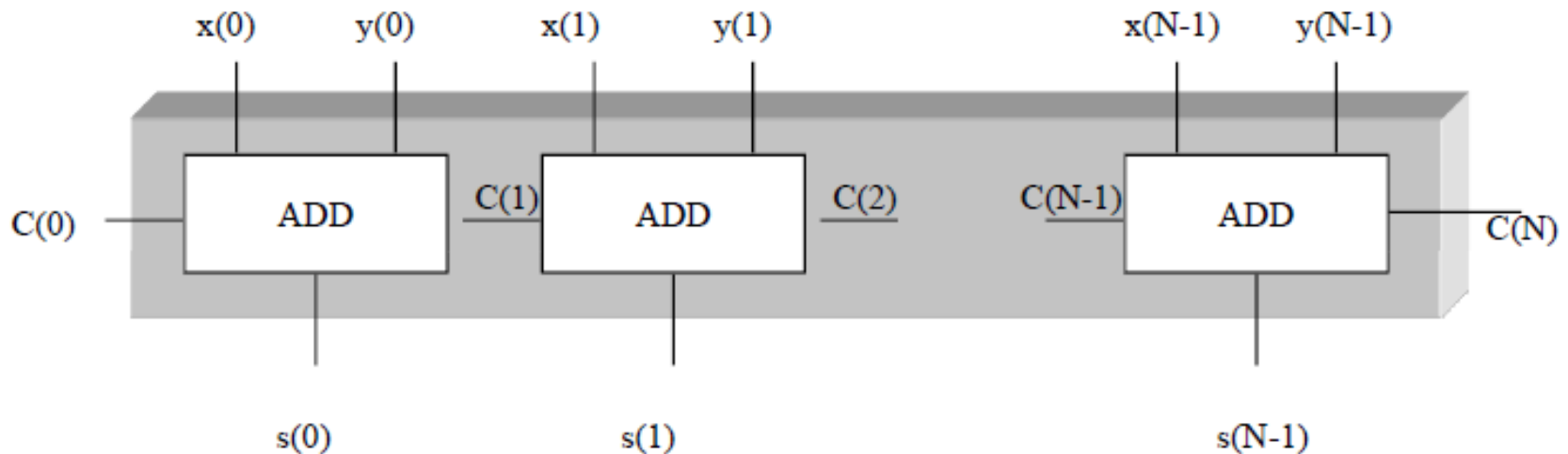
```
for I in 0 to 3 loop  
    -- logic  
end loop;
```

The diagram shows a code block for a FOR loop. A red arrow points from the '0' to the '3' in the range '0 to 3', with the text 'Fixed range' written above it. Another red arrow points from the 'end loop;' line back to the 'for' line, indicating the loop structure.

Pas besoin de déclarer la variable d'itération

Généricité

❖ Exemple explicatif: additionneur N bits



- ❑ construit à partir d'un additionneur 1 bit (un full-adder)
- ❑ assemblage des N additionneurs 1 bit afin de réaliser l'additionneur complet
- ❑ La valeur de N est inconnue avant l'instanciation du composant

- On dispose de l'entité Add :

```
entity Add is
  port (
    A, B, Cin : in std_logic;
    S, Cout : out std_logic);
end Add ;
```

- L'entité Additionneur générique s'écrit :

```
entity AdditionneurN is
  generic (N : Natural := 8);
  port (
    X, Y : in std_logic_vector ( N-1 downto 0) ;
    Cin : in std_logic;
    S : out std_logic_vector (N-1 downto 0);
    Cout : out std_logic);
end AdditionneurN ;
```

architecture structurelle of AdditionneurN is

```
  component Add
    port (
      A, B, Cin : in std_logic;
      S, Cout : out std_logic);
  end component;

  signal C : std_logic_vector(0 to N);
begin
  for I in 0 to N-1 generate
    Instance : Add
      port map (X(I), Y(I), C(I), S(I), C(i+1));
  end generate;
  C(0) <= Cin;
  Cout <= C(N);
end structurelle ;
```


❖ Intérêts de la généricité

- description de composants généraux comme les registres N bits et les additionneurs N bits
- permettre la réutilisation des composants
- Assure une plus grande rapidité de développement

❖ Méthode de programmation

- l'utilisation de **generic (.....)** dans la spécification d'entité
- l'utilisation d'une boucle for dans l'instanciation des composants à l'échelle de l'architecture:

```
for I in .... to .... generate
```

```
.....
```

```
end generate
```