Électronique numérique

Le language VHDL

Schmitz Thomas

February 27, 2018

Université de Liège

Contact

Schmitz Thomas

mail: T.Schmitz@ulg.ac.be

office: 1.81a

phone: +32 4 366 2706

Outline I

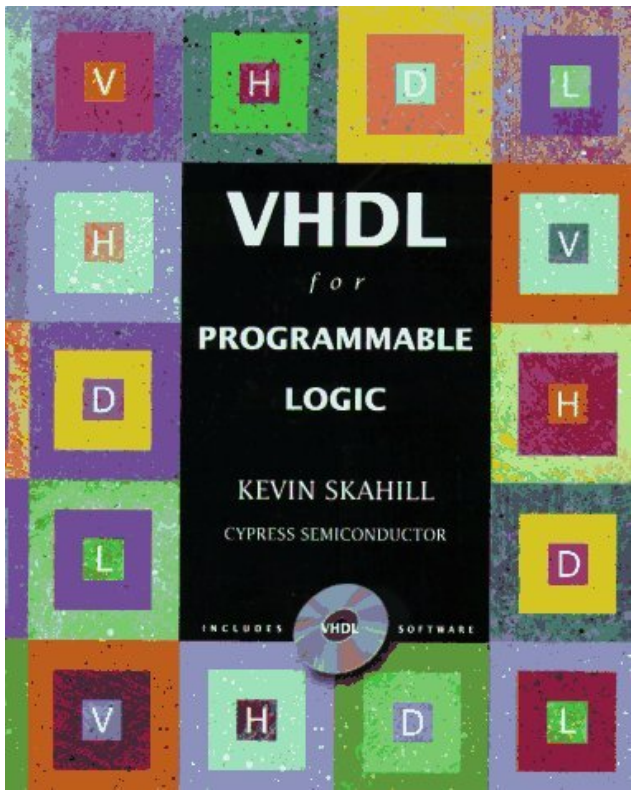
1. Introduction
   - Presentation
   - Présentation des concepts
   - Présentation des outils
2. VHDL bases
   - Exemples introductifs
   - Une différence fondamentale
3. Sequential logic
   - Base
   - Machines d'état
4. Project
5. Syntax
   - Syntaxe de base
   - Déclarations parallèles
   - Déclarations séquentielles
6. Bibliography

Section 1

Introduction

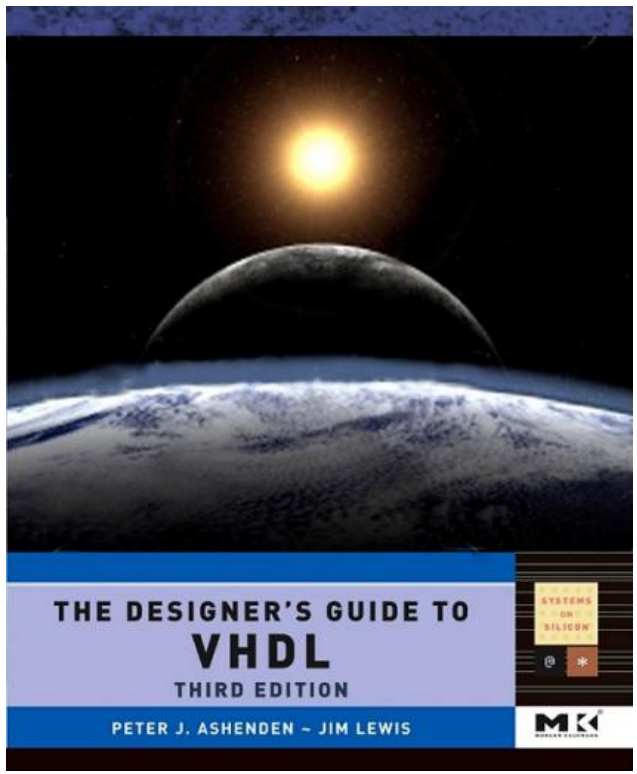
References

VHDL for Programmable Logic [2]



http://www.textfiles.com/bitsavers/pdf/cypress/warp/VHDL_for_Programmable_Logic_Aug95.pdf

The Designer's Guide to VHDL [1]



La logique programmable

Définition

Un circuit logique programmable, est un circuit intégré logique qui peut être reprogrammé après sa fabrication. Il est composé de nombreuses cellules logiques élémentaires librement assemblables.

- Différent d'un processeur,
- Peut représenter n'importe quel circuit digital,
- Peut être reprogrammé un grand nombre de fois.

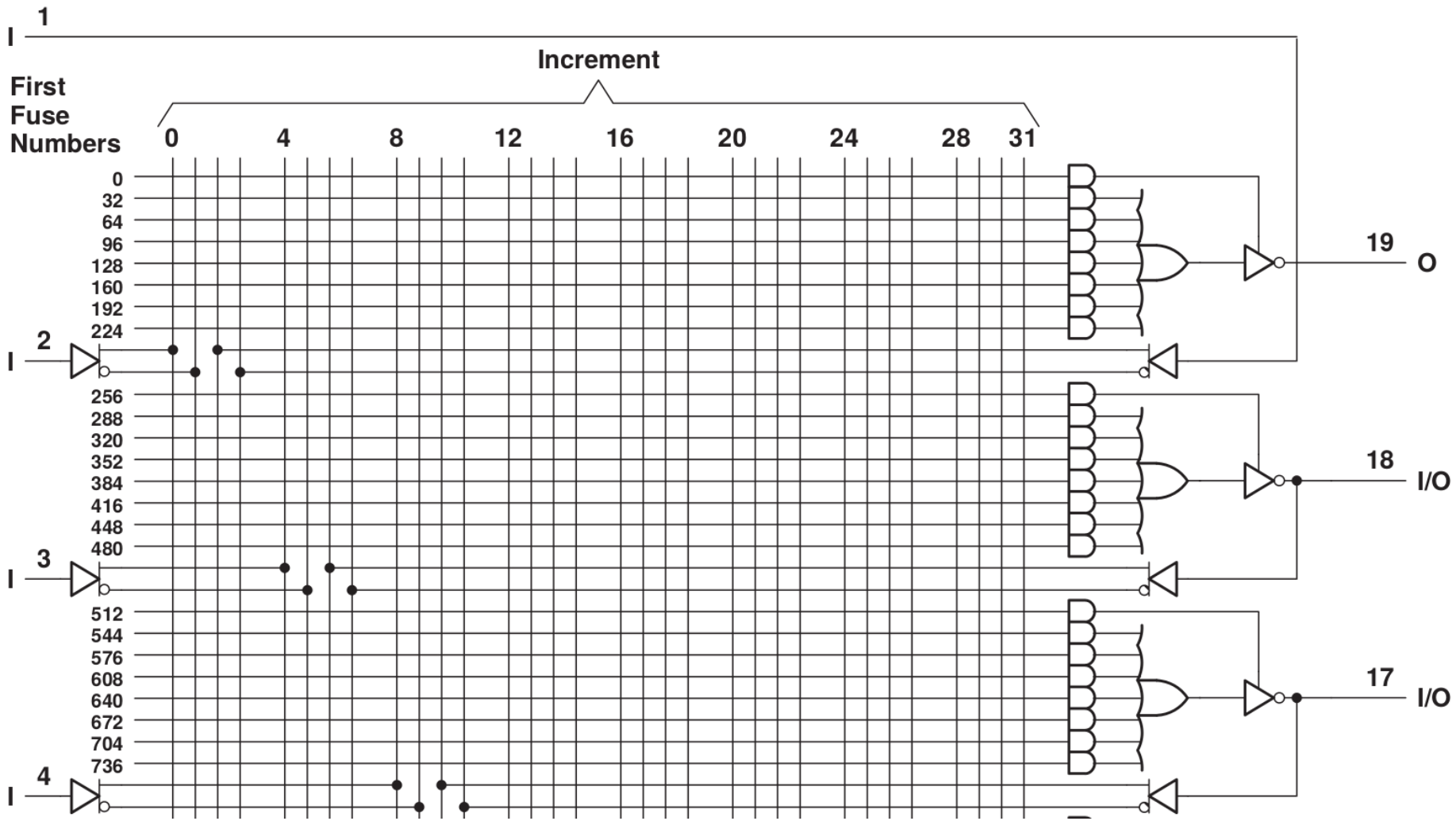
On retrouve:

- PAL
- CPLD
- FPGA
- Dérivés...

La logique programmable

PAL (ex: 16L8)

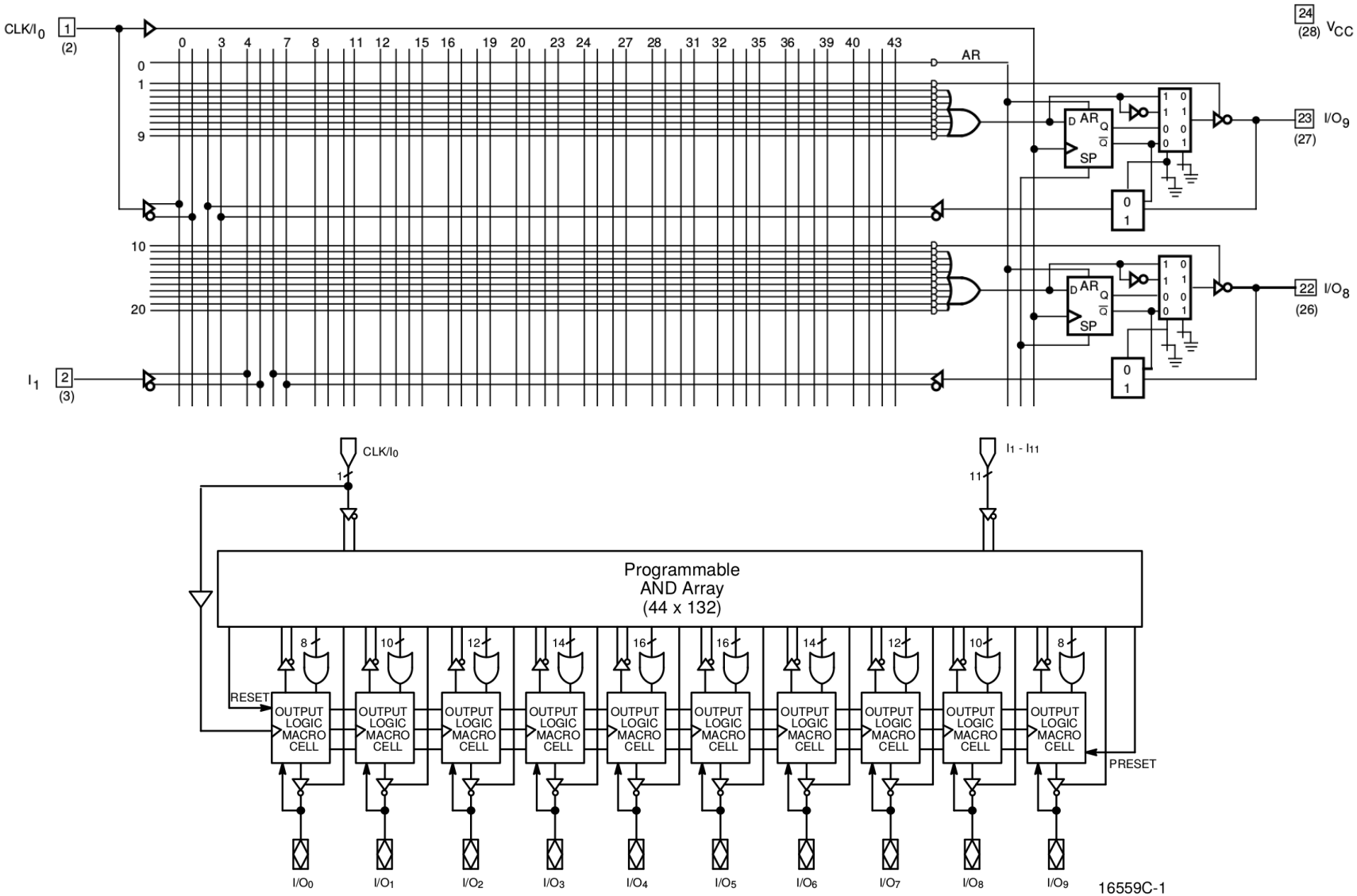
Les composants les plus anciens et les plus simples (réseau de portes assemblées sous la forme d'une grille):



La logique programmable

PLD (ex: 22V10)

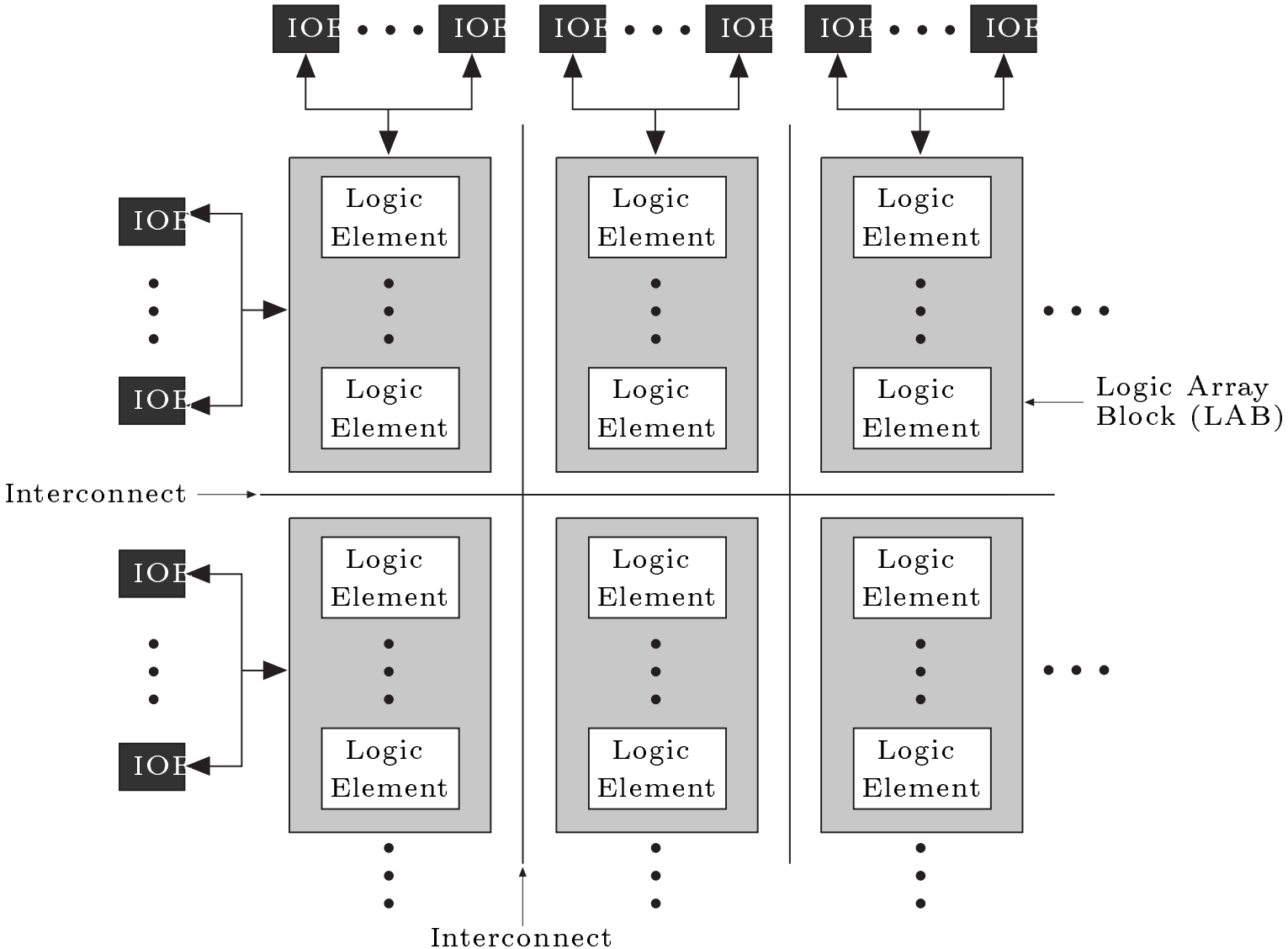
PAL + macrocell (registre et multiplexeur):



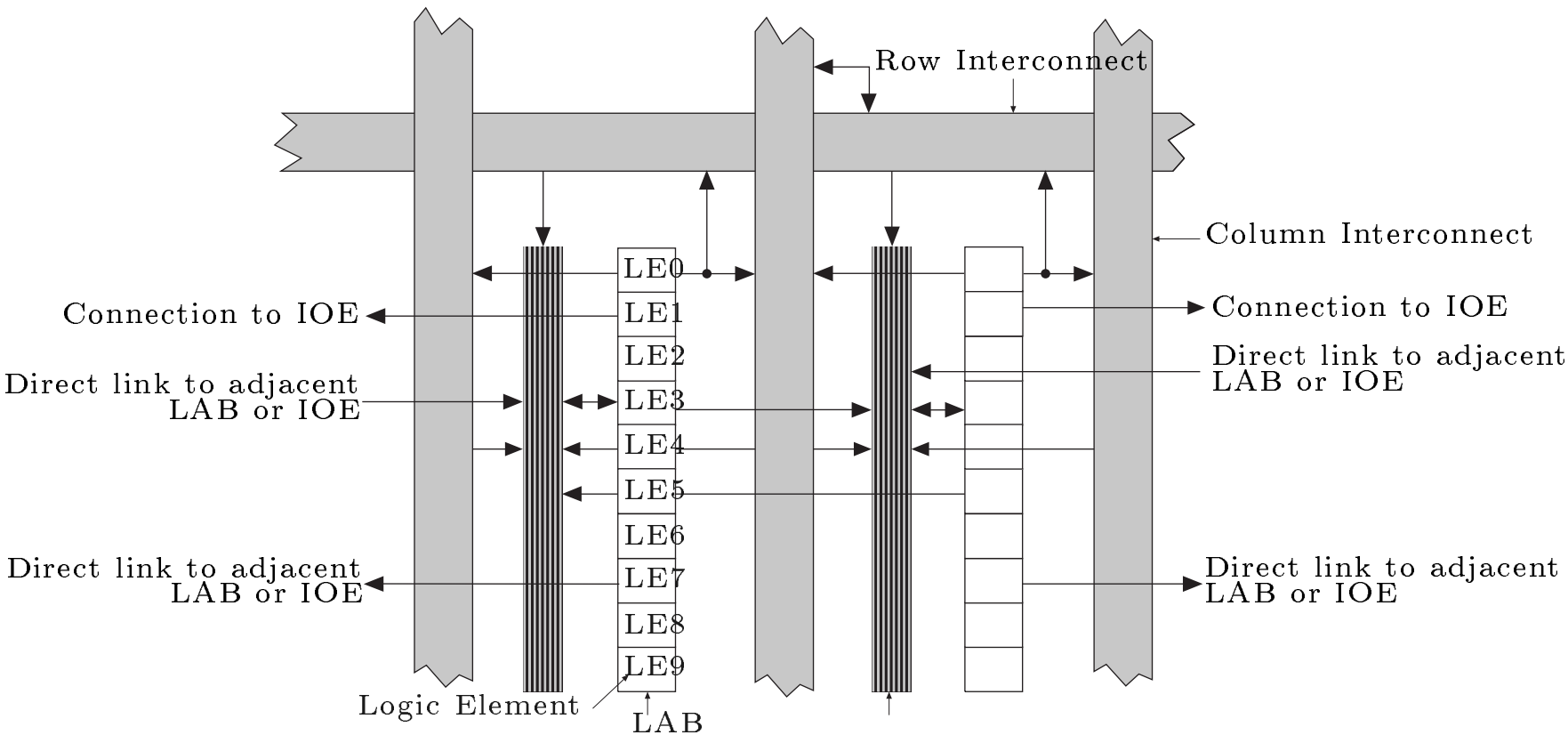
La logique programmable

CPLD

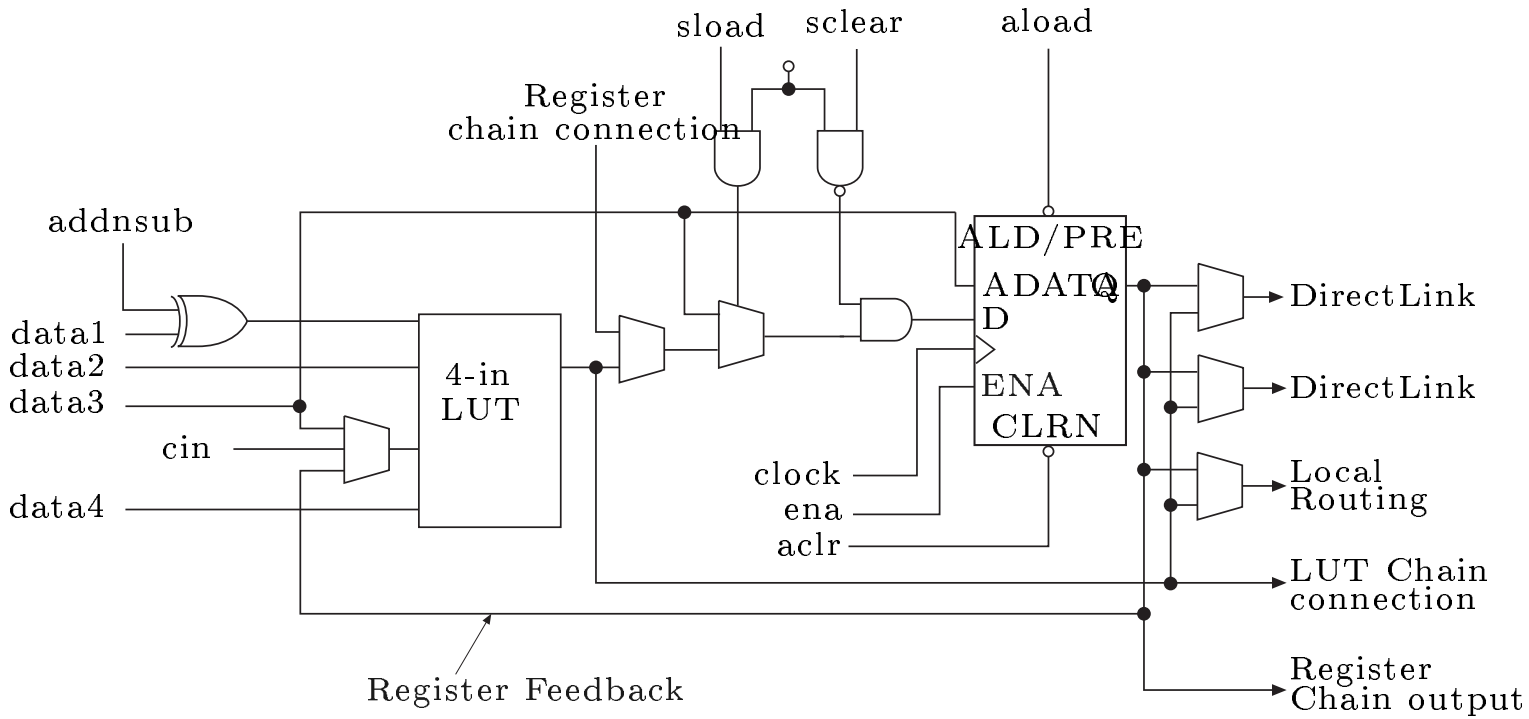
Vue globale de l'architecture (ensemble de PLDs connectées entre elles)



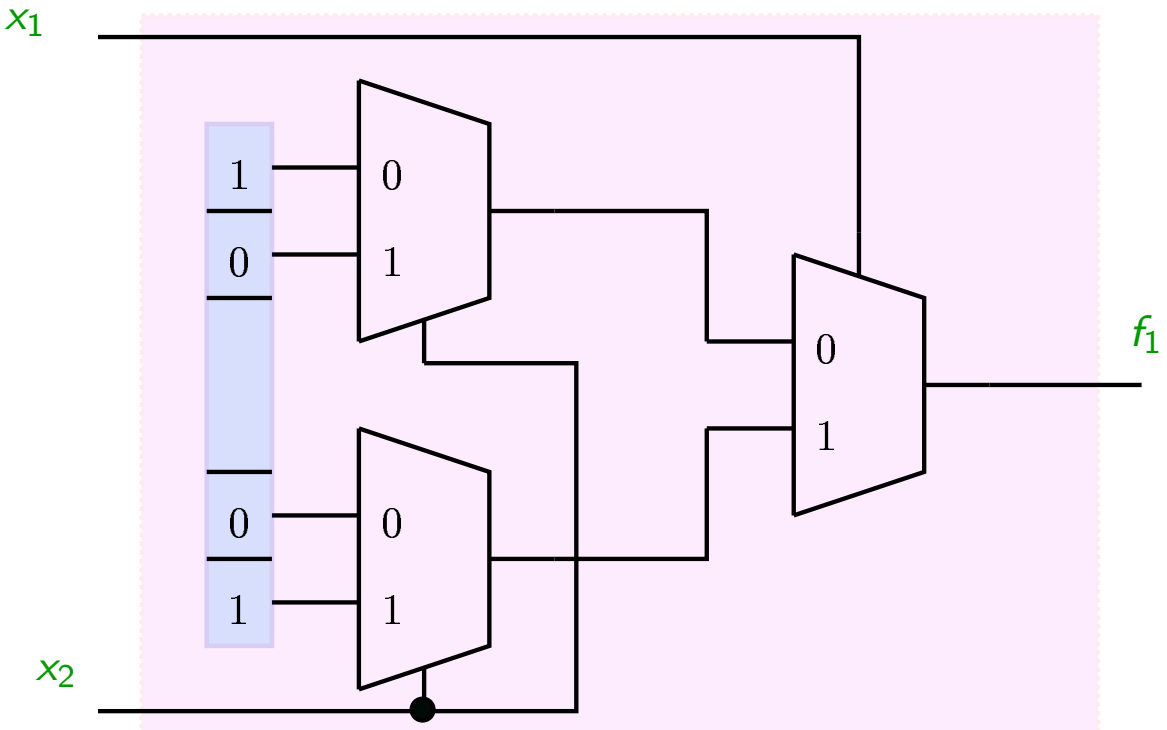
Zoom sur un LAB (Logic Array Block)



Zoom sur un LE (Logic Element)



La LUT à n entrées permet de représenter n'importe quelle fonction booléenne à n variables



x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

$$f_1 = \overline{x_1}.\overline{x_2} + x_1 x_2 \qquad (1)$$

Composant beaucoup plus dense qu'un CPLD (environ 1000 fois plus de LE).

Avantages

- Interconnection des LE plus flexibles,
- Possède des éléments dédiés tels que:
  - Mémoire RAM
  - Gestion des horloges (PLLs, etc)
  - Fonctions DSP (multiplicateurs/additionneurs)
  - Processeurs
  - etc

Utilité?

CPLD:
- Circuits numériques relativement simples,
- Interfaçage entre composants numériques dédiés (Glue logic),
- Conversion de données, etc...

FPGA:
- Simulation de processeurs,
- Tâches massivement parallèles (filtrage, encodage vidéo, etc...)
- Co-processeur,
- Remplacement d'ASIC, etc...

Le VHDL

Wikipedia

VHDL est un langage de description de matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. Son nom complet est VHSIC Hardware Description Language (VHSIC = Very High Speed Integrated Circuit).

De plus, le VHDL:

- peut être simulé,
- peut être traduit en schéma de portes logiques.

Remarquons que:

- En simulation, le code va être exécuté séquentiellement, instruction par instruction,
- Une fois implémenté, le circuit décrit par le code va être conçu grâce à de la logique combinatoire, et/ou, de la logique séquentielle.

Conclusion

La description d'un circuit combinatoire complexe pourra prendre plusieurs instructions à être exécuté en simulation, mais sera réalisé en un temps de propagation une fois implémenté!

Alternatives

Un autre language: le Verilog

- Syntaxe différente (Vérilog plus proche du C, l'autre de l'ADA)
- Difficulté différente
- Même but et résultat

En Europe, on utilise plutôt le VHDL, tandis qu'outre-atlantique, c'est le Verilog.

Référence

Pour une liste objective des différences entre ces deux languages, voir [3]

Alternatives

Une autre philosophie: la description schématique (schematic)

- Très simple d'apprentissage,
- Très bas niveau,
- Très rapide.

En revanche, cette description n'est pas standard, donc pas forcément pérène.

Remarque

Il est toutefois possible de mélanger le VHDL avec la description schématique, ce qui peut dans certain cas être pertinent.

Outils

Outils de base:

- Un éditeur de texte,
- Un simulateur,
- Un analyseur (transformation du code VHDL en éléments logiques de base)
- Un Place & Route (placement et connection des éléments logiques dans le composant).

Beaucoup de logiciels prennent en charge toutes ces fonctionnalités:

- Xillinx ISE
- Altera Quartus
- Lattice ISP Lever
- Altium Designer
- etc...

Section 2

VHDL bases

Subsection 1

Exemples introductifs

Contenu d'un programme VHDL (code source exemple => ici)

Tout programme VHDL doit être composé au minimum d'une paire indissociable entité/architecture.

- Entité: elle décrit les entrées/sorties du composant,
- Architecture: elle décrit le fonctionnement interne du composant.

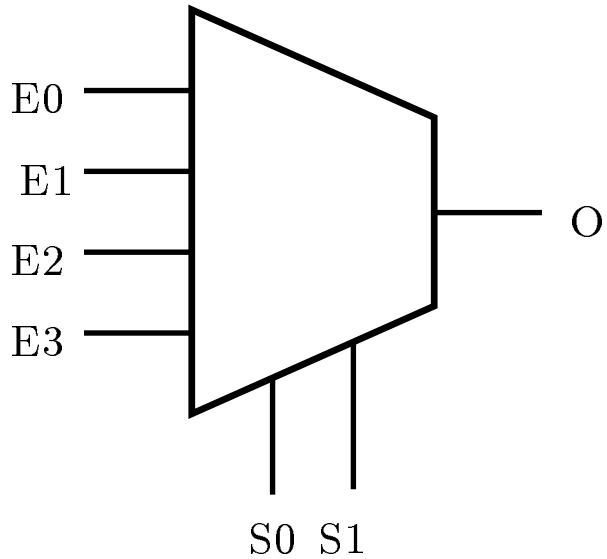
Il a également besoin de bibliothèques définissant les types et les opérations des différents signaux utilisés. Typiquement:

```
1 library ieee ;
2 use ieee.std_logic_1164.all ; -- définition du type bit, bit vector, ...
3 use ieee.numeric_std.all ; -- opérations signées et non signées
4 use ieee.std_logic_arith.all ; -- opérations signées et non signées sur les vecteur
```

Entité

Déclaration

```
1 entity mux41 is port (
2   E0, E1, E2, E3 : in std_logic ;
3   S0, S1         : in std_logic ;
4   O              : out std_logic
5   );
6 end entity mux41;
```



- Déclaration d'une entité `mux41`
- Déclaration des ports d'entrées/sorties de cette entité
- Déclaration du type des entrées/sorties (in, out, ...)

Entité

L'entité correspond à la vue externe du composant.

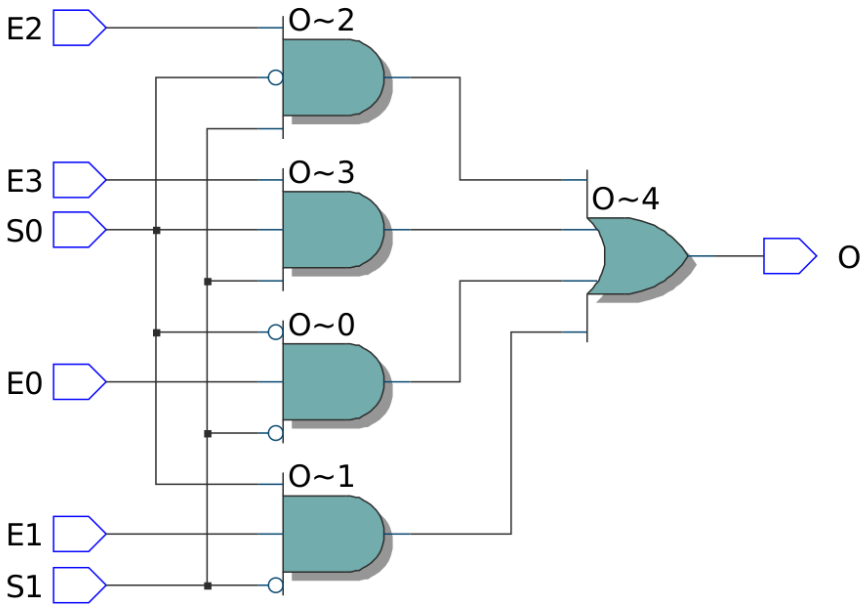
- La déclaration des entrées/sorties (ports) reprend toujours le nom du signal et le mode:

in:	un port d'entrée,
out:	un port de sortie,
inout:	un port bidirectionnel,
buffer:	un port de sortie, dont le signal est aussi utilisé en interne,

Architecture flot de données

Déclaration

```
1 architecture mux41_arch of mux41 is
2 begin
3   O <= ((not S0) and (not S1) and E0) or
4         (S0 and (not S1) and E1) or
5        ((not S0) and S1 and E2) or
6         (S0 and S1 and E3) ;
7 end architecture mux41_arch ;
```



Particularité

Une architecture "flot de données" décrit le fonctionnement de l'entité via des équations booléennes, ou des formes conditionnelles.

Descriptions d'une architecture

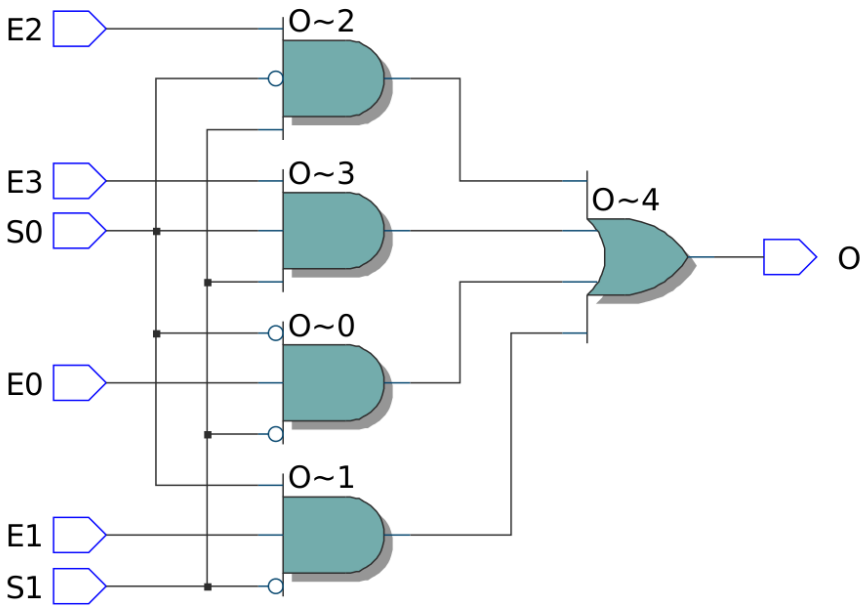
La description par flot de données peut devenir lourde pour des fonctions plus complexes. Il existe deux autres moyens de décrire une architecture :

- Structurelle : tel un schéma, des composants sont instanciés et associés ensembles par des signaux.
- Comportementale : le comportement est décrit sous forme d'instructions séquentielles rassemblées en processus.

Architecture structurelle

Déclaration

```
architecture mux41_arch of mux41 is
  signal  not_S0, not_S1 : std_logic ;
  signal  X : std_logic_vector(3 downto 0) ;
begin
  inv0 : not1 port map ( S0 , not_S0 ) ;
  inv1 : not1 port map ( S1 , not_S1 ) ;
  gate0 : and3
    port map (not_S0, not_S1, E0, X(0)) ;
  gate1 : and3
    port map (    S0, not_S1, E1, X(1)) ;
  gate2 : and3
    port map (not_S0,     S1, E2, X(2)) ;
  gate3 : and3
    port map (    S0,     S1, E3, X(3)) ;
  gate4 : or4
    port map (X(0), X(1), X(2), X(3), O) ;
end architecture mux41_arch ;
```



- Utilisation d'éléments existants (dans un autre fichier ou une librairie),
- Interconnections de ces éléments via les `port map`,

- Ce type d'architecture utilise des éléments existants (dans un autre fichier ou une librairie) et les connecte ensemble. Une instanciation est formée comme suit:

  Un nom: pour identifier l'instanciation,
  Un composant: qui est instancié,
  Des connections: effectuées grâce à l'opération port map, qui indique comment connecter l'élément instancié avec les différents signaux.

Architecture comportementale

Déclaration

```
1  architecture mux41_arch of mux41 is
2  begin
3    mux41: process(E0, E1, E2, E3, S0, S1)
4    begin
5      if (S0 = '0' and S1 = '0') then
6        O <= E0 ;
7      elsif (S0 = '1' and S1 = '0') then
8        O <= E1 ;
9      elsif (S0 = '0' and S1 = '1') then
10       O <= E2 ;
11     else
12       O <= E3 ;
13     end if ;
14   end process mux41 ;
15 end architecture mux41_arch ;
```

- Déclaration d'un process `mux41`,
- et de sa liste de sensibilité,
- Description du processus via une suite d'instructions séquentielles,
- Utilisations d'instructions conditionnelles `if-then-elsif-else`,
- Utilisation de signal assignment (`<=`),

Utilité

Une architecture comportementale décrit la façon dont le système doit fonctionner, via une suite d'instructions séquentielles rassemblées dans un processus.

- Ce type d'architecture nécessite au moins un processus, qui comprend:

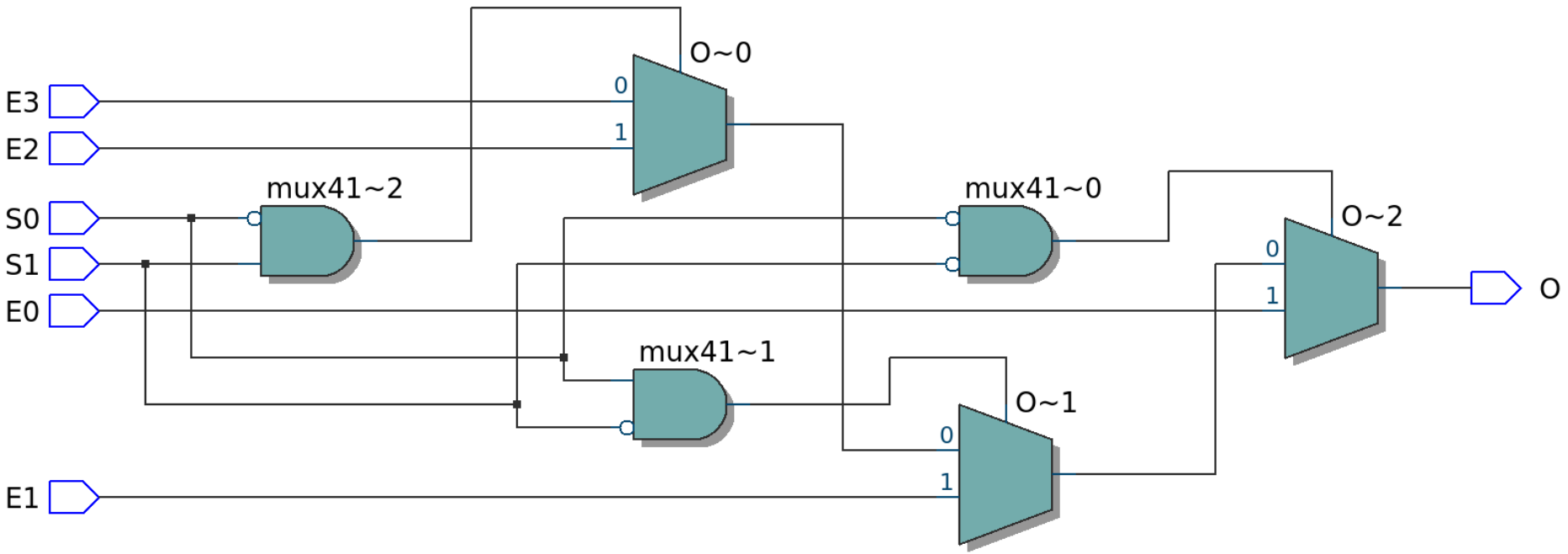
  Nom: permettant d'identifier de manière unique le processus,

  Liste de sensibilité: permettant d'identifier les signaux qui ont un impact sur le processus. Tout changement dans au moins un de ces signaux provoque la réévaluation du processus,

  Liste d'instructions: qui, à la manière d'un programme C sont exécutées de manière séquentielle pour la simulation, ou pour générer le circuit final.

Architecture comportementale

Circuit généré



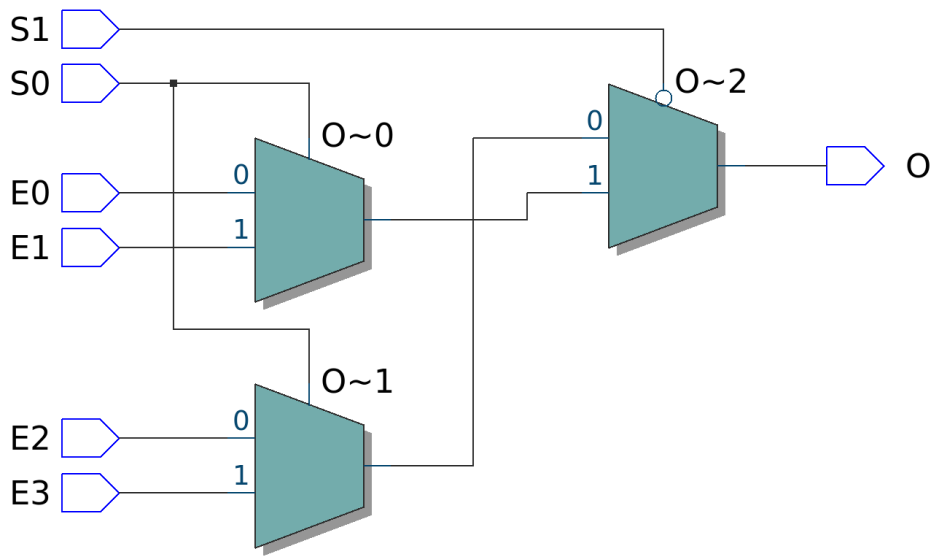
Observations

Le circuit généré est différent de ceux générés pour les architectures "Flot de données" et "Structurelle". Néanmoins, la fonction réalisée est identique. En conclusions, pour une même fonction réalisée, le circuit correspondant généré à partir d'un fichier VHDL pourra différer suivant sa description, le compilateur, le composant cible, etc.

Architecture comportementale

Particularité

```
1  architecture mux41_arch of mux41 is
2  begin
3    mux41: process(E0, E1, E2, E3, S0, S1)
4    begin
5      if (S1 = '0') then
6        if (S0 = '0') then
7          O <= E0 ;
8        else
9          O <= E1 ;
10       end if ;
11     else
12       if (S0 = '0') then
13         O <= E2 ;
14       else
15         O <= E3 ;
16       end if ;
17     end if ;
18   end process mux41 ;
19 end architecture mux41_arch ;
```



- La manière d'écrire un code VHDL influence directement le circuit généré.

Section 2

VHDL bases

Subsection 2

Une différence fondamentale

Simulation vs Synthèse

La simulation peut être décomposée comme suit:

- Analyse du code, avec exécution séquentielle de celui-ci.
- Au temps t, chaque signal se voit éventuellement affecté de transactions programmées au temps $t + \delta$.
- Application des transactions au temps $t + \delta$.

Ainsi, les transactions ne sont jamais effectuées immédiatement, mais sont effectuées après l'analyse complète du code!

Lors de la synthèse, en revanche, tout est exécuté en parallèle. Il peut donc y avoir une différence entre le circuit qu'on imagine, et la simulation de celui-ci. Il convient donc d'avoir les règles ci-dessus en tête, afin de bien faire comprendre au compilateur ce que vous voulez!

Simulation vs Synthèse

Il est important de noter que tout code écrit pour la synthèse pourra être simulé, alors que tout code écrit pour la simulation n'est pas forcément synthétisable.

Exemples

Illustration d'un problème possible

```
1  entity my_and8 is port(
2    a : in std_logic_vector( 7 downto 0 ) ;
3    x : buffer std_logic
4    );
5  end my_and8 ;
6
7  architecture wont_work of my_and8 is
8  begin
9  anding: process(a)
0
1    begin
2      x <= '1' ;
3      for i in 7 downto 0 loop
4        x <= a(i) and x ;
5      end loop ;
6
7    end process ;
8  end architecture wont_work ;
```

```
1
2
3
4
5
6  architecture will_work of my_and8 is
7  begin
8  anding: process(a)
9    variable tmp : bit ;
10   begin
11     tmp := '1' ;
12     for i in 7 downto 0 loop
13       tmp := a(i) and tmp ;
14     end loop ;
15     x <= tmp ;
16   end process ;
17 end architecture will_work ;
```

Section 3

Sequential logic

Subsection 1

Base

Exemples introductifs

Flip flop sensible au flanc (Flip flop D)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity dff_logic is port(
4    d, clk : in std_logic ;
5    q      : out std_logic
6    );
7  end entity dff_logic ;
8
9  architecture dff_logic_arch of dff_logic is
10 begin
11   process( clk )
12   begin
13     if( clk'event and clk = '1' )
14     then
15       q <= d ;
16     end if ;
17   end process ;
18 end architecture dff_logic_arch ;
```

- Liste de sensibilité sur `clk`,
- Détection d'un flanc montant
  `if( clk'event and clk = '1' )`
- Absence de `else`: le flip flop conserve sa valeur, puisqu'aucune valeur n'est spécifiée dans ce cas.

Remarque

La plupart des simulateurs ne supportent pas un `else` après un
`if( clk'event and clk = '1' )`.
En effet, cette construction serait ambiguë.
(L'évènement ne durant qu'un court instant...)

Exemples introductifs

Flip flop sensible à la valeur (Latch transparent)

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  entity ff_logic is port(
4    d, clk : in std_logic;
5    q      : out std_logic
6    );
7  end entity ff_logic ;
8
9  architecture ff_logic_arch of ff_logic is
10 begin
11   process( clk , d )
12   begin
13     if( clk = '1' ) then
14       q <= d ;
15     end if ;
16   end process ;
17 end architecture ff_logic_arch ;
```

- Liste de sensibilité sur `clk` et `d`,
- Condition sur la valeur de `clk`, et non sur le flanc,
- Absence de `else` pour la mémoire.

Exemples introductifs

Flip flop avec reset asynchrone

```
1  use ieee.std_logic_1164.all;
2  entity rff_logic is port(
3    d, clk, rst : in std_logic;
4    q           : out std_logic
5    );
6  end rff_logic;
7
8  architecture rff_logic_arch of rff_logic is
9  begin
10   process ( clk , rst )
11   begin
12     if( rst = '1' ) then
13       q <= '0';
14     elsif rising_edge( clk ) then
15       q <= d;
16     end if;
17   end process;
18 end architecture rff_logic_arch;
```

- Fonction `rising_edge()`,
- Reset dans la liste de sensibilité,
- Reset asynchrone,
- Si reset synchrone, le mettre après le if du `rising_edge()`

Exemples introductifs

Le `wait`

Il est possible d'utiliser une clause **`wait`**:

```
1 architecture dff_logic_arch2 of dff_logic is
2 begin
3   process begin
4     wait until ( clk = '1' ) ;
5       q <= d ;
6     end process ;
7 end architecture dff_logic_arch2 ;
```

Cela fonctionnera, à condition que le **`wait until()`** soit la première instruction du process, rendant impossible l'ajout d'un reset asynchrone.

Subsection 2

Machines d'état

Introduction

Jusqu'à présent, la conception d'une machine d'état demandait les étapes suivantes:

- Définition d'un diagramme d'état,
- Transformation des états en codes binaires,
- Tables de Karnaugh,
- Détermination des équations booléennes.

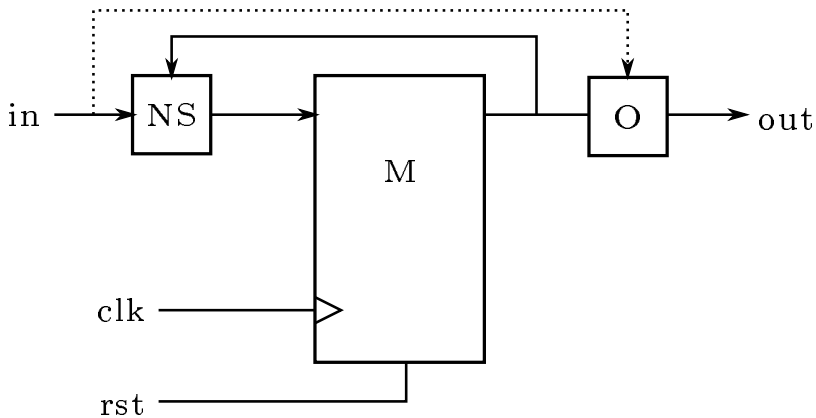
En VHDL, il est bien entendu possible de faire pareil, mais il y a plus simple:

- Définition du diagramme d'état,
- Code.

Différents styles

Le codage d'une machine d'état peut se faire de plusieurs façons, selon vos habitudes, facilités, etc.
De manière générale, une machine d'état se présente comme suit:

- Une fonction permettant de calculer l'état suivant (`NS`),
- Une fonction permettant de calculer les sorties (`O`),
- Une mémoire (`M`), permettant de retenir l'état courant.



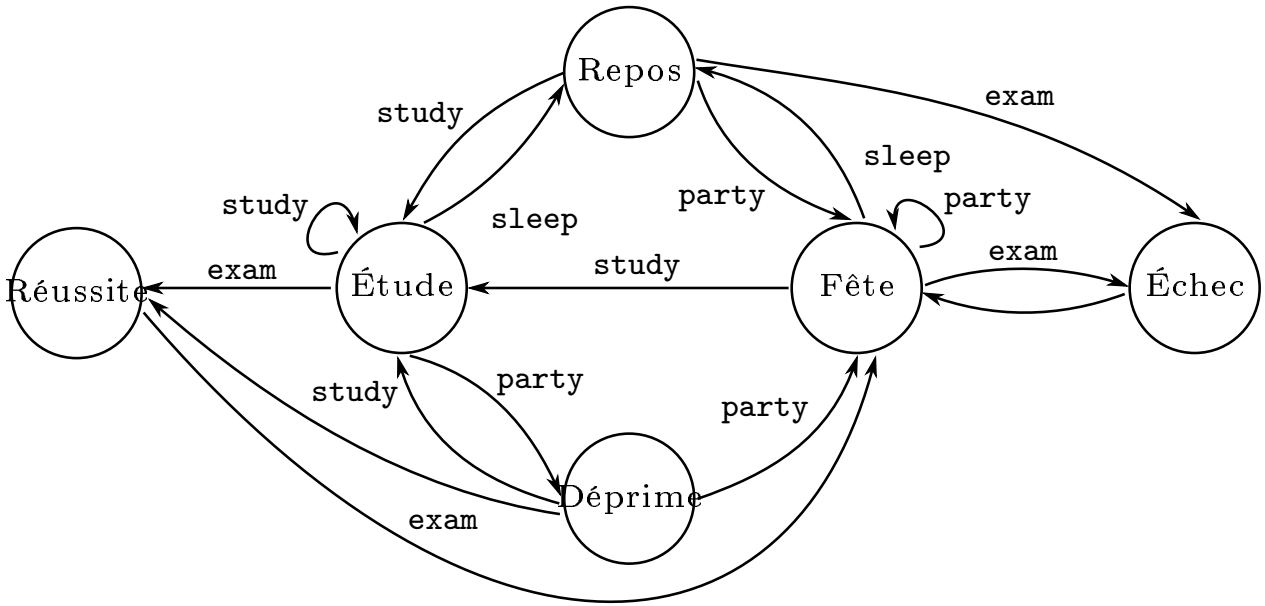
1 process: (NS + M + O),

2 process: M + (NS + O); O + (NS + M),

3 process: NS + M + O.

Exemple

Modélisation d'un étudiant



- États étudiant : Repos, Étude, Réussite, Déprime, Fête, Échec
- Modélisation des signaux d'entrées :
  - Professeur : donne un examen.
  - Besoin de l'étudiant : besoin d'étudier, de dormir ou de faire la fête.
  - Horloge du système : une activité par jour.

Exemple

3 process

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity student is port(
5    professor : in std_logic ;
6    need      : in std_logic_vector( 1 downto 0 ) ;
7    day       : in std_logic ;
8    stud_out  : out std_logic_vector( 2 downto 0 )
9    );
0  end entity student ;
1
2  architecture student_arch of student is
3    type student_state is (Repos, Fete, Etude, Deprime,
4                           Reussite, Echec) ;
5
6    constant Exam : std_logic := '1';
7    constant Sleep : std_logic_vector( 1 downto 0 ) := "00" ;
8    constant Study : std_logic_vector( 1 downto 0 ) := "01" ;
9    constant Party : std_logic_vector( 1 downto 0 ) := "10" ;
0
1    signal present_state : student_state := Repos ;
2    signal next_state    : student_state := Repos ;
3
4    begin
5
6    -- First process (compute next state, combinatorial)
7    next_state_calc : process( professor, need, present_state )
8    begin
9      case present_state is
```

```
28       when Repos =>
29         next_state <= Repos ;
30         if professor = Exam then
31           next_state <= Echec ;
32         else
33           if need = Study then
34             next_state <= Etude ;
35           elsif need = Party then
36             next_state <= Fete ;
37           end if ;
38         end if ;
39
40       when Fete =>
41         next_state <= Repos ;
42         if professor = Exam then
43           next_state <= Echec ;
44         else
45           if need = Study then
46             next_state <= Etude ;
47           elsif need = Party then
48             next_state <= Fete ;
49           end if ;
50         end if ;
```

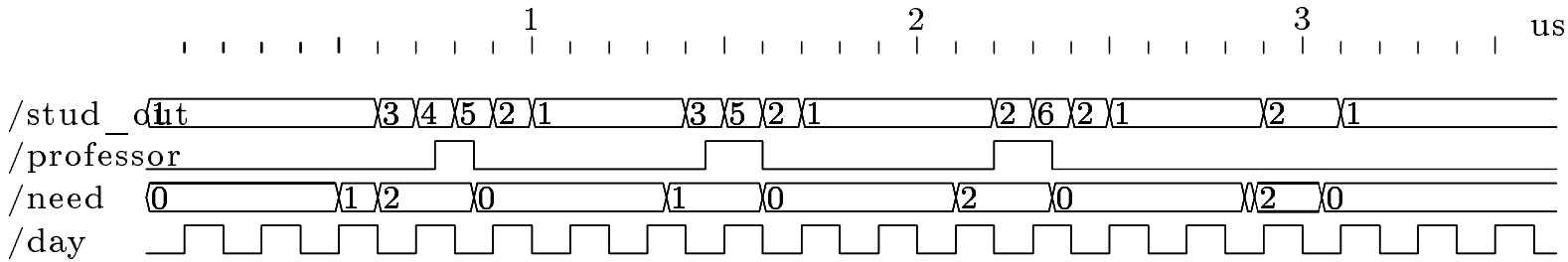
Exemple

3 process

```
      when Etude =>
        next_state <= Repos ;
        if professor = Exam then
          next_state <= Reussite ;
        else
          if need = Study then
            next_state <= Etude ;
          elsif need = Party then
            next_state <= Deprime ;
          end if ;
        end if ;

      when Deprime =>
        next_state <= Deprime ;
        if professor = Exam then
          next_state <= Reussite ;
        else
          if need = Study then
            next_state <= Etude ;
          elsif need = Party then
            next_state <= Fete ;
          end if ;
        end if ;

      when Echec =>
        next_state <= Fete ;

      when Reussite =>
        next_state <= Fete ;
        end case ;
    end process next_state_calc ;

    -- Second process
    -- (update current state, sequential)
    update: process( day )
    begin
      if (day'event) then
        present_state <= next_state ;
      end if;
    end process update ;

    -- Third process (implicit)
    -- (compute output, combinatorial)
    with present_state select
    stud_out <= "001" when Repos,
                "010" when Fete,
                "011" when Etude,
                "100" when Deprime,
                "101" when Reussite,
                "110" when Echec,
                "111" when others ;

end architecture student_arch ;
```

Exemple

Simulation



Légende:

stud_out:

- 1 - Repos
- 2 - Fête
- 3 - Étude
- 4 - Déprime
- 5 - Réussite
- 6 - Échec

need:

- 0 - Sleep
- 1 - Study
- 2 - Party

Remarques:

- La modification de l'état courant est déclenchée sur une transition du signal `day`.
- Deux processus dans ce cas (le troisième est implicite), mais le calcul des sorties aurait pu être intégré dans un processus sensible au `present_state`.
- Dernier scénario: l'étudiant change d'envie trop rapidement et seule la dernière est prise en compte.

Conclusions

Autres styles:

- Modification des sorties dans le `case`,
- Processus `update` intégré dans le processus de calcul de l'état suivant (liste de sensibilité changée)
- Code exemple pour filtre FIR en VHDL

Conclusion

A vous de trouver votre style de code!

Section 4

Project

Objectifs et instructions

Objectif

L'objectif de ce projet est de vous familiariser avec:

- La création d'un circuit électronique simple,
- le VHDL.

Ce projet consistera donc en un réalisation d'un système fonctionnel, soit sur breadboard, soit, mieux, sur carte pré-trouée ou un circuit imprimé.

Échéances

Trois étapes à ce projet:

25 Mars: Envoi par mail d'un bref descriptif de votre projet, avec

- Description de maximum 5 lignes du projet,
- Description de maximum 5 lignes de la structure du code, avec le nombre de ressources utilisées,
- Matériel nécessaire en plus de la carte de développement.

Cette description me permettra de voir si votre projet est réalisable. Sinon, je vous enverrai quelques suggestions.

1 Mai : Envoi par mail d'un rapport pour votre projet, avec

- Le code VHDL,
- Des simulations démontrant le bon fonctionnement du code,
- Les schémas électriques,
- Toute autres informations que vous jugerez nécessaires.

Session : Démonstration de la réalisation en live

- Date fixée au début de la session,
- Présentation orale d'environ 10-15 minutes.

Attention

Titre du fichier envoyé : ELEN0040Projet(ou PreProjet)_JDaniels_WLawson_BLabel

Conseil - Réflexion

Avant de vous lancer :

- Réfléchissez au nombre d'entrée/sortie et au nombre de bit de mémoire nécessaires.
- N'oubliez jamais que vous ne programmez pas, vous décrivez un circuit électronique !

Conseil - Réalisation

- Breadboard: simple, mais attention aux court-circuits et faux contacts.
- Carte pré-trouée : plus propre, permet d'apprendre à souder.
- PCB : prise en main de logiciels de CAO, réalisation d'un circuit.

Exemple de composants utiles:

- Leds,
- Afficheurs 7 segments,
- Boutons poussoirs (Attention aux rebonds!),
- Potentiomètres,
- Décodeurs,
- etc...

Attention

Vérifiez TOUJOURS les tensions d'alimentation!

Exemples

Utile:
- Système domotique (gestion du chauffage, d'éclairage, etc...),
- Système de gestion de la glycémie d'un patient,
- Robot suiveur,
- Cadena crypté,

Jeux:
- Mastermind,
- Tireur,
- Simon,
- Pong,
- Tour de Hanoï,

Bloqué ?

- Regardez les codes d'exemples sur le Mux41 et Fir filter
- Apprendre le VHDL sur youtube
- Regardez des codes VHDL sur d'autres sites.

Bon travail!

Section 5

Syntax

Exécution séquentielle et parallèle

Fondamental

- A l'extérieur d'un processus, les instructions/processus s'exécutent en parallèle
- A l'intérieur d'un processus, les instructions s'exécutent séquentiellement lors de la simulation, ou afin de décrire le circuit sous-jacent

- Ainsi, certaines instructions sont plus spécifiques à une description parallèle, telles que les instructions de choix:
  Si une variable vaut x, alors une autre variable vaudra y
  Ce genre d'instruction peut être exécutée en parallèle puisque on peut vérifier cette phrase à tout moment.
- En revanche, certaines instructions sont plus spécifiques à une description séquentielle:
  Une variable vaut x, puis si une autre variable change, alors elle vaudra y
  Ne peut pas s'exécuter en permanence puisqu'elle attend des changements.
- Les instructions plus spécifiques parallèle peuvent se trouver aussi dans les processus.

Subsection 1

Syntaxe de base

Syntaxe de base

Fonction	Exemple ou règle
Commentaires	-- Commentaire sur une ligne entity reg4 is -- Déclaration de l'entité
Identificateurs	• Caractères alphabétiques, chiffres ou underscores • Doit démarrer avec un caractère alphabétique • Ne peut se terminer avec un underscore • Ne peut contenir deux underscores successifs • Insensible à la casse (sauf pour les énumérations de type littéral);
Mots réservés	• and, or, nor, etc... • if, else, elsif, for, etc... • entity, architecture, etc...

Fonction	Exemple ou règle
Nombres	Real literal: 0, 23, 2E+4, etc... Integer literal: 23.1, 42.7, 2.9E-3, etc... Base literal: 2#1101#, 8#15#, 10#13#, 16#D#, etc... Underscore: 2_867_918, 2#0010_1100_1101#
Caractères et chaînes	'A' -- caractère "A string" -- Chaîne de caractères "A string in a ""A string"". " B"0100011" -- Chaîne binaire X"9F6D3" -- Chaîne hexadécimale 5B#11# = B"00011" -- Chaîne à taille fixe

Notation EBNF

Définition

Métalanguage permettant de décrire avec précision la syntaxe d'un langage comme le VHDL.

Fonction	Exemple ou règle
Définition: ⇐	Affectation <= nom_variable := expression;
Optionnel: [ ]	Fonction <= nom_fonction [(arguments)];
0 ou plus: {}	Déclaration_de_process <= process is {définition_process} begin {déclarations} end process;
Répétition: ...	Déclaration_case <= case expression is déclaration {...} end case;
liste: {"delimiteur"...}	Liste_identificateurs <= identificateur{, ...}
Ou: |	mode <= in | out | inout

Constantes

Syntaxe

constant identificateur {, ...} : sous-type [:= expression]

Exemples:

```
1 constant number_of_bytes : integer := 4;
2 constant number_of_bits : integer := 8*number_of_bytes;
3 constant size_limit, count_limit : integer := 255;
4 constant prop_delay : time := 3ns;
```

Usage:

- Similaire aux déclarations préprocesseur en `C`.
- Pratique pour rendre le code lisible.

Variables

Syntaxe

variable identificateur {, ...} : sous−type [:= expression]

variable_à_assigner <= nom := expression;

Exemples:

```
variable number_of_bytes : integer := 4;
variable size_limit, count_limit : integer := 255;
variable prop_delay : time := 3ns;
...
number_of_bytes := 2;
number_of_bits := number_of_bytes * 8;
```

Usage:

- Similaire aux variables `C`, à l'intérieur d'un processus.
- L'affectation d'une valeur à une variable prend effet immédiatement.
- Pas de signification physique.
- Ne peut être déclarée et accessible que dans un process.

Variables

Attention!

Différent d'une affectation de signal! La variable est modifiée tout de suite, le signal entraîne une modification future!

Attention!

Une variable n'a pas de signification physique, et n'est en général pas synthétisable. Les variables sont souvent utilisées comme indices de boucles.

Type

Syntaxe

type identificateur is type_definition;
subtype identificateur is subtype_definition;

type_definition <= range expression (to | downto) expression;
subtype_definition <= type [range expression (to | downto) expression]

Exemples:

```
1 type players is range 0 to 4;
2 type days is 0 to 7;
3 subtype bit_index is integer range 31 downto 0;
```

Usage:

- Contraindre des constantes, variables, etc, à certaines valeurs.
- Très utile pour la lisibilité du code.

Remarque

- On peut déclarer des types avec des entiers ou des flottants.
- La valeur par défaut est la première déclarée (plus petite si `to`, plus grande si `downto`).

Type

Enumérations

Syntaxe

type identificateur is type_definition;

type_definition <= ((identificateur | caractère) {, ...}) ;

Exemples:

```
1 type player is (Player1, Player2, Player3, CPU)
2 type days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun)
3 ...
4 variable today : days;
5 today := Mon;
```

Usage:

- Facilité de lecture du code.
- Optimisation.

Remarque

- Il s'agit d'un type mais avec des noms, donc ce qui s'applique aux types s'applique aux énumérations également.

Types

Prédéfinis

Booléens: `type boolean is (false, true);`

Bits: `type bits is ('0', '1');`

std_ulogic: ou std_logic

```
type std_ulogic is ( 'U',    -- Uninitialized
                     'X',    -- Forcing Unknown
                     '0',    -- Forcing 0
                     '1',    -- Forcing 1
                     'Z',    -- High Impedance
                     'W',    -- Weak  Unknown
                     'L',    -- Weak  0
                     'H',    -- Weak  1
                     '-');   -- Don't care
```

Caractères: `type character is (nul, soh, ' ', '*', '1', 'A', '...') ;`

Remarque

Pour le type **std_logic**, il convient d'inclure une librairie:

```
library ieee;
use ieee.std_logic_1164.all;
```

Syntaxe

array_def <= array (discrete_range {, ...}) of elements
discrete_range <= discrete | expression (to | downto) expression

Exemples:

```
1 type word is array (0 to 31) of std_logic;
2  ...
3 type controller_state is ( initial , idle , active , error );
4 type state_counts is array (controller_state range idle to error) of natural;
5  ...
6 variable buffer : word;
7  ...
8 buffer(0) := '0';
```

Types

Tableaux - multidimensionnels

Exemple:

```
type table16x8 is array(0 to 15, 0 to 7) of std_logic;
constant 7seg: table16x8 := (
  "00111111", -- 0
  "00000110", -- 1
  "01011011", -- 2
  "01001111", -- 3
  "01100110", -- 4
  "01101101", -- 5
  "01111101", -- 6
  "00000111", -- 7
  "01111111", -- 8
  "01101111", -- 9
  "01110111", -- A
  "01111100", -- B
  "00111001", -- C
  "01011110", -- D
  "01111001", -- E
  "01110001"); --F
```

Attributs

Un attribut fournit une information sur un élément:

- Entité,
- Architecture,
- Type,
- Tableau,
- ...

Les attributs disponibles dépendent de l'élément sur lequel ils portent.

Attributs

de tableaux

Syntaxe:

`A'left(N)`	borne gauche
`A'right(N)`	borne droite
`A'low(N)`	valeur minimale
`A'high(N)`	valeur maximale
`A'range(N)`	variation maximale d'index
`A'downrange(N)`	variation maximale d'index
`A'length(N)`	taille du tableau
`A'ascending`	vrai si les index de A sont croissantes
`A'element`	sous type des éléments

Remarque:

- *N* représente la dimension étudiée.

Attributs

de signaux

Syntaxe:

`S'delayed(T)`	signal S décalé de T
`S'stable(T)`	vrai si S a été stable pendant T
`S'quiet(T)`	vrai si S a été stable depuis T
`S'transaction`	vrai si il y a eu une transaction sur S
`S'event`	vrai si il y a un évènement sur S au pas de simulation présent
`S'active`	vrai si il y a une transaction sur S au pas de simulation présent
`S'last_event`	temps depuis le dernier évènement
`S'last_active`	temps depuis la dernière transaction
`S'last_value`	dernière valeur avant le dernier évènement

Remarque:

- Beaucoup de ces attributs sont utilisés pour des vérifications de timing.

Opérateurs

Syntaxe

```
and, or, nand, nor, xor, xnor, not      -- logiques
+, -, &                                 -- additifs
*, /, mod, rem                          -- multiplicatifs
abs, **                                 -- divers
<=, :=                                  -- assignation
=>                                      -- association
sll, srl, sla, sra, rol, ror            -- décalages
=, /=, <, <=, >, >=                     -- relations
```

Exemples:

```
1  X(1) <=   (a(3) and not(s(1)) and not (s(0))
2         or (b(3) and not(s(1)) and s(0))
3         or (c(3) and s(1) and not(s(0)))
4         or (d(3) and s(1) and s(0));
5  bit_vector <= bit_vector sll 2;
```

Remarques:

- Il n'y a pas de précédence des opérateurs en VHDL (pas de priorité par défaut).
- L'usage des parenthèses est donc obligatoire pour éviter des erreurs de compilation.

Subsection 2

Déclarations parallèles

Condition with-select-when

Syntaxe

```
with select_expression select
  nom <= {expression when value,}
          expression when value;
```

Exemple:

```
1  with S select
2    O <= E0 when "00",
3    O <= E1 when "01",
4    O <= E2 when "10",
5    O <= E3 when others;
```

Usage:

- Affectation d'un signal, soumise aux valeurs d'un signal de sélection.

Remarque:

- Modélise typiquement un multiplexeur.
- Les différents choix doivent être mutuellement exclusifs, et tous représentés!

Condition when-else

Syntaxe

```
nom <= {expression when condition else}
       expression;
```

Exemple:

```
1  result <= (a - b) when (mode = substract) else (a + b);
```

Usage:

- Assignation d'un signal, soumise à différentes conditions.

Attention

Les différentes conditions ne sont pas forcément mutuellement exclusives. Ainsi, si plusieurs conditions sont validées, la première rencontrée aura priorité. Dès lors, faites attention à l'écriture de votre code, qui pourrait être implémenté différemment que prévu dans ces cas particuliers (voir par exemple encodeur prioritaire)!
Voir [2] 167 - 172.

Subsection 3

Déclarations séquentielles

Condition if-then-else

Syntaxe

```
if condition then
  sequence_of_statements
{ elsif condition then
  sequence_of_statements}
[ else
  sequence_of_statements]
end if;
```

Exemple:

```
1      if (S0 = '0' and S1 = '0') then O <= E0;
2    elsif (S0 = '1' and S1 = '0') then O <= E1;
3    elsif (S0 = '0' and S1 = '1') then O <= E2;
4    elsif (S0 = '1' and S1 = '1') then O <= E3;
5    else O <= '-';
6    end if;
```

Usage:

- Choix simple ou série de choix simples

Condition case-when

Syntaxe

```
case expression is
  {when condition => sequence_of_statements}
end case;

condition <= identifier | expression | discrete_range | others
discrete_range <= expression (to|downto) expression
```

Exemples:

```
1 case S is
2   when "00" => O <= E0;
3   when "01" => O <= E1;
4   when "10" => O <= E2;
5   when "11" => O <= E3;
6   when others => O <= '-';
7 end case;
```

```
1 case day is
2   when Mon to Wed =>
3     O <= '1';
4   when Sun downto Fri =>
5     O <= '0';
6   when others =>
7     O <= 'Z';
8 end case;
```

Usage:

- Choix multiples sur un signal.

Boucles

Syntaxe

```
[loop_label:] loop
  {sequence_of_statements
  }
end loop [loop_label];
```

```
[loop_label:] while
    condition loop
  {
    sequence_of_statements
  }
end loop [loop_label];
```

```
[loop_label:] for
    identificateur in
    discrete_range loop
  {sequence_of_statements
  }
end loop [loop_label];
```

```
discrete_range <= expression (to|downto) expression

exit [when condition];
next [when condition];
```

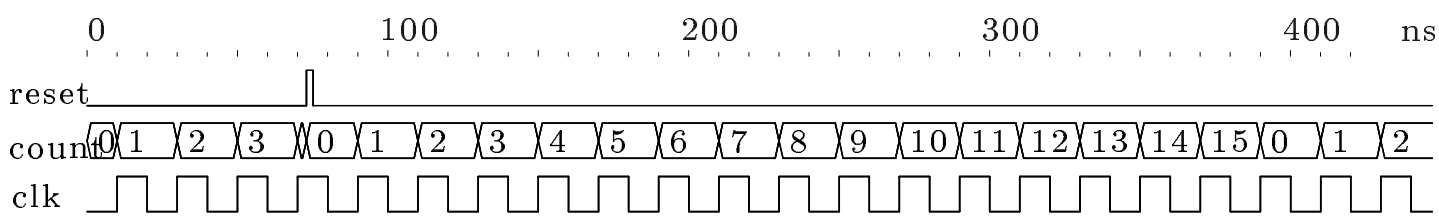
Usage

- Les différentes boucles peuvent toute effectuer la même chose,
- mais un choix judicieux du type de boucle permet un code plus concis et lisible.
- `exit` permet de quitter la boucle.
- `next` permet de passer à l'itération suivante, sans exécuter les instructions qui suivent.

Compteur modulo 16 avec reset

```
Library IEEE;
Use     IEEE.std_logic_1164.all;
entity counter is
  port (clk, reset:    in std_logic;
        count:         out integer);
end entity counter;
```

```
architecture behavior of counter is
begin
  incrementer: process is
    variable count_value : integer := 0;
    begin
    count <= count_value;
      loop
        loop
          wait until clk = '1' or reset = '1';
          exit when reset = '1';
          count_value := (count_value + 1) mod 16;
          count <= count_value;
        end loop;
        count_value := 0;
        count <= count_value;
        wait until reset = '0';
      end loop;
    end process incrementer;
end architecture behavior;
```



Décodeur 3-8

```
Library IEEE;
Use     IEEE.std_logic_1164.all;
entity dec38 is
  port( E0, E1, E2:   in std_logic;
        S:            out std_logic_vector(7 downto 0));
end entity dec38;

architecture behavior of dec38 is
begin
  process (E0, E1, E2)
  variable N : integer;
  begin
    N := 0;
    if E0 = '1' then N := N+1; end if;
    if E1 = '1' then N := N+2; end if;
    if E2 = '1' then N := N+4; end if;

    S <= "00000000";
    for I in 0 to 7 loop
      if I = N then
        S(I) <= '1';
      end if;
    end loop;
  end process;
end behavior;
```

Bibliographie I

P.J. Ashenden.
The Designer's Guide to VHDL.
Systems on Silicon Series. Morgan Kaufmann, 2008.

K. Skahill and J. Legenhausen.
VHDL for programmable logic.
Electrical engineering/digital desing. Addison-Wesley, 1996.

D.J. Smith.
VHDL and verilog compared and contrasted-plus modeled example written in VHDL, verilog and c.
In Design Automation Conference Proceedings 1996, 33rd, pages 771–776. IEEE, 1996.