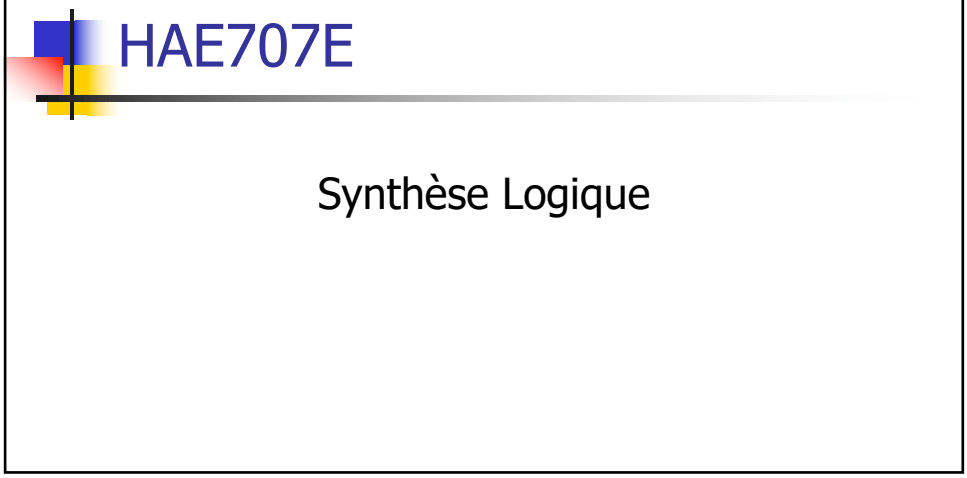
M1 EEA

Synthèse Logique
HAE707E

virazel@lirmm.fr



1

Plan du Cours

- Chapitre 1 : Introduction au VHDL
- Chapitre 2 : Les Bases du VHDL
- Chapitre 3 : Codage de FSM
- Chapitre 4 : Les Composants Reprogrammables

2

2

Synthèse Logique

Chapitre 1

Introduction au VHDL

3

Plan

- Qu'est ce que le VHDL
- Histoire du VHDL
- Contexte et Problématique
- Flot de Conception

4

4

Qu'est ce que le VHDL

- VHDL
  - **V**HSIC **H**ardware **D**escription **L**anguage
  - VHSIC : **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
- Langage de description de systèmes matériels
  - Comportement
  - Structure
- Développement de modèles exécutables
  - Simulation
  - Synthèse
- Standard IEEE
- Supporté pour tous les outils EDA (Electronic Design Automation)

5

5

Histoire

- 1980 Début du projet VHDL financé par le DoD (Departement of Defence)
- 1985 Première version publique
- 1987 Première version du standard IEEE Std 1076-1987
- 1993 Mise à jour du standard (IEEE Std 1076-1993)
- 2002 Mise à jour du standard (IEEE Std 1076-2002)

6

6

Contexte

- Forte croissance du marché électronique
- Constante amélioration des produits
  - Performances, taille, puissance …

7

7

Contexte

- Diminution de la taille et du coût des circuits grâce à l'amélioration des procédés d'intégration
  - Précision accrues des techniques de conception des circuits imprimés, nouveaux composants montés en surface, nouveaux procédés de conception de **composants submicroniques** (90 nm et wafer 300 mm) …
- Le marché des composants logiques programmables "mange" celui des ASIC

8

8

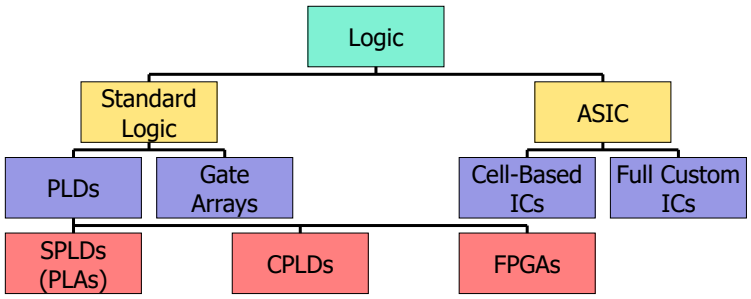
Problématique

- Réduction du *Time-To-Market*
- Adoption de nouvelles techniques de conception
  - Le langage VHDL : « **V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage »
  - Composants logiques programmables (PLD pour *Programmable Logic Device*)

9

9

PLD



SPLD	Simple Programmable Logic Device
PAL	Programmable Array of Logic
CPLD	Complex PLD (Programmable Logic Device)
FPGA	Field Programmable Gate Array

10

10

PLD

- CPLD (Complex PLD) ou FPGA
  - CPLD : de l'ordre de 10M de transistors
  - FPGA : de l'ordre de 100M de transistors
  - ASICs : de l'ordre de 1000M de transistors
- Programmation de fonctions logiques complexes pour
  - **Gagner du temps** lors de la conception
  - De **faibles volumes** de productions
  - La réalisation du **premier prototype**

11

11

VHDL

- Le VHDL permet de ne plus manipuler de netlist (schéma au niveau composants élémentaires) en proposant
  - des constructions d'un langage de haut niveau
  - des librairies (pour la réutilisation de composants)
- Le VHDL est compatible avec les outils de synthèse et de simulation (c'est un standard)

12

12

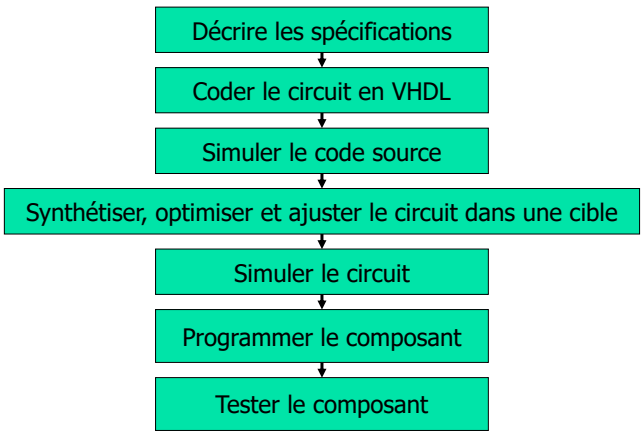
La synthèse

- Synthèse : passage d'un niveau de description à un plus bas niveau (raffinement)
- Les défauts de la synthèse
  - Perte du contrôle de l'implémentation bas-niveau à cause du niveau d'abstraction utilisé
    - Il faut maîtriser l'outil!
  - Les implémentations réalisées par l'outil de synthèse ne sont pas toujours optimales
    - Il faut adapter ses descriptions à ses objectifs
  - La qualité des outils de synthèse est variable

13

13

Flot de conception



14

14

I. Décrire les spécifications

- Les spécifications permettent de fixer
  - La fonction du circuit
  - Son initialisation
  - Les temps, la fréquence maximale d'utilisation, les chemins critiques …
- Afin de
  - Choisir une architecture pour le circuit
  - Choisir un composant

15

15

II. Coder le circuit en VHDL

- Penser votre code en terme de matériel
  - Se placer à la place du logiciel de synthèse

```
21  library IEEE;
22  use IEEE.std_logic_1164.all;
23
24  entity synth is
25  port( A, B, C, D : in integer;
26     sel : in std_logic_vector(1 downto 0);
27     Z : out integer);
28  end entity synth;
29
30  architecture behavioral of synth is
31  begin
32  with sel select
33     Z <=  A+B when "00",
34         C+D when "10",
35         0 when others;
36  end architecture behavioral;
```

Taille de(s) additionneur(s) ?

Plusieurs additionneurs ?

Quand plusieurs implémentations sont possibles **votre code doit guider les choix de l'outil de synthèse**

16

16

III. Simuler le code source

- Dans le cas d'un circuit complexe, simuler son code avant de faire la synthèse permet de gagner du temps (en évitant les erreurs de conception et la nécessité de faire une nouvelle synthèse)
- Attention, une vérification exhaustive est souvent impossible! (il existe des métriques mesurant la qualité d'une validation)

17

17

IV. Synthétiser, Optimiser et Ajuster

- Synthèse : les outils de synthèse convertissent la description VHDL en une netlist **compatible avec la technologie cible**
- Les étapes de la synthèse
  - Transformation du niveau comportemental vers le niveau RTL & *Inférence* d'opérateurs
  - Optimisation
  - Ajustage (« fit » ou « mapping ») ou « Placement - Routage »

18

18

IV. Synthétiser, Optimiser et Ajuster

- Transformation "comportemental vers RTL"
  - **Comportemental**
    - description des relations entre les entrées/sorties (**Ici**, avec un séquencement par rapport à une horloge)
  - **RTL** : « Register Transfer Level »
    - Description sous forme de registres et de fonctions combinatoires entre les entrées/sorties
- Inférence d'opérateurs
  - Recherche d'opérateurs qui peuvent être associés directement à des modules "en dur" du circuit cible (exemples : XOR, ADDER …)

19

19

IV. Synthétiser, Optimiser et Ajuster

- Optimisation :
  - L'optimisation **dépend de la cible** (et/ou de contraintes spécifiques fixés par l'utilisateur) :
    - CPLD : minimisation des fonctions « Somme de produits »
- Ajustage (CPLD) ou Placement - Routage (FPGA) :
  - Implanter les équations logiques sur une cible par itérations successives des solutions de placement et de routage …

20

20

IV. Synthétiser, Optimiser et Ajuster

- Bilan
  - De nombreuses entreprises ont créé des outils de synthèse
    - Synopsys, Cadence , Mentor Graphics, …
  - Chaque outil donne des implantations différentes car ils utilisent des algorithmes spécifiques

21

21

V. Simuler le circuit

- Vérifier, pas seulement les fonctionnalités mais aussi les **temps de propagation**
- Si échecs alors
  - Changer de circuits cibles ou de fréquence maximale dans une même classe de circuits
  - Changer les contraintes de synthèse
  - Modifier son code VHDL

22

22

V. Simuler le circuit

- Simulation post-synthèse possible mais dans la majorité des cas
  - Pas de simulation fonctionnelle après synthèse, car "*il n'y a pas de raison pour que la synthèse soit fausse*" …
  - Pas de simulation temporelle car "*les temps des simulations seraient alors trop longs*"
  - **Analyse directe des rapports de synthèse**

23

23

VI. Programmer le composant

- La synthèse produit un fichier (par exemple JEDEC) qui permet de programmer le composant cible à l'aide d'un logiciel (et d'une structure matérielle dédiée à cette opération)
- Puis TESTER …

24

24

Synthèse Logique

Chapitre 2

Les Bases du VHDL pour la Synthèse

Plan

- **Types**
- Unités de conception
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

2

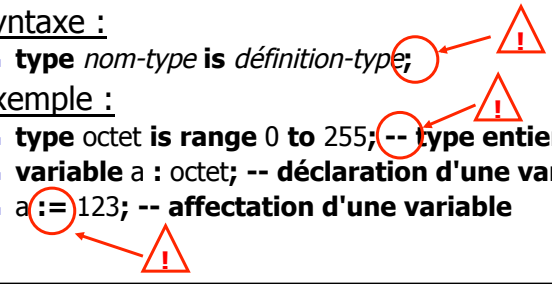
Types

- En VHDL, MAJUSCULES et minuscule sont confondues
- VHDL est un langage fortement typé
  - Tout objet (variable, signal, constante…) manipulé a un format prédéfini. Uniquement des valeurs de ce format peuvent être affectées à cet objet
- Il existe plusieurs catégories de type dont
  - *Scalaires* (numériques et énumérés)
  - *Composés* (tableaux et vecteurs)

3

Types

- Existence de types prédéfinis
  - type scalaire ⇒ bit, boolean, integer, real…
  - type composé ⇒ bit_vector…
- Type scalaire
  - possibilité de définir de nouveaux types
  - Syntaxe :
    - **type** *nom-type* **is** *définition-type*;
  - Exemple :
    - **type** octet **is range** 0 **to** 255; **-- type entier**
    - **variable** a **:** octet**; -- déclaration d'une variable de type octet**
    - a **:=** 123**; -- affectation d'une variable**



4

Types

- Types scalaires
  - Types énumérés : liste de valeurs

    type bit **is** ('0', '1')**;**

    type boolean **is** (false, true)**;**
    - Valeur d'initialisation à gauche dans la liste
  - Types *numériques* : domaine de définition
    - Domaine de définition : **range**, **to** ou **downto**

    type valim **is range** 0.0 **to** 5.0**;**

Borne de type flottant

5

Types

- Types composés
  - collections d'éléments **de même type repérés** par des valeurs d'indices

Exemple 1 :

type word **is array** (0 **to** 7) **of** bit**;**

constant mot1 **: word :=**"00000000"**;**

Exemple 2 :

- définition d'un type "vecteur non contraint" :

type bit_vector **is array** (natural **range <>**) **of** bit**;**

- déclaration d'un vecteur de bit de largeur 32 :

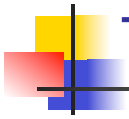
variable vecteur **:** bit_vector(31 **downto** 0)**;**

6

Types

- Exercices
  - Définir un type *etat* composé de la liste des valeurs suivantes : idle, data, jam, nosfb, error
  - Définir le type *table8x4* d'un tableau à 2 dimensions avec des indices allant de 0 à 7 et de 0 à 3, contenant des bits
  - Déclarer et initialiser une variable *exclusive_or* de type *table8x4* contenant des valeurs telles que les trois premiers bits correspondent aux entrées d'un xor et le dernier bit correspond à la valeur retournée par le xor

7



Types

8

Plan

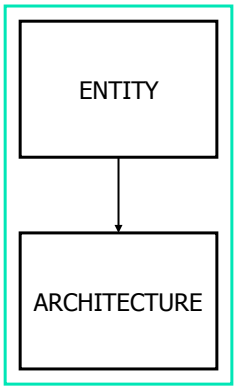
- Types
- **Unités de conception**
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

9

Unités de Conception

Unités primaires :
Déclaration des interfaces

Unités secondaires :
Définition fonctionnelles



10

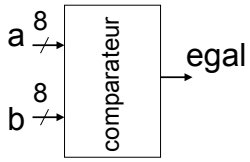
Unités de Conception

- Unités de conception primaires ⇒ ENTITY
- Définition
  - ENTITY (entité) et PACKAGE (paquetage)
  - Partie de programme qui peut être compilée séparément
    - Fichier texte contenant une ou plusieurs unités de conception primaires ou secondaires
  - ENTITY ⇒ Vue externe d'un composant
- Spécification d'entité
  - Ports d'entrées / sorties
  - Type
  - Mode : entrée (in), sortie (out), entrée/sortie (inout)

Unités de Conception

- Exemple d'entité

```
entity comparateur is
port (
      a : in bit_vector(7 downto 0);
      b : in bit_vector(7 downto 0);
      egal : out bit);
end ;
```



- Le mode IN *protège* le signal en écriture.
- Le mode OUT *protège* le signal en lecture

Unités de Conception

- Exercice
  - Écrire l'entité d'un additionneur *add4* de deux mots *a* et *b* sur 4 bits en entrée, avec une retenue entrante *ci*, et deux sorties *sum* sur 4 bits et la retenue *co*

13

Unités de Conception

- Unités de conception secondaires : ARCHITECTURE
- Toute architecture est relative à une entité
- Définition de l'architecture :
  - L'architecture définit les fonctionnalités et les relations temporelles d'un composant

```
architecture simple of comparateur is
-- zone de déclaration (ici commentaire uniquement)
begin
        egal <= '1' when a = b else '0';
        ....
end simple ;
```

14

Unités de Conception

- Il peut y avoir plusieurs architectures pour un même composant : sans délai, avec des délais…

```
architecture simple of comparateur is
begin
        egal <= '1' when a = b else '0';
        ....
end simple ;

architecture complexe of comparateur is
begin
        egal <= '1' after 10 ns when a = b else '0' after 5 ns;
        ....
end complexe ;
```

deux architectures d'un même composant

Plan

- Types
- Unités de conception
- **Simulations évènementielles**
- Instructions séquentielles et concurrentes
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

Simulations Evènementielles

- Particularités des systèmes numériques
  - Un système numérique ne "change d'état" (notion d'évènement) qu'à des instants discrets
  - Les sorties de chaque module numérique ne peuvent évoluer que lorsqu'il y a une évolution au moins d'une de leurs entrées
  - Toutes les opérations sont effectués en parallèle

17

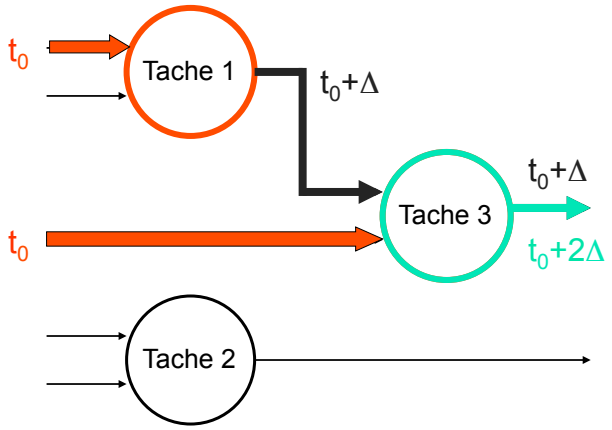
Simulations Evènementielles

- Ils existent plusieurs types d'objets
  - constante ⇒ la valeur portée ne change pas
  - variable ⇒ affectation immédiate de leur valeur
  - **signal ⇒ NOUVEAU TYPE D'OBJET**
- <u>Définition des signaux :</u>
  - Objets mémorisant l'information à des instants discrets
  - Les signaux connectent les composants entre eux (équivalent au fil dans un schéma)
  - Ils sont caractérisés par :
    - Un type
    - Une valeur initiale : ' 0 ' par défaut pour le type bit,
    - Un pilote (ou driver)

18

Simulations Evènementielles

- Intérêt des signaux dans la simulation matérielle

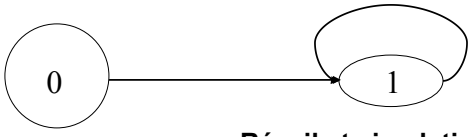


19

Simulations Evènementielles

- Intérêt des signaux dans la simulation matérielle

Avancée du temps de simulation jusqu'à t_{min}



Initialisation des signaux à leur valeur initiale

Réveil et simulation des process concernés
Production de **nouveaux événements**
Extraction des signaux avec t_{min}

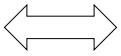
Si t_{min} est égal au temps t_0
alors il y a itération (ou *délais Delta)*

20

Simulations Evènementielles

- Définition d'un pilote :
  - Liste de couples date-valeur (date comptée relativement à l'heure actuelle du simulateur)
  - Exemple :

S **<=** '0', '1' **after** 10 ns, '0' **after** 25 ns;



Heure	Valeur
0 (Δ)	'0'
10ns	'1'
25ns	'0'

Pilote de S

Le pilote est une mémoire associée à chaque signal

21

Simulations Evènementielles

- Contrairement à une variable, un signal n'est jamais affecté instantanément mais uniquement lorsque l'exécution du *process* est suspendu

- Définition d'un processus :
  - Élément calculatoire élémentaire du simulateur
  - Les processus sont concurrents : leur ordre dans le programme n'a pas d'importance

22

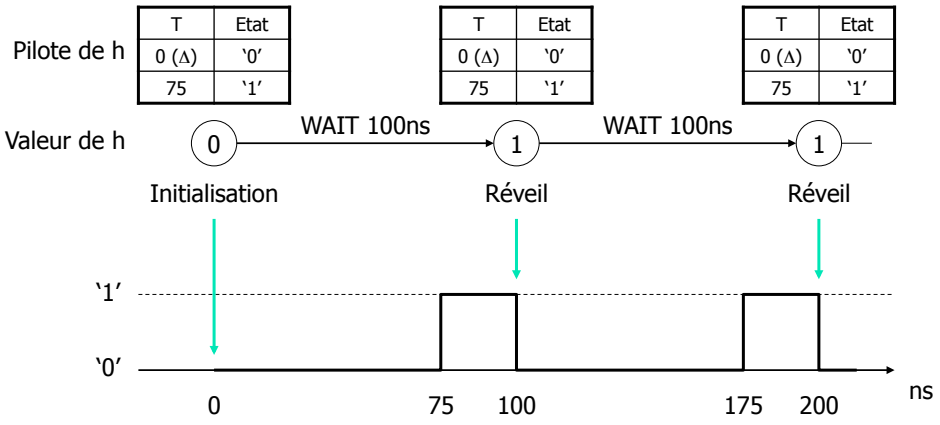
Simulations Evènementielles

- Exemple : Simulation d'un générateur d'horloge

```
...
signal h : bit;
begin
        horloge : process -- déclaration de processus
        -- zone de déclaration du process
        begin -- début de la zone de définition du processus
                h <= '0', '1' after 75 ns;
                wait for 100 ns;
        end process; -- fin du processus
...
```

Mise en suspend du process pour une durée déterminée

Simulations Evènementielles



Simulations Evènementielles

- Un processus
  - vit à partir du chargement du code exécutable dans le simulateur
  - est exécuté une fois au début de la simulation
  - meurt avec la fin de la simulation
  - peut être endormi ou suspendu (indéfiniment ou temporairement) avec l'instruction **WAIT**
  - est exécuté chaque fois qu'un évènement intervient dans sa liste de sensibilité (définie ci-après)

25

Simulations Evènementielles

- Instruction **WAIT**
  - Synchronise (en les suspendant) les processus
  - Quatre possibilités (pouvant être combinées dans un même processus) :
    - **WAIT ON** évènement (sur un ou plusieurs signaux)
    - **WAIT FOR** durée
    - **WAIT UNTIL** condition
    - **WAIT** (stoppe le processus indéfiniment)

26

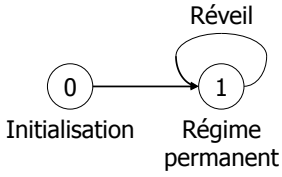
Simulations Evènementielles

- Le **AFTER** et le **WAIT FOR** sont interdits dans les descriptions pour la synthèse
- S'il y en a, ils sont généralement ignorés par les outils de synthèse
  - Comment l'outil de synthèse pourrait-il présager du délai alors que ce délai dépend principalement du routage…?

27

Simulations Evènementielles

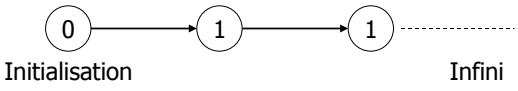
- Chaque processus s'exécute une fois à l'initialisation jusqu'à rencontrer un **WAIT**;
- puis, lorsque la condition de réveil est validée,
- l'exécution continue de façon cyclique
- Vie d'un processus



28

Simulations Evènementielles

- Remarque :
  - Il est syntaxiquement possible d'avoir un process sans *liste de sensibilité* ni **WAIT**
  - Cependant, le temps de simulation n'avance pas car ce process reboucle sur lui même infiniment (il n'est jamais suspendu)



29

Simulations Evènementielles

- Illustration de l'équivalence entre la *liste de sensibilité* et l'instruction **WAIT**

```
proc1: process (a, b, c)
       begin
         x<=a and b and c;
       end process;
```

Liste de sensibilité

```
proc2: process
       begin
         x<=a and b and c;
         wait on a, b, c;
       end process;
```

Attente d'évènements

30

Plan

- Types
- Unités de conception
- Simulations évènementielles
- **Instructions séquentielles et concurrentes**
- Descriptions structurelles et comportementales
- Description de la maquette de test
- Conclusion

31

Instructions Séquentielles et Concurrentes

- Définition d'une instruction séquentielle :
  - Instruction **OBLIGATOIREMENT** à l'intérieur d'un process
  - Instruction dont la **POSITION** dans le process à une importance sur le résultat final
- Définition d'une instruction concurrente :
  - Instruction dont la position dans le programme n'a pas d'influence
- Rappel :
  - Les instructions (séquentielles et concurrentes) se placent toujours uniquement entre le **begin** et le **end** de l'architecture

32

Instructions Séquentielles et Concurrentes

- A l'intérieur d'un processus, il ne peut y avoir que des instructions séquentielles :
  - Exécution classique dans l'ordre d'écriture
  - « ça ressemble à du C »

33

Instructions Séquentielles et Concurrentes

- Une variable ne peut exister que dans un contexte séquentiel (dans un process)
  - Affectation immédiate

    X **:=** 1+2**;**

    X prend immédiatement la valeur 3 (sans pilote)

- Seul un signal, suivant le contexte, peut être affecté de façon concurrente ou séquentielle
  - Affectation dès que le process est suspendu

34

Instructions Séquentielles et Concurrentes

```
...
architecture exercice of var_sig is
        signal aa, aaa : integer := 3;
        signal bb, bbb : integer := 2;
begin
        p1: process
                variable a: integer := 7;
                variable b: integer := 6;
                begin
                        wait for 10 ns;
                        a := 1;          -- a est égal à 1
                        b := a + 8 ;     -- b est égal à 9
                        a := b - 2 ;     -- a est égal à 7
                        aa <= a;         -- 7 dans pilote de aa
                        bb <= b ;        -- 9 dans pilote de bb
                end process;
```

35

Instructions Séquentielles et Concurrentes

```
        p2: process
                begin
                        wait for 10 ns;
                        aaa <= 1 ;         -- 1 dans pilote de aaa
                        bbb <= aaa + 8;    -- 11 dans pilote de bbb
                        aaa <= bbb - 2;    -- 0 dans pilote de aaa
                end process;
end;
```

Seule la dernière affectation compte !!!

36

Instructions Séquentielles et Concurrentes

- Équivalence séquentiel/concurrent
  - Toute instruction concurrente peut être décrite grâce à une instruction séquentielle équivalente "utilisant un processus"

37

Instructions Séquentielles et Concurrentes

- Affectation séquentielle simple

```
p1: process
        begin
                wait on a, b ;
                s <= a and b;
end process;
```

38

Instructions Séquentielles et Concurrentes

- Affectation séquentielle conditionnelle

```
p2: process
    begin
        wait on etat ;
        if etat = "1001" then
                neuf <= ´1´ ;
        else
                neuf <= ´0´ ;
        end if;
end process;
```

Permet de réaliser la synthèse de portes logiques combinatoires
(si toutes les sorties sont assignées sinon création de latch)

39

Instructions Séquentielles et Concurrentes

Non obligatoire

```
neuf  <=  '1' when etat = "1001" else '0';
```

- Dans les deux cas (**if** et **when**)
  - Non exclusivité des conditions MAIS ordre de priorités!
  - Attention à la logique inutile lors de la synthèse

40

Instructions Séquentielles et Concurrentes

- Affectation séquentielle sélective

```
p3: process
    begin
        wait on e0, e1, e2, e3, ad;
        case ad is
            when "00" => s <= e0 ;
            when "01" => s <= e1 ;
            when "10" => s <= e2 ;
            when others => s <= e3 ;
        end case;
    end process;
```

Exclusivité des conditions
MAIS
Pas d'ordre de priorités!

41

Instructions Séquentielles et Concurrentes

- Affectation concurrente sélective

```
signal e0, e1, e2, e3, s : bit;
signal ad : bit_vector( 1 downto 0);
begin
    with ad select
        s <=    e0 when "00",
                e1 when "01",
                e2 when "10",
                e3 when others;
end;
```

Pas de priorité

Obligatoire quand toutes les conditions n'ont pas été couvertes

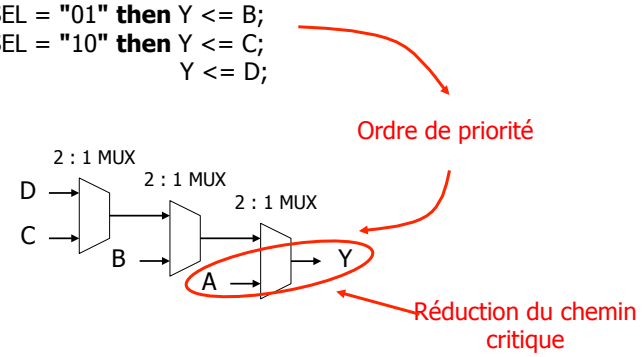
- Structure de type multiplexeur

42

Instructions Séquentielles et Concurrentes

- Ordre de priorité : **if** versus **case**

```
if      SEL = "00" then Y <= A;
elseif  SEL = "01" then Y <= B;
elseif  SEL = "10" then Y <= C;
else                    Y <= D;
end if;
```

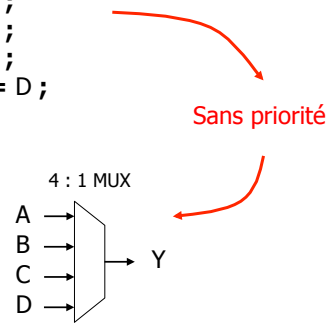


43

Instructions Séquentielles et Concurrentes

- Ordre de priorité : **if** versus **case**

```
case SEL is
   when "00" => Y <= A ;
   when "01" => Y <= B ;
   when "10" => Y <= C ;
   when others => Y <= D ;
end case;
```



44

Instructions Séquentielles et Concurrentes

- Boucles

```
while i<10 loop
    i := i + 1;
    ...
end loop;
```

Il faut déclarer la variable i

```
for i in 10 downto 1 loop
    -- instructions utilisant i
    ...
end loop;
```

Il ne faut pas déclarer la variable i

45

Instructions Séquentielles et Concurrentes

- Exemple : and logique de tous les bits de a_bus

```
architecture fonctionne of mon_and is
begin
p1:process (a_bus)
    variable x_temp : bit;
    begin
        x_temp := ' 1 ';
        for i in 7 downto 0 loop
            x_temp := a_bus(i) and x_temp;
        end loop;
        x <= x_temp;
    end process;
end fonctionne;
```

46

Instructions Séquentielles et Concurrentes

- Précédence
  - Il n'y a pas d'ordre de précédence entre les opérateurs logiques élémentaires
  - Il faut obligatoirement des parenthèses s'il y a un doute… sinon cela conduit à une erreur de compilation

X <= A or B and C

X <= A or (B and C)

47

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique combinatoire
  - Décrire (en utilisant uniquement des instructions concurrentes) le module *simple_alu* possédant deux entrées a et b de type vecteur de bit de largeur 4, une entrée ctrl de type booléen et une sortie z de type vecteur de bit de largeur 4 également
  - *simple_alu* est sensible à a, b et ctrl.
  - de plus :
    - Si ctrl est vrai alors
      - z = a et b
    - Sinon
      - z = a ou b

48

Instructions Séquentielles et Concurrentes

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule RS

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule Latch

51

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule D avec RAZ synchrone

52

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule D avec RAZ asynchrone

53

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'une bascule JKFF

54

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'un registre

55

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'un compteur 4 bits

56

Instructions Séquentielles et Concurrentes

- Exercice d'application : Modélisation d'éléments de logique synchrone
  - Modélisation d'un compteur permettant de compter de 3 (0011) à 11 (1011) par pas de 2
    3 : 0011
    5 : 0101
    7 : 0111
    9 : 1001
    11 : 1011
  - Ce compteur doit aussi posséder une entrée d'initialisation (init) asynchrone permettant de placer le compteur dans son état initial 3 (0011).

57

Solution

58

Instructions Séquentielles et Concurrentes

- Conseils pour la synthèse
  - Ne pas donner de valeur d'initialisation pour les signaux
  - Spécifier explicitement le nombre de bits pour chaque signal
  - Utiliser les valeurs '–' lors de l'affectation de signaux permet à l'outil de synthèse d'optimiser les fonctions booléennes
  - Utiliser les parenthèses permet de contrôler le traitement concurrent de fonctions combinatoires (donc les délais)
  - Utiliser l'instruction "select" permet de générer moins de logique car sans notions de priorité

59

Instructions Séquentielles et Concurrentes

- Quelques exemples

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inference is
port(       sel : in std_logic;
            x, y, z : in std_logic;
            w : out std_logic);
end entity inference;

architecture behavioral1 of inference is
begin
 process(x,y,z,sel) is
 begin
  if (sel='1') then
   w <= x and y;
  else
   w <= x and z;
  end if;
  end process;
end architecture behavioral1;
```

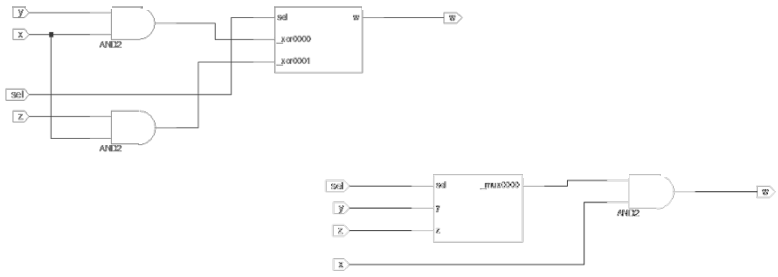
Est-ce que behavioral 1 et 2 produisent le même résultat de synthèse ?

```
architecture behavioral2 of inference is
begin
process(x,y,z,sel) is
        variable right : std_logic;
        begin
        if (sel='1') then
                right := y;
        else
                right := z;
        end if;
        w <= x and right;
end process;
end architecture behavioral2;
```

60

Instructions Séquentielles et Concurrentes

- La synthèse de l'architecture behavioral1 génère deux opérateurs AND alors que behavioral2 un seul AND précédé d'une sélection des opérandes



61

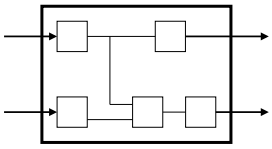
Plan

- Types
- Unités de conception
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- **Descriptions structurelles**
- Description de la maquette de test
- Conclusion

62

Descriptions structurelles

Circuit



Code VHDL

```
entity comparateur is
port (
        signal a : in bit_vector(7 DOWNTO 0);
        signal b : in bit_vector(7 DOWNTO 0);
        signal egal : out bit);
end entity;
…..
```

Description Comportementale
⇒ *Décrit ce que la fonction doit réaliser*

Description Structurelle
⇒ *Décrit comment la fonction est réalisée*

63

Descriptions structurelles

- Description de type hiérarchique des interconnexions entre composants
  - Contient un ou plusieurs composants
    - COMPONENT
  - Structures imbriquées à un ou plusieurs niveaux hiérarchiques

64

Descriptions structurelles

- Exemple : Compteur 4 bits synchrone avec autorisation de comptage

```
entity compteur4 is
port ( h, raz, compter : in bit;
       sortie : out bit_vector( 3 downto 0);
       plein : out bit);
end;
architecture structure1 of compteur4 is
signal d, s , sb : bit_vector( 3 downto 0);
component bascule
port (h, d, raz : in bit;
      s, sb : out bit);
end component;
component calcul
port ( s, sb : in bit_vector( 3 downto 0);
       compter : in bit;
       d : out bit_vector( 3 downto 0);
       plein : out bit);
end component;
```

Déclarer les signaux internes nécessaires

Déclarer tous les composants nécessaires

Descriptions structurelles

```
begin
    ba : bascule -- instanciation par position
    port map ( h, d(3), raz, s(3), sb(3));
    bb : bascule -- instanciation par dénomination
    port map ( h => h, d => d(2), raz => raz, s => s(2), sb => sb(2));
    bc : bascule -- instanciation par position et dénomination
    port map ( h, d(1), sb => sb(1), s => s(1), raz => raz);
    bd : bascule -- instanciation par dénomination
    port map ( sb => sb(0), s => s(0), h => h, d => d(0), raz => raz);

    combi : calcul
    port map ( s, sb, compter, d, plein);

    sortie <= s;
end structure1;
```

Instancier chaque composant en indiquant sa liste de connexions

Descriptions structurelles

- Description du composant 'Calcul'

```
entity calcul is
port (      s, sb       : in bit_vector( 3 downto 0);
            compter     : in bit;
            d           : out bit_vector( 3 downto 0);
            plein       : out bit);
end;
architecture par_10 of calcul is
signal pas_compter : bit;
begin
    pas_compter <= not compter;
    d(3) <= (compter and s(2) and s(1) and s(0))
    or (s(3) and (sb(0) or pas_compter)) ;
    d(2) <= (compter and sb(2) and s(1) g s(0))
    or ( s(2) and (pas_compter or sb(1) or sb(0)));
    d(1) <= (compter and sb(3) and sb(1) and s(0))
     or ( s(1) and (pas_compter or sb(0)));
    d(0) <= compter xor s(0);
    plein <= s(3) and s(0);
end ;
```

Description comportementale (sans instanciation)

67

Descriptions structurelles

- Lorsque plusieurs architectures d'une même entité existent, que se passe t-il?
  - Sans spécification explicite de la configuration, c'est la dernière architecture analysée dans la bibliothèque WORK qui est utilisée
- Comportement aléatoire à éviter!

68

Descriptions structurelles

- Instanciation directe

```
combi : calcul
port map ( s, sb, compter, d, plein);

combi : entity calcul(par_10)
port map ( s, sb, compter, d, plein);
```

- L'instanciation directe permet
  - de préciser l'architecture utilisée
  - d'éviter la déclaration préalable du composant

69

Descriptions structurelles

- Une spécification de configuration permet de modifier ces associations par défaut en précisant quelle unité de conception doit être employée pour un composant

for *liste_labels* : *composant* **use entity** *librairie.entité*[(*architecture*)]

Nom des labels des composants

Nom du composant à configurer

Unité de conception à utiliser

70

Descriptions structurelles

- GENERATE permet de réaliser des instanciations multiples

```
ba : bascule -- instanciation par position
port map ( h, d(3), raz, s(3), sb(3));
bb : bascule -- instanciation par dénomination
port map ( h => h, d => d(2), raz => raz, s => s(2), sb => sb(2));
bc : bascule -- instanciation par position et dénomination
port map ( h, d(1), sb => sb(1), s => s(1), raz => raz);
bd : bascule -- instanciation par dénomination
port map ( sb => sb(0), s => s(0), h => h, d => d(0), raz => raz);
```

```
implant : for i in 0 to 3 generate
            b : bascule
            port map (h, d(i), raz, s(i), sb(i));
end generate;
```

71

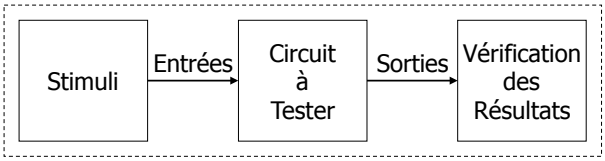
Plan

- Types
- Unités de conception
- Simulations évènementielles
- Instructions séquentielles et concurrentes
- Descriptions structurelles
- **Description de la maquette de test**
- Conclusion

72

Description de la maquette de test

- Comment simuler les description VHDL
  - Utiliser un Testbench ⇒ Maquette de Test



Un Testbench est une entité dénuée d'entrées/sorties

Description de la maquette de test

```
entity test_compteur4 is
end;

architecture tb of test_compteur4 is
-- déclaration des signaux utiles pour les connexions
signal h, raz, compter, plein : bit;
signal sortie : bit_vector(3 downto 0);
begin
        -- instanciation directe
        c1: entity work.compteur4(decade)
                port map(h, raz, compter, sortie, plein);

        h <= not(h) after 10 ns;
        compter <= '0', '1' after 100 ns;
        raz <= '1', '0' after 200 ns, '1' after 400 ns, '0' after 500 ns;
end;
```

Aucun port d'entrée/sortie

Instancier le composant à tester

Description des stimuli

Synthèse Logique

Chapitre 3

Codage de FSM

Plan

- **Introduction**
- Moore v.s. Mealy
- Exercices

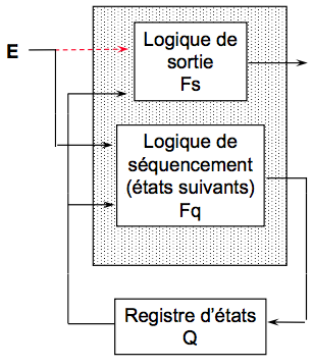
Introduction

- Les machines à états (FSM) sont des composants couramment utilisés, nous allons voir comment :
  - Les décrire
  - En faire la synthèse pour atteindre les performances optimales (notion de compromis Temps/Fréquence)
  - Les utiliser dans un système complexe (avec une hiérarchie)
  - Nous terminerons ce chapitre et le cours par l'étude d'exemples de conception

3

Moore v.s. Mealy

- Deux blocs combinatoires (calcul de l'état suivant et des sorties)
- Un registre d'état à N entrées parallèles
- Etats codés sur N bits, N = $\log_2$ (Q)



Quintuplet : (E, S, Q, Fs, Fq)

E: Ensemble des entrées primaires
S : Ensemble des sorties primaires
Q : Ensemble des états
Fs : Fonction de sortie (Moore / Mealy)
Fs = f (Q) => Moore
Fs = f (Q, E) => Mealy
Fq : Fonction de transition d'états

4

Méthode d'Huffman-Mealy

- Modélisation du cahier des charges
  - Graphe d'état
  - Table d'état
- Minimisation du nombre d'états
  - Règles de minimisation
  - Détermination du nombre de bascules minimum
- Codage des états
  - Codage des états
  - Codage des entrées de bascules
- Synthèse
  - Synthèse des entrées de bascules
  - Synthèse des sorties

5

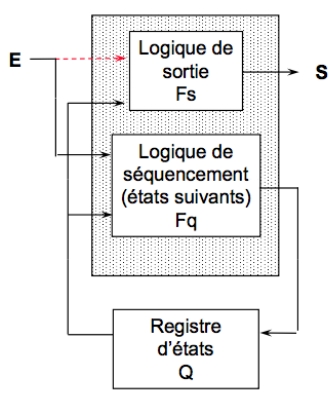
Méthode d'Huffman-Mealy

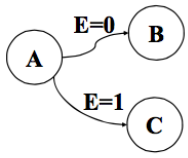
- Modélisation du cahier des charges
  - Graphe d'état
  - Table d'état
- ~~Minimisation du nombre d'états~~
  - ~~Règles de minimisation~~
  - ~~Détermination du nombre de bascules minimum~~
- ~~Codage des états~~
  - ~~Codage des états~~
  - ~~Codage des entrées de bascules~~
- ~~Synthèse~~
  - ~~Synthèse des entrées de bascules~~
  - ~~Synthèse des sorties~~

Codage VHDL

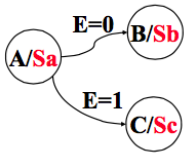
6

Graphe d'Etat

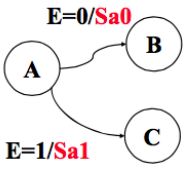




Machine de Moore



Machine de Mealy



7

Exemple : Modélisation

- Cahier des charges : Le système considéré a une entrée (E) et une sortie (S). Il reçoit sur son entrée des bits arrivant en série. La sortie (S) doit passer à 1 chaque fois qu'une séquence 010 apparait sur l'entrée (E) puis repasser à 0 sur le bit suivant quel que soit sa valeur.

8

Exemple : Codage VHDL

- FSM à deux process

```
-- déclaration des ports d'entrées/sorties de la FSM
entity example is port (
    h, e    : in bit;
    s       : out bit);
end example;

architecture state_machine of example is
-- Déclaration d'un type énuméré composé des noms des états
    type StateType is (A, B, C, D);
-- Déclaration de deux signaux du type précédent
    signal present_state, next_state : StateType;
begin
```


Exemple : Codage VHDL

```
state_comb : process(present_state, E)
begin
    case present_state is
        when A => S <= '0' ;
                  if E = '0' then next_state <= B ;
                  else next_state <= A ;
                  end if ;
        when B => S <= '0' ;
                  if E = '1' then next_state <= C ;
                  else next_state <= B ;
                  end if ;
        when C => S <= '0' ;
                  if E = '0' then next_state <= D ;
                  else next_state <= A ;
                  end if ;
        when D => S <= '1' ;
                  if E = '0' then next_state <= B ;
                  else next_state <= A ;
                  end if ;
    end case;
end process state_comb;
```

On définit quand on est dans un état **toutes les sorties** et **l'état suivant** en fonction des entrées

Exemple : Codage VHDL

```
state_clocked : process(h)
begin
    if (h'event and h= '1') then
            present_state <= next_state ;
    end if;
end process state_clocked;

end architecture state_machine;
```

Second process qui décrit le passage d'un état à l'autre sur les fronts montants de l'horloge

- Remarque :
  - Cette description correspond à une machine de Moore car les sorties dépendent uniquement de l'état de la machine et pas des entrées.
  - Pour une machine de Mealy, on aurait des if…then ou des case… pour affecter les sorties en fonction des entrées dans le premier process

11

Comparaison Moore / Mealy

Graphe d'état (Moore)

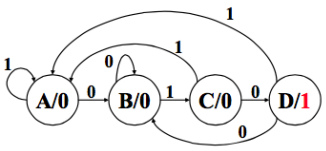


Table d'état

Etats	Etats Suivants 0	Etats Suivants 1	Sortie
A	B	A	0
B	B	C	0
C	D	A	0
D	B	A	1

Graphe d'état (Mealy)

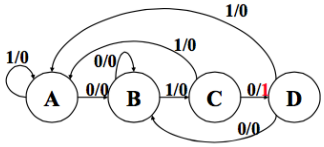


Table d'état

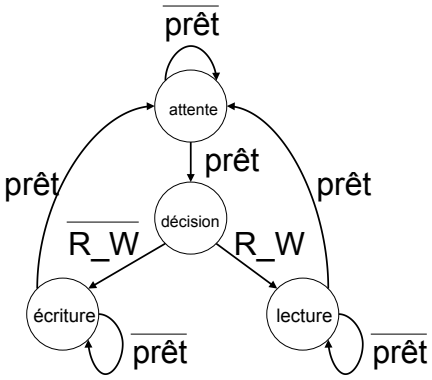
Etats	Etats Suivants 0	Etats Suivants 1	Sortie 0	Sortie 1
A	B	A	0	0
B	B	C	0	0
C	D	A	1	0
D	B	A	0	0

12

Comparaison Moore / Mealy

13

Exercice 1



États	Sorties OE	WE
Attente	0	0
Décision	0	0
Écriture	0	1
Lecture	1	0

14

Solution

15

Solution

16

Solution

17

Remarque

- Noter que l'utilisation de **if**…**then** au lieu de **case**… est aussi possible dans le premier **process** pour tester le *present_state*.
- Cependant, le **if**…**then** ajoute une notion de priorité et donc de la logique supplémentaire…

18

Description de FSM

- Problème de la mémorisation implicite :
  - Dans un process « non clocké » (combinatoire), si dans un case… un signal est spécifié seulement sous certaines conditions, alors **la synthèse produit des bascules (non désirées!) pour mémoriser l'état du signal dans tous les cas non spécifiés**
  - Dans le process state_comb (« non clocké »), **il faut affecter toutes les sorties dans toutes les conditions**

19

Description de FSM

- Problème du reset :
  - La simulation utilise comme valeur initiale les valeurs à gauche des énumérés : ici, idle est l'état initial de la machine
  - Après synthèse, l'état initial peut être n'importe quel état (et dans certains cas, cet état peut ne pas être atteignable par la machine), aussi il est recommandé **d'ajouter un reset dans l'un ou l'autre des deux process**

20

Description de FSM

- Solution Synchrone

```
...
state_comb:process(reset, present_state, read_write, ready)
        begin
        if (reset='1') then
                oe <= '-'; we <= '-';
                next_state <= idle;
        else
                case present_state is
                        when idle => oe <= '0'; we <= '0';
                                if ready = '1' then
                        ....
                end case;
        end if;
end process state_comb;
...
```

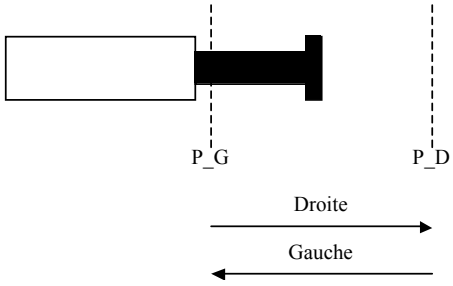

Description de FSM

- Solution Asynchrone

```
...
state_clocked:process(clk, reset)
begin
        if (reset='1') then
                present_state <= idle;
        elsif (clk'event and clk='1') then
                present_state <= next_state;
        end if;
end process state_clocked;
...
```

Exercice 2

- Considérer le vérin de la figure ci-dessous. Les capteurs P_D et P_G sont à 1 quand le vérin est respectivement à droite et à gauche. Les ordres Droite et Gauche font sortir et rentrer le vérin respectivement. La commande Start lance le système.



23

Exercice

- Cahier des charges
  - Le vérin sort jusqu'à la position P_D.
  - Il attend 3 cycles d'horloge
  - Puis rentre pour atteindre la position P_G
  - Attend encore 3 cycles d'horloge
  - La commande Start relance le même cycle.
- Exercice :
  - Représenter le fonctionnement du vérin sous forme de graphe d'états
  - Donner le code VHDL du système.

24

Solution

25

Solution

26

Solution

27

Synthèse Logique

Chapitre 4

Les Composants Reprogrammables

Plan

- **Introduction**
- SPLD
- CPLD
- FPGA
- Conclusion

Introduction

- Objectifs
  - Nous allons succinctement étudier les principales architectures et technologies des composants reprogrammables suivants
    - SPLD ⇒ Simple Programmable Logic Device
    - CPLD ⇒ Complex Programmable Logic Device
    - FPGA ⇒ Field Programmable Gate Array
  - Connaître les spécificités de ces composants afin de définir des critères de choix

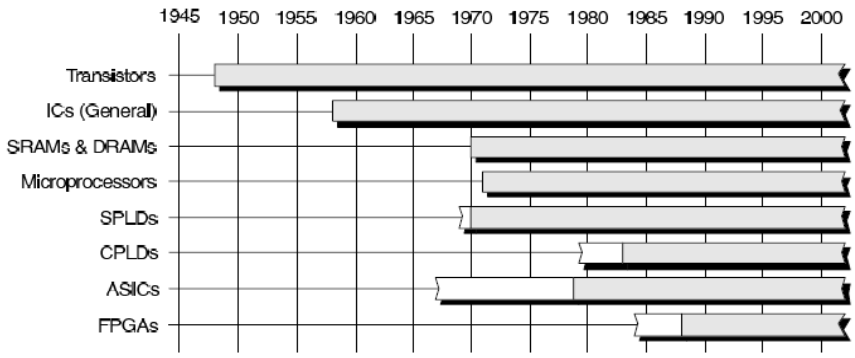
3

Introduction

- Pourquoi utiliser la logique reprogrammable?
  - Dans les années 70 sont apparus les familles de circuits intégrés TTL 74 et CMOS 4000 pour la conception de systèmes numériques
    - Composants discrets de faible complexité (SSI: Small Square Integration) ou moyenne (MSI : Medium …)
  - Ancien flot de conception
    - Spécifications / table de vérité / expression booléenne / implémentation (suivant composants disponibles, coût et performances)

4

Introduction



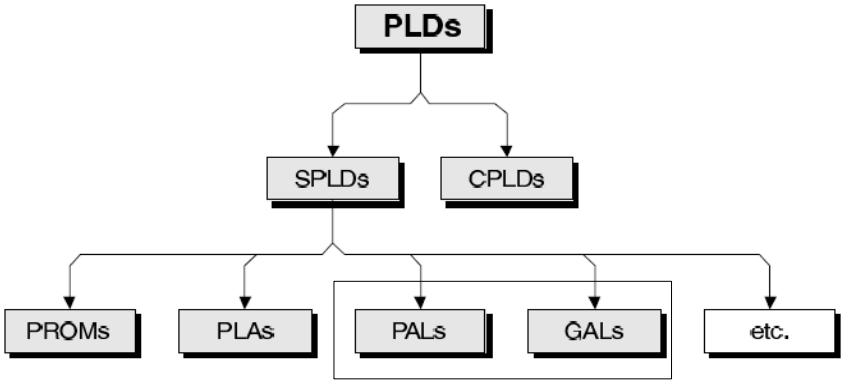
5

Plan

- Introduction
- **SPLD**
- CPLD
- FPGA
- Conclusion

6

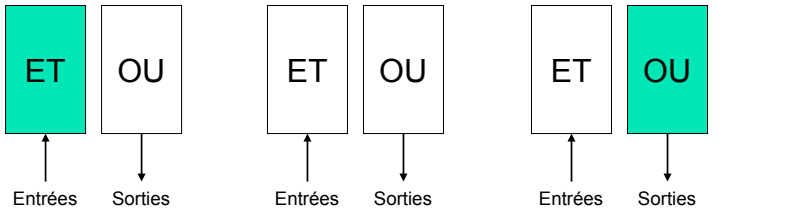
SPLD



1979

7

SPLD

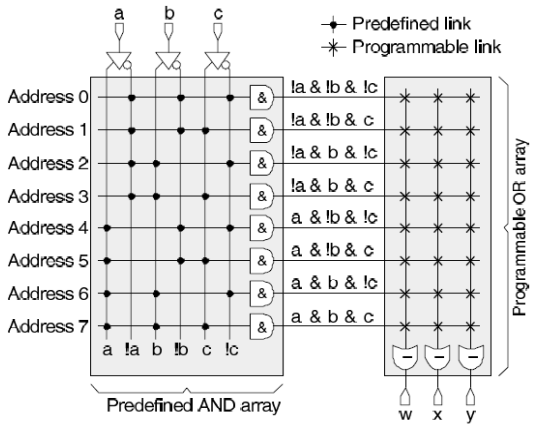


Programmation figée

8

SPLD

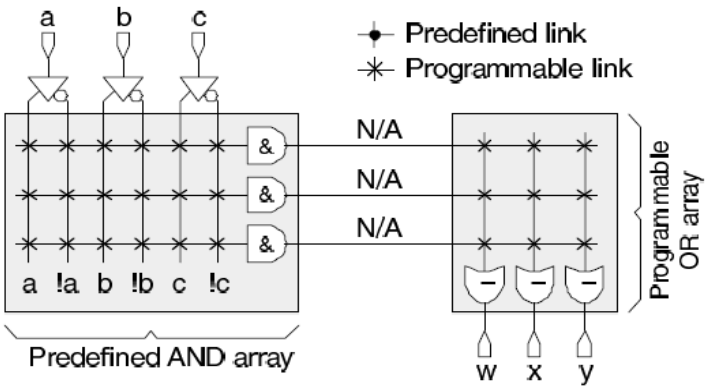
- Architecture des CPLD - PROM



9

SPLD

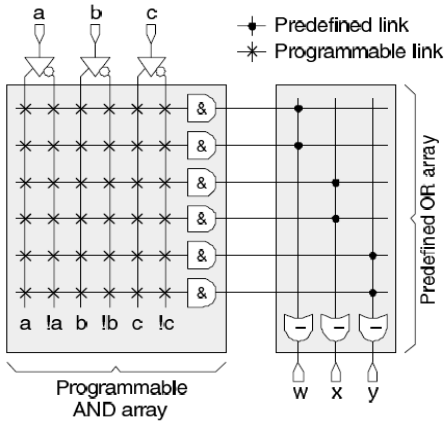
- Architecture des CPLD - PLA



10

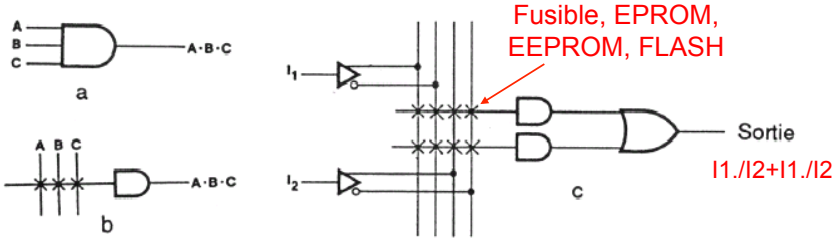
SPLD

- Architecture des CPLD – PAL/GAL



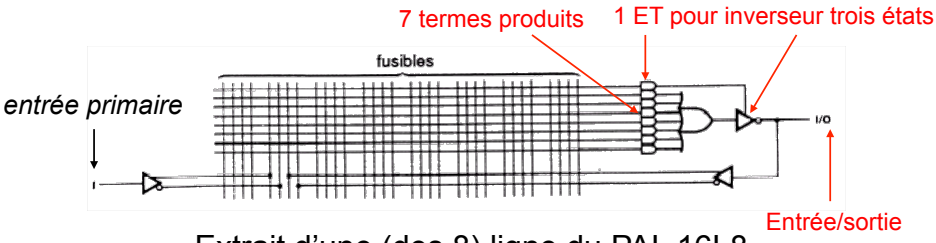
SPLD

- Architectures des SPLD – PAL
  - PAL ⇒ Programmable Array Logic
  - Tableaux de portes AND programmables et OR fixes



SPLD

- 16L8 ⇒ 16 entrées (vers 8*8=64 NAND) et 8 sorties (8 OR fixes)
  - 10 entrées primaires
  - 6 entrées proviennent des sorties (I/O)

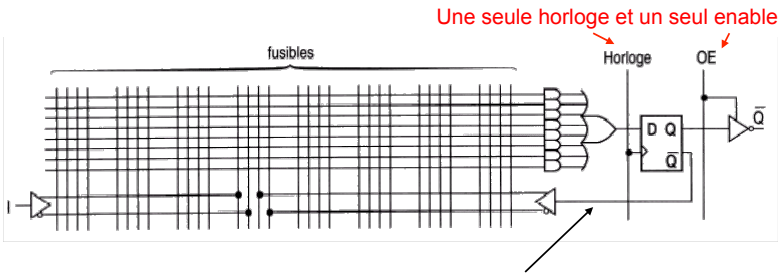


Extrait d'une (des 8) ligne du PAL 16L8

13

SPLD

- 16R8
  - R indique la présence de registres
  - Possibilité d'implanter des fonctions séquentielles (machine d'états, compteur…)
  - NAND à 16 entrées possibles et 8 sorties avec registres (16R4: 4 sorties avec registre et 4 sorties combinatoires)



Rebouclage permettant de générer des fonctions combinatoires d'un état

14

Plan

- Introduction
- SPLD
- **CPLD**
- FPGA
- Conclusion

15

CPLD

Société	Type de circuit	Site Web
Altera	CPLD, FPGA	www.altera.com
Actel	FPGA	www.actel.com
ATMEL	FPGA	www.atmel.com/atmel/products
Cypress	FPGA, CPLD	www.cypress.com/pld/index.html
Lattice Semiconductor Corporation (+ Vantis)	FPGA, CPLD	www.latticesemi.com
Lucent	FPGA	www.lucent.com/micro/netcom/orca
Quick Logic	FPGA	www.quicklogic.com
Xilinx	FPGA, CPLD	www.xilinx.com

16

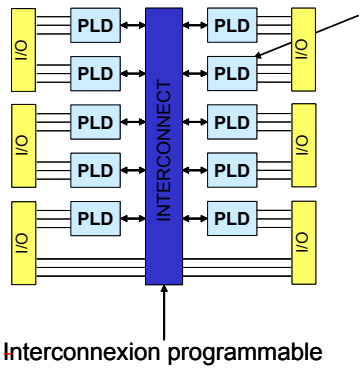
CPLD

- Complex Programmable Logic Device
  - Un CPLD contient plusieurs blocs logiques du type 22V10 (22 entrées / 10 sorties)
  - Les blocs communiquent par l'intermédiaire d'interconnexions programmables

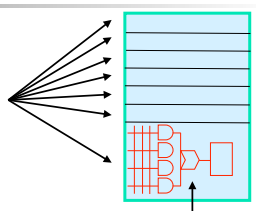
17

CPLD

- Architecture



1 PLD contient plusieurs macrocellules



Macrocellule + Tableau de ET programmable et OU fixe

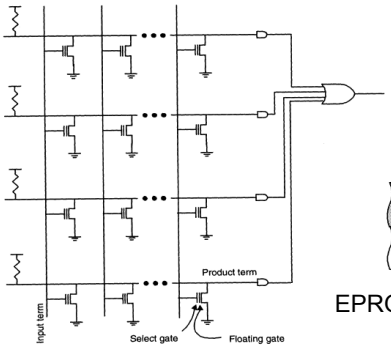
XCR3064XL Xilinx :

- **64 macrocellules**
- **1500 portes**
- **64 bascules**

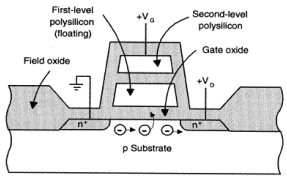
18

CPLD

- Les technologies EPROM, EEPROM et FLASH sont utilisées pour établir les connections



Floating gate avalanche injection MOS transistor => Capture des e⁻ Transistor jamais conducteurs



EPROM : Programmable électriquement et effaçable par UV

19

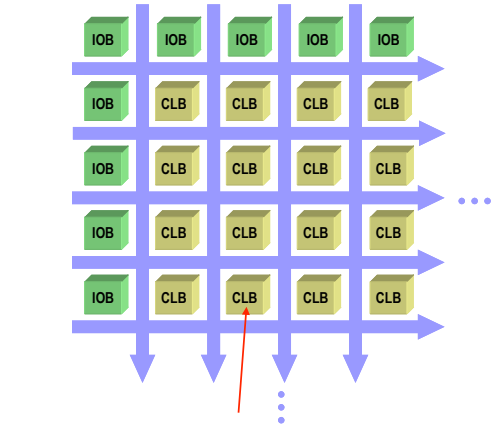
Plan

- Introduction
- SPLD
- CPLD
- **FPGA**
- Conclusion

20

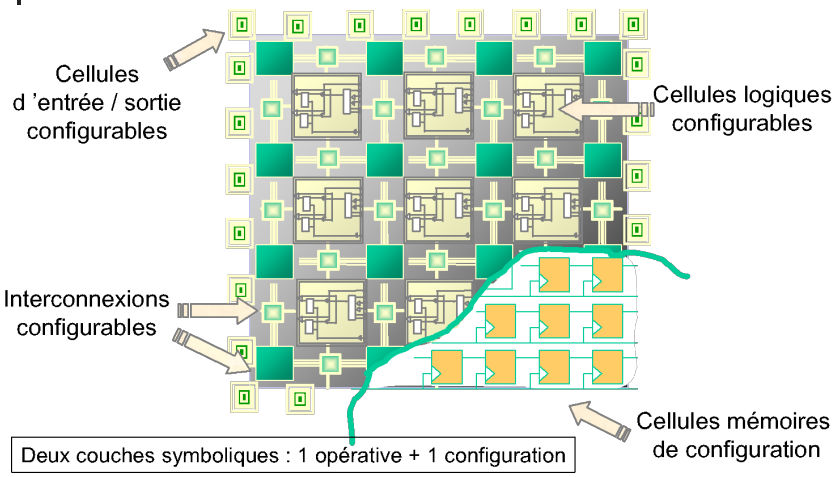
FPGA

- Field Programmable Gate Array
  - Tableau de blocs logiques programmables interconnectés entre eux (et vers les entrées/sorties) par des canaux de routage



21

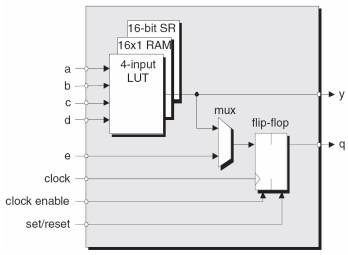
FPGA



22

FPGA

- Architecture Xlinx
  - Chaque fabriquant de FPGA a sa propre terminologie pour décrire ses circuits
  - Chez Xilinx
    - Élément de base : Logic Cell
    - Slice = 2 Logic Cell
    - Configurable Logic Bloc
      - CLB = 2 ou 4 Slices



Logic Cell

23

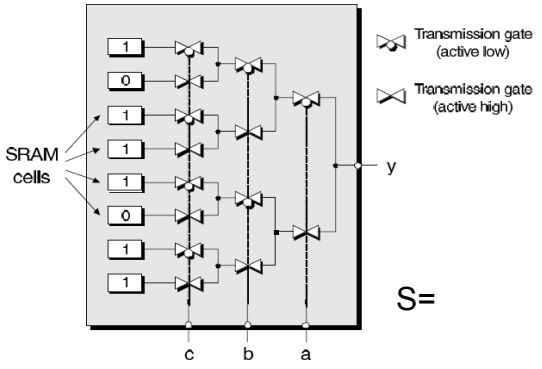
FPGA

- Architecture Altera
  - Bloc élémentaire : Logic Element (LE)
    - Le LE est très similaire au LC de Xilinx
  - Le ALM (Adaptative Logic Module) correspond au Slice de chez Xilinx
    - Le ALM correspond à 2 LE
  - Le LAB (Logic Array Bloc) correspond au CLB de chez Xilinx
    - LAB = 4 ALM

24

FPGA

- LUT ⇒ Look-Up Table
  - Une mémoire de taille 2^n peut implanter n'importe quelle fonction d'au plus n variables



A	B	C	S

S=

FPGA

- Deux principales technologies existent
  - Anti-Fusible (Actel, Cypress, Quicklogic, Xilinx…)
  - SRAM (Altera, Lucent Technologies, Atmel, Xilinx…). C'est la « technologie majeure ».
- De ces technologies dépendent l'architecture du routage et des blocs logiques

FPGA

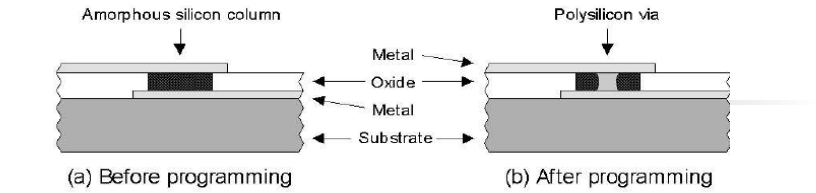
Technology	Symbol	Predominantly associated with ...
Fusible-link		SPLDs
Antifuse		FPGAs
EPROM		SPLDs and CPLDs
E²PROM/ FLASH		SPLDs and CPLDs (some FPGAs)
SRAM		FPGAs (some CPLDs)

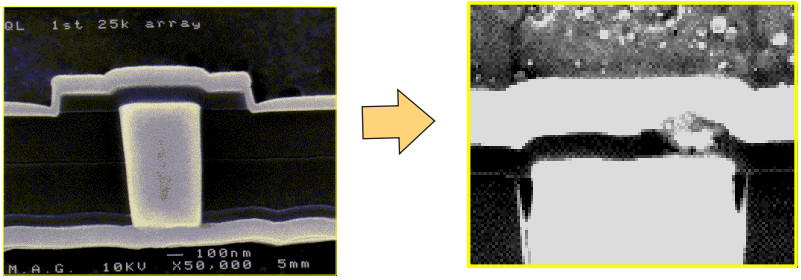
27

FPGA

- Anti-Fusible
  - Simplement un via (initialement isolant) qui, lors de la programmation (avec une « sur »-tension de 10 à 12V) devient conducteur, créant ainsi un contact entre deux lignes
  - Un anti-fusible est de la taille d'un via! (auquel il faut ajouter un transistor et une logique d'adressage pour la programmation)
  - Grande densité d'intégration des éléments programmables
  - Mais une fois programmés, ils ne sont pas reprogrammables!

28





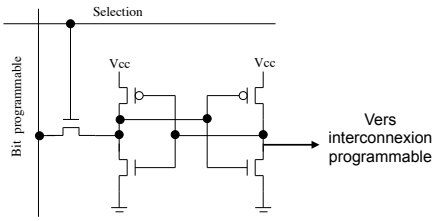
100 nm

Vue transversale fusible "ViaLink" de Quicklogic (avant et après programmation)

29

FPGA

- Technologie SRAM
  - Les points mémoires SRAM peuvent être utilisés pour contrôler des transistors créant des interconnexions ou configurer les LUT

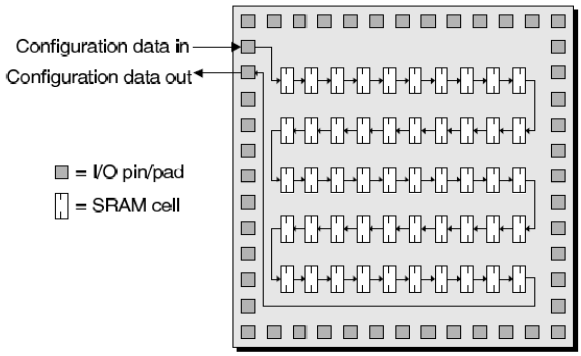


Cellule mémoire SRAM

30

FPGA

- Technologie SRAM
  - La programmation des FPGA SRAM est réalisée par l'intermédiaire d'une interface série et d'un Bitstream (flux de bit)



31

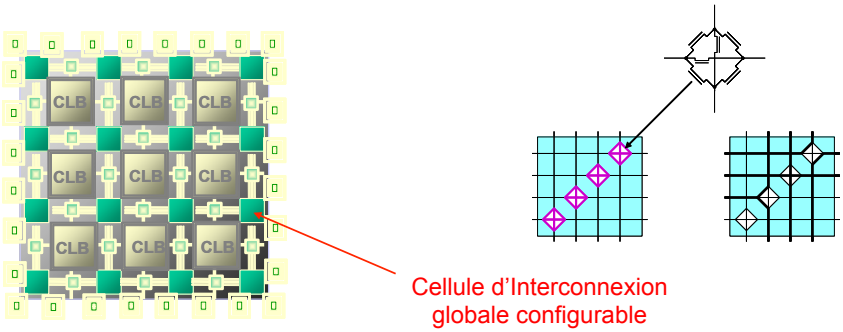
FPGA

- Technologie SRAM
  - Les FPGA SRAM sont reprogrammables in-situ par l'intermédiaire de leur port JTAG commandé à partir du port parallèle d'un PC
    - Le port JTAG est utilisé pour le test des composants sur carte : on parle de boundary scan (IEEE 1149.1)
  - Les FPGA SRAM sont volatiles : ils perdent leur configuration quand le système est éteint
    - Il faut leur associer sur la carte une EEPROM pour charger leur configuration (via JTAG) au démarrage

32

FPGA

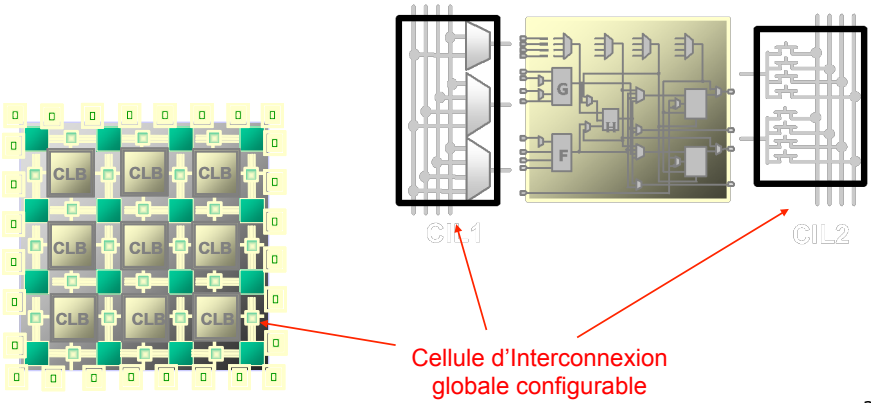
- Interconnexions globales
  - 6 transistors (associés à 6 points mémoires) sont utilisés pour créer des interconnexions de types « Nord-Sud-Est-Ouest »



33

FPGA

- Interconnexions locales



34

Feature	SRAM	Antifuse
Technology node	State-of-the-art	One or more generations behind
Reprogrammable	Yes (in system)	No
Reprogramming speed (inc. erasing)	Fast	---
Volatile (must be programmed on power-up)	Yes	No
Requires external configuration file	Yes	No
Good for prototyping	Yes (very good)	No
Instant-on	No	Yes
IP Security	Acceptable (especially when using bitstream encryption)	Very Good
Size of configuration cell	Large (six transistors)	Very small
Power consumption	Medium	Low

35

FPGA

- Composants enfouis
  - Les FPGA sont fournis avec des « Hard IP », composants câblés et optimisés pour la technologie du FPGA
    - Multiplieurs, blocs RAM, MAC (Multiplieur-Accumulateur pour les applications de traitement du signal), processeur, contrôleur d'horloge…

36

Plan

- Introduction
- SPLD
- CPLD
- FPGA
- **Conclusion**

37

Conclusion

- FPGA versus CPLD
  - CPLD
    - Peu de blocs logiques mais des blocs logiques avec un grand nombre d'entrées/sorties
    - Un seul grand bloc d'interconnexions : peu flexibles
    - Souvent non volatiles et programmables
    - Performances élevées
  - FPGA
    - Plusieurs milliers de blocs logiques avec peu d'entrées/sorties
    - Majoritairement volatiles ou programmables une seule fois
    - Des interconnexions « omniprésentes » dans l'architecture
    - Temps dépendant du routage

38

Conclusion

- Choisir un composant
  - Celui qui s'ajuste le mieux à votre application
    - Coût
    - Performance : fréquence max., temps maximum d'initialisation des bascules, délais horloge vers sortie (tco)…
    - Quantité de logique nécessaire : approximation du nb de portes équivalentes (1000, 10.000 portes?), cellules logiques ou slice…

39

Conclusion

- Option de l'architecture : nb d'horloges, type de contrôle sur les données de sorties, nb de reset, nb de signaux critiques, structures câblées internes (multiplieurs, DCM, blocs RAM, DSP)…
- Consommation
- Boîtiers : réduction de la taille (attention à la compatibilité entre famille)

40

Conclusion

- … SOC Mixtes Programmables
  - PSoC = Programmable System-On-Chip
    - Microcontroleur, timers, UART…
    - Logique programmable
    - Partie analogique : capacités commutés + amplificateurs opérationnels + ADCs + DACs + comparateurs…

41