Langage Assemblage et Jeu d'Instructions

Cours (9x2h00) :

- Frédéric Mallet - fmallet@unice.fr

TP (9x2h00 - 1 groupe) :

- Frédéric Mallet - fmallet@unice.fr

http://deptinfo.unice.fr/~fmallet/laji/

Structure du cours

- Performances : évolution et comparaison
- Codage de l’information
- Fonctions logiques et éléments mémoires
- Systèmes à microprocesseurs
- La famille Intel - 80x86
- Conception d'architectures numériques – VHDL
- Éléments avancés (parallélisme, mémoire, ...)

Conception d'architectures matérielles

- 1974 : premier processeur CISC (Von Neumann)
- 1981 : premier DSP, Architecture Harvard
- 1980s : premiers processeurs RISC
- Depuis lors:
  - L'équivalence "matériel-logiciel" est admise
  - Pour élaborer des versions compétitives de **compilateurs**, de **systèmes d'exploitation**, de bases de données et même d'applications, le **programmeur** doit accroître ses connaissances de **l'organisation de l'ordinateur**.

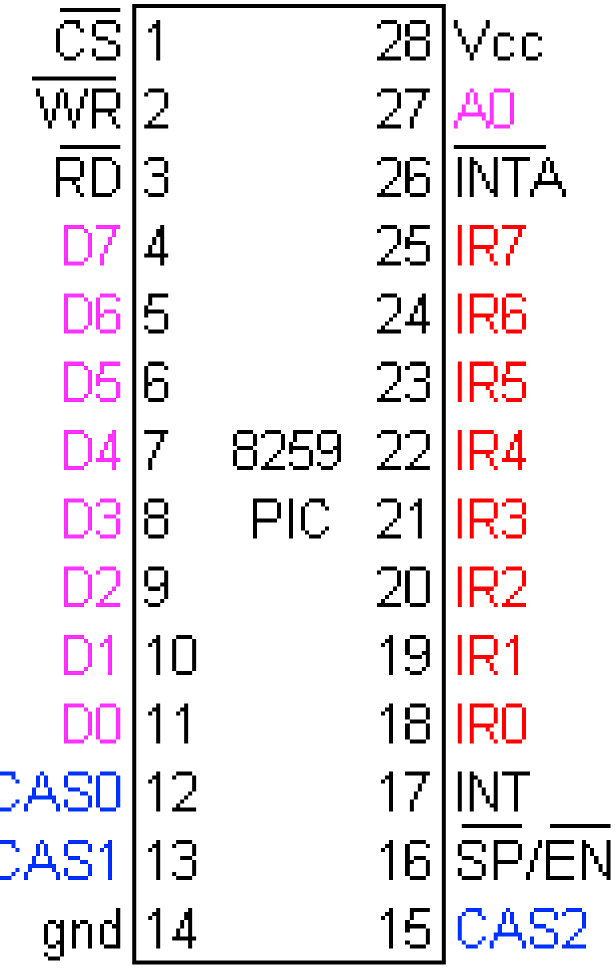
Signaux logiques et numériques

- Les circuits numériques fonctionnent avec deux **niveaux électriques** stables :
  - Niveau haut (H) et niveau base (L)
- En logique (algèbre de Boole)
  - On introduit les **valeurs logiques** : faux (0) et vrai(1)
  - A partir de là, on construit des fonctions logiques
- On associe niveaux électriques et valeurs logiques
  - H ↔ 1, L ↔ 0 (**convention positive**)
  - H ↔ 0, L ↔ 1 (**convention négative**)
- Après le choix d'une convention, on parle de **niveau logique**
  - E.g. convention positive : 0 ou bas, 1 ou haut

Signal logique et activité

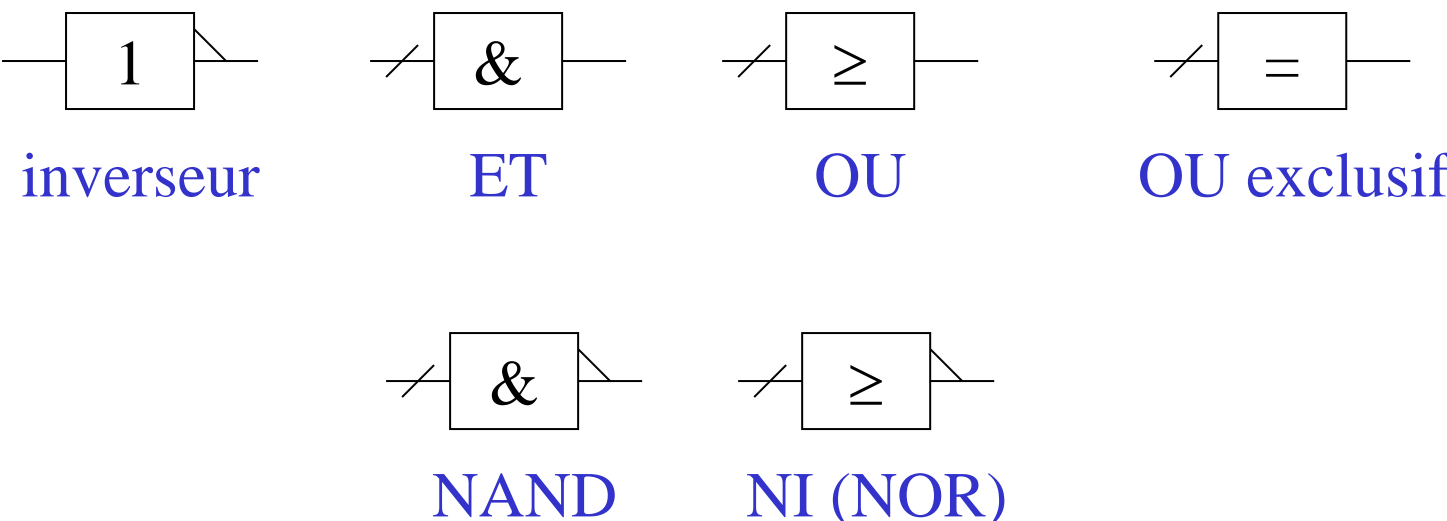
- Un signal logique a trois attributs :
  - Un **NOM** qui rappelle sa fonction dans le système
  - Une **VALEUR** logique (0 ou 1)
  - Un niveau **ACTIF** ou **INACTIF**, par référence au niveau logique correspondant
- On parle
  - D'activité au niveau HAUT
  - Et d'activité au niveau BAS (notée par un rond ou un triangle)

L'exemple du 8259



- CS (chip select) est actif au niveau bas
  - Le composant ne travaille que lorsque qu'il est sélectionné
  - C'est-à-dire lorsque CS est actif : **niveau logique bas**
- WR (Write) et RD (Read) aussi
  - Séquence d'interruption :
    - IR*n* devient actif (**niveau haut**)
    - Émission de INT (**niveau haut**)
    - Réception de INTA (**niveau bas**)
    - Transmet le vecteur d'interruption (Bus D)

Représentation IEEE/ANSI



- / indique n entrée/sorties
- ◺ et parfois remplacé par ○
- Parfois, les deux notations sont mélangées

VHDL - introduction

- Langage conçu pour décrire le **comportement de systèmes numériques**
  - De simples portes, à un processeur complet
  - Permet la **spécification**, la **conception**, l'analyse (par **simulation**) et la **synthèse** de systèmes
  - Spécification : construction de **bancs de tests**
    - Ensemble de stimuli + réponse attendue
    - Les bancs de test sont développés en même temps que le circuit
- Vocation du langage pour la **documentation**
- Deux constructions essentielles à la description de circuits numériques
  - Hiérarchie et **concurrence**

VHDL - historique

- VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- 1980s : IBM, Texas Instruments, Intermetrics : DoD
- IEEE 1076 – 1987 : Institute of Electrical and Electronics Engineers, révision 93.
- IEEE 1174 : Standard Logic Package
  - Définition d'un type avec 9 valeurs logiques
- 1076.3 : réalisation physique de ces valeurs logiques et opérations arithmétiques pour la synthèse
- 1076.4 : annotations temporelles pour la simulation
  - Pour concurrencer le modèle VITAL de Verilog-HDL

Une méthode de conception

- **Concevoir à un niveau supérieur** que les portes logiques (RTL, ou plus haut)
- **Les portes logiques sont générées** automatiquement
  - N'exige pas une connaissance poussée du silicium
  - Plus faciles d'explorer plus de choix
  - Facilite la réutilisation de modules
- **Le modèle est validé par simulation**
  - Sémantique de simulation de VHDL
  - Les temps de simulation peuvent être importants pour les gros systèmes
  - Il faut mesurer la **couverture** de la simulation : pas une vérification formelle !

Abstraction du matériel et entités

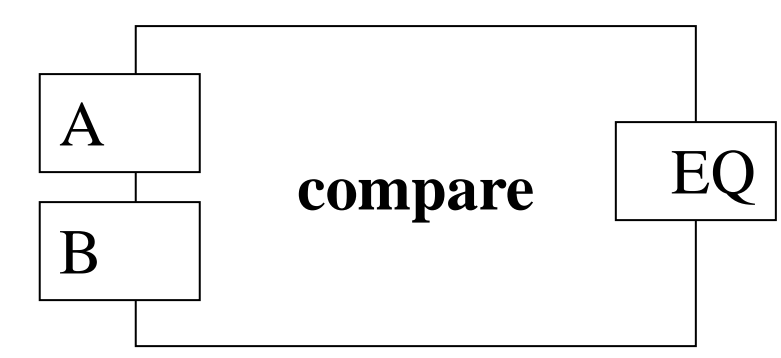
- Un modèle est défini par
  - 1 **vue externe** (ENTITY) qui définit l'interface de communication avec les autres parties du système
  - 1 ou +s **vues internes** (ARCHITECTURE) associées
- On peut par exemple :
  - Définir une vue interne très haut niveau (spécification) pour construire des **jeux de tests** pour la simulation
  - Puis **raffiner certaines parties** pour obtenir une description synthétisable de façon incrémentale. Les jeux de tests sont réutilisés et complétés au fur et à mesure.

Un comparateur - ENTITY

- **Entrées :** 2 données A et B sur 8 bits à comparer
- **Sortie :** 1 bit, '1' si A=B, '0' sinon
- ENTITY décrit les ports (entrées/sorties)

```
ENTITY compare IS
  PORT (A, B : IN bit_vector(0 TO 7);
        EQ : OUT bit);
END compare;
```

- `bit_vector` : tableau de bits
- Case insensitive



Port = nom + type + mode

- Un **port** est un **signal** d'interface
- Il possède un **nom**, un **type** et un **mode**
- Il peut avoir une valeur initiale
  - Cette valeur n'est en général pas exploitée pour la synthèse
  - E.g. **PORT** (o : **OUT** bit := '1');
- L'ordre d'apparition est libre
  - Mais cet ordre est exploité par toute entité utilisatrice (vue interne, bloc supérieur, …)
  - Il est possible d'associer les ports dans un ordre différent, mais mieux vaut y penser avant

5 modes (direction du port)

- **IN** : entrée, port mono-directionnel
  - Sa valeur est seulement utilisable en lecture (vu de l'intérieur de l'entité)
- **OUT** : sortie, port mono-directionnel
  - Impossible de lire la valeur, seulement écriture
- **INOUT** : entrée/sortie (bidirectionnel)
  - En lecture et en écriture
- **BUFFER** : entrée/sortie
  - Une seule source possible ! Y préférer le mode OUT
- **LINKAGE** : entrée/sortie, port lié à un signal

Typage fort !

- TYPE = signal physique

Bit	'1', '0'
Bit_vector	Tableau de Bits
Boolean	True, False
Time	300 ns, 900 ps
Integer	-5, 10, 0
Real	1.0, -1.0E-5
Character	'a','b','2','$'

1 bit EQU <= '1';

n bits OUT <= "101011";

Pas pour la synthèse !
Pas de conversion implicite
Opérateurs de conversion

Types énumérés et types entiers

- **Les types énumérés**
  - **TYPE** bit **IS** ('0', '1') ;   // prédéfini
  - **TYPE** boolean **IS** (false, true);   // prédéfini
  - **TYPE** state **IS** (running, ready, blocked); // utilisateur
- **Les types entiers** (codés sur 32 bits par défaut)
  - **TYPE** index **IS RANGE** 0 **TO** 1023
  - **SUBTYPE** index_10 **IS** integer **RANGE** 0 **TO** 1023
    - Sera codé seulement sur 10 bits
  - Deux sous-types prédéfinis : NATURAL et POSITIVE
    - **SUBTYPE** natural **IS** integer **RANGE** 0 **TO** integer'**HIGH**;
    - **SUBTYPE** positive **IS** integer **RANGE** 1 **TO** integer'**HIGH**;

Types physiques

- Types pour représenter un temps (TIME), une longueur, une tension ou un courant
- TYPE time IS RANGE 0 TO 1e20
- Unités
  - **femtoseconde** : `fs`
  - **picoseconde** : `ps = 1000*fs`
  - **nanoseconde** : `ns = 1000*ps`
  - **microseconde** : `us = 1000*ns`
  - **milliseconde** : `ms = 1000*us`
  - **seconde** : `sec = 1000*ms`
  - **minute** : `min = 60*sec`

Un comparateur - ARCHITECTURE

- L'**ARCHITECTURE** propose une réalisation

```
-- une implémentation concurrente
ARCHITECTURE compare1 of compare IS
BEGIN
   EQU <= '1' WHEN (A=B) ELSE '0' ;
END compare1;
```

- Une seule équation concurrente
  - <= est l'**affectation concurrente** (pas séquentielle !)
  - = est l'opérateur de comparaison (syntaxe *à-la*-ADA)
  - A, B et EQU sont des signaux
- -- marque les commentaires

Unité de conception

- L'ENTITY décrit l'interface physique du composant
  - Elle contient toute l'information nécessaire pour **connecter** plusieurs entités les unes avec les autres
  - PORT = nom + type + mode (**IN**, **OUT**, **INOUT**)
- L'ARCHITECTURE décrit le comportement
  - Par exemple, un ensemble d'instructions concurrentes
- **ENTITY + ARCHITECTURE = unité de conception**
  - Au moins une ARCHITECTURE pour chaque ENTITY
  - +s ARCHITECTUREs à différents niveaux de description
  - **Raffinements successifs pour arriver à un niveau synthétisable**

5 unités de conception (/compilation)

- **ENTITY, ARCHITECTURE**
  - Au minimum une ENTITY et une ARCHITECTURE
- PACKAGE, PACKAGE BODY
  - Zone de mémorisation commune à plusieurs unités de conception (fonctions, procédures, constantes, composants, types et sous-types, signaux globaux)
- CONFIGURATION
  - Liste d'associations entre une ARCHITECTURE et une ENTITY
  - Quelle ARCHITECTURE est **associée** à quelle ENTITY
  - Optionnelle !

Bibliothèque

- Les unités de compilation sont stockées dans une bibliothèque
  - WORK est la bibliothèque par défaut
- On dispose de bibliothèques prédéfinies dans le langage (clause `use`)
  - La clause `library` permet de rendre visible une bibliothèque (STD et WORK sont visibles par défaut)
    - `Library ieee ;`
  - `use ieee.std_logic_1164.all ;`
    - Permet d'utiliser une unité de compilation depuis une bibliothèque VISIBLE
  - `std_logic` donne plus de flexibilité que le type `bit`
    - 9 niveaux logiques (force des signaux)

Trois niveaux d'abstraction possibles

Haut niveau

- **Comportemental**
  - Ce qui se place d'un point de vue temporel
  - Ignore la technologie, pas synthétisable
- **Flot de données**
  - Description combinatoire (ensemble de fonctions booléennes) : $S = f(E)$
  - Ensemble de registres : $Q+ = g(Q, E)$
  - Plus haut niveau accepté par un synthétiseur
- **Structurel**
  - Connexion de 'morceaux' plus petits
  - Diagramme bloc

Bas niveau

Niveau comportemental

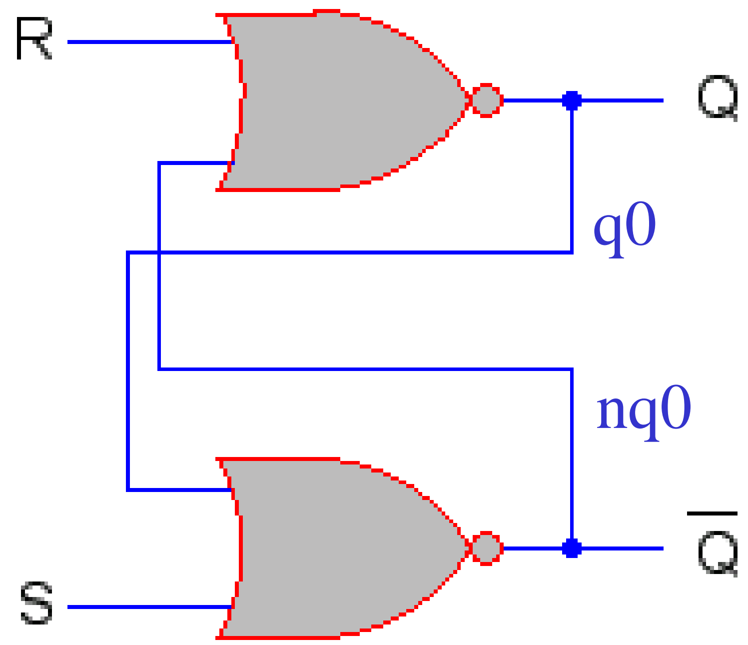
- Machines à états, description d’un algorithme, diagrammes temporels
- Certaines instructions concurrentes permettent d’exprimer des **propriétés temporelles**
  - `A <= B after 10 ns;`
- Cela permet la spécification précise de comportement
- Cette information est **ignorée par les synthétiseurs** qui utilisent la technologie
  - E.g. Un registre prend 4 ns pour basculer

Niveau flot de données

- Très spécifique sur le placement des registres
  - Pas d'instruction VHDL pour les registres
  - Doit utiliser une bibliothèque de registres adaptée à la technologie
- Description générale de la partie combinatoire
  - Le synthétiseur génère les portes logiques
- Niveau RTL (Register Transfer Logic)

Exemple - Latch SR asynchrone

S(et)	R(eset)	Q^+
0	0	Q
0	1	0
1	0	1
1	1	Non utilisé



- Complètement combinatoire
  - **La rétroaction et le délai réalisent la mémorisation**
  - Set : fixe la sortie à 1, Reset : fixe la sortie à 0

Le latch asynchrone – flot de données

```
ARCHITECTURE flot_sr OF latch IS
  SIGNAL q0 : bit := '0';     -- Signaux temporaires avec
  SIGNAL nq0 : bit := '1';    -- valeur initiale
BEGIN
   q0  <= r NOR nq0;
   nq0 <= s NOR q0;

   nq  <= nq0;
   q   <= q0;
END flot_sr;
```

```
ENTITY latch IS
   PORT (s,r  : IN bit;
         q,nq : OUT bit);
END latch;
```

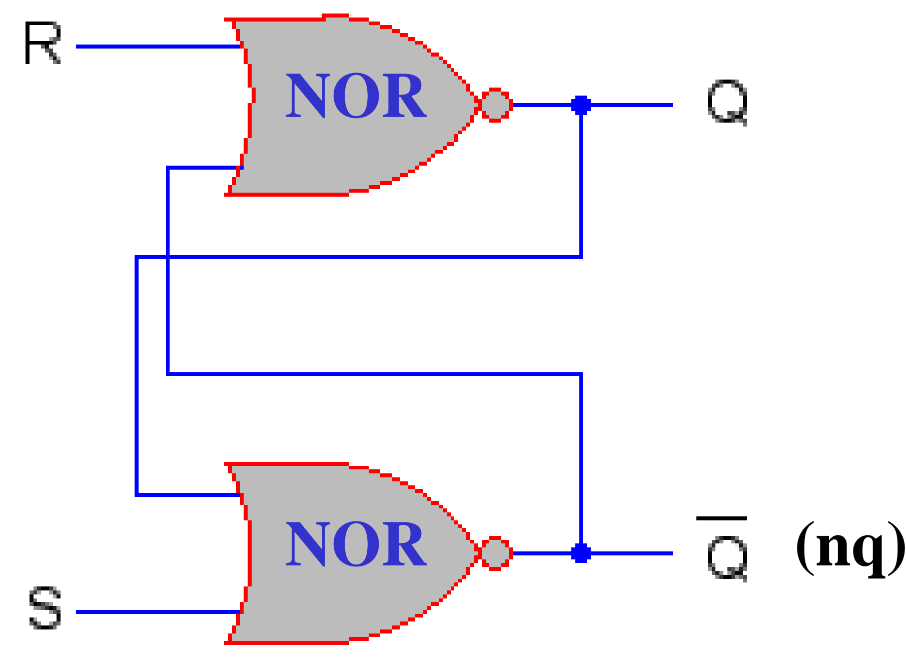
Affectation concurrente <=

- L'ordre des instructions n'a pas d'importance
- L'affectation concurrente
  - **N'est pas un transfert de valeur** (affectation séquentielle)
  - **Décrit une relation** entre un ensemble de portes logiques calculées à partir d'une expression (e.g. r NOR nq0) et un signal logique
- À tout instant
  - La valeur logique de `q0` est égale à la valeur logique de `r NOR nq0` (équipotentielle)
  - La sémantique de simulation par micro-pas garantie cette propriété

Niveau structurel

- Du niveau transistor au niveau diagramme bloc
  - Normalement pas utilisé au niveau transistor !
  - Approche descendante
- Au niveau des portes logiques : **netlist**

```
ENTITY porte_nor IS
   PORT (A,B : IN bit;
         O : OUT bit);
END porte_nor;
```



Le latch asynchrone - structurel

```
ARCHITECTURE struct_sr OF latch IS
  COMPONENT porte_nor IS
    PORT (a,b : IN bit; o : OUT bit);
  END COMPONENT;
  SIGNAL q0  : bit := ‘0’;
  SIGNAL nq0 : bit := ‘1’;
BEGIN
  nor1 : porte_nor PORT MAP(r, nq0, q0);
  nor2 : porte_nor PORT MAP(s, q0, nq0);
  q <= q0;
  nq <= nq0;
END struct_sr;
```

Déclaration d'un composant

```
COMPONENT porte_nor IS
  PORT (a,b : IN bit; o : OUT bit);
END COMPONENT;
```

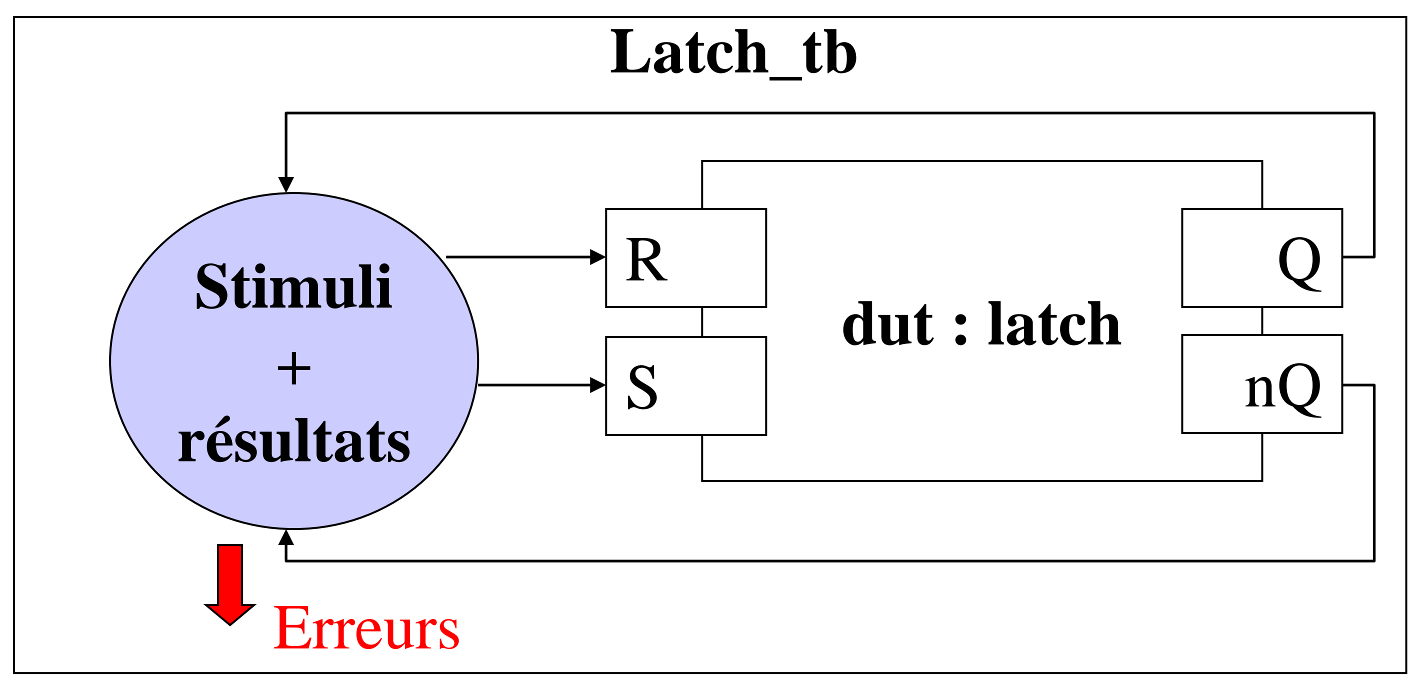
- L'architecture déclare qu'elle va utiliser **un ou +s composants** qu'elle appelle `porte_nor` et qui doivent implémenter cette interface
  - Si une entité `porte_nor` existe, un lien est effectué (on ne présuppose pas encore de la vue interne correspondante)
  - L'entité liée doit avoir la même interface
    - Une configuration permet de changer l'ordre des ports, mais pas les modes ou les types

Instanciation de composant

```
nor1 : porte_nor PORT MAP(r, nq0, q0);
nor2 : porte_nor PORT MAP(s, q0, nq0);
```

- **On déclare deux instances** du composant `porte_nor`
  - On associe des signaux à chacun des ports de l'instance
    - Les ports sont pris dans l'ordre de la déclaration (`s` est associé à `a`, `nq0` est associé à `B`, `q0` est associé à `o`)
    - Le compilateur vérifie que les modes sont respectés
    - Il est interdit de mettre des constantes littérales
  - On ne sait toujours pas quelle vue interne va être utilisée
    - Ici, il n'y a pas d'ambiguïté, une seule vue interne existe, sinon le compilateur en choisit une (ou cf. CONFIGURATION)

Jeu de test : *test bench*



- Un jeu de test est une ou +s unités de conception qui appliquent des stimuli sur les entrées du (**Design Under Test**) et observent les sorties
- On lève les erreurs éventuelles : mot clé **assert**

Jeu de test pour le *latch*

```
ENTITY latch_tb IS -- ni entrée, ni sortie
END latch_tb; -- peut avoir des paramètres
ARCHITECTURE latch_tb1 OF latch_tb IS
  COMPONENT latch IS
    PORT (s,r : IN bit; q,nq : OUT bit);
  END COMPONENT;
  SIGNAL si, ri, qi, nqi : bit;
  -- Configuration (choisit l'architecture à tester)
  FOR dut : latch USE ENTITY work.latch(flot_sr);
BEGIN
  dut : latch PORT MAP (si, ri,  qi, nqi);
-- processus de test
END latch_tb1;
```

Processus de test – un exemple

```
PROCESS
BEGIN
  si <= ‘1’;
  ri <= ‘0’;
  WAIT FOR 100ns;
  ASSERT (qi = ‘1’ AND nqi = ‘0’)
     REPORT "Set échoué!" SEVERITY error;
  si <= ‘0’;
  WAIT FOR 100ns;
  ASSERT (qi=‘1’ AND nqi = ‘0’)
     REPORT "Mem échoué" SEVERITY error;
END PROCESS;
```

- Un PROCESS est exécuté comme UNE instruction concurrente
- Le WAIT FOR fait avancé le temps de simulation

La simulation

- Identifier l'unité de conception principale (**toplevel**)
- **Élaboration**
  - Analyse des unités de compilation
  - Lier les composants aux ENTITY
  - Instancier les composants (éventuellement en utilisant la configuration)
- **Sémantique de simulation**
  - Chaque *process* dont au moins une entrée a été modifiée est exécuté
    - Les sorties sont calculées mais pas mise à jour
    - Les instructions concurrentes sont des *process* simples
  - Lorsque tous les *process* ont été exécutés, toutes les sorties sont mises à jour
  - Le temps de simulation est (si il y a lieu) mis à jour

La bibliothèque standard

- Le bit représente des signaux avec 2 valeurs logiques possibles
  - Souvent ce n'est pas suffisant !
  - Que se passe-t-il lorsque le latch SR a ses entrées à '0', '0' ?
  - On a déjà parlé de logique 3 états
- **Le type *std_ulogic*** défini une logique 9 états
  - ('0', '1', 'X', 'Z', 'W', '-')
  - C'est un **type non résolu** : un signal de type std_ulogic ne peut avoir qu'une seule source !
- **Le type *std_logic*** est le type résolu correspondant
  - Lorsqu'il y a plusieurs sources contradictoires, on applique une **fonction de résolution** prédéfinie

Le type *std_ulogic*

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

```
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
                     );
```

Le type *std_logic*

- On préfère *std_logic* à *bit*

```
subtype std_logic is resolved std_ulogic;
```

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	1	0	0	0	0	X
1	U	X	X	X	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

Les types X01, X01Z, UX01 et UX01Z

```
subtype X01  is resolved std_ulogic range
  'X' to '1'; -- ('X','0','1')

subtype X01Z  is resolved std_ulogic range
  'X' to 'Z'; -- ('X','0','1','Z')

subtype UX01  is resolved std_ulogic range
  'U' to '1'; -- ('U','X','0','1')

subtype UX01Z  is resolved std_ulogic range
  'U' to 'Z'; -- ('U','X','0','1','Z')
```

Les tableaux

- Les indices sont de type entier ou énuméré
- **Tableau non contraint** de bits

```
TYPE bit_vector IS ARRAY(natural RANGE <>) OF bit;
TYPE mot IS ARRAY(natural RANGE <>) OF std_logic;
```

- **Tableaux contraints**

```
SUBTYPE address IS bit_vector (0 TO 23);
SUBTYPE octet IS mot (0 TO 7);
TYPE memory IS ARRAY (0 TO 1023) OF octet;
```

- Signal de type tableau (pas nécessairement à 0)
  - d1 : **IN** mot (0 **TO** 15);
  - d2 : **OUT** mot (15 **DOWNTO** 0);

Compléments sur les tableaux

- Les indices des tableaux n'inclus pas nécessairement 0
- On peut avoir des tableaux de tableaux

```
TYPE mem1 IS ARRAY(0 TO 3, 1 DOWNTO 0) OF bit;
  code1 : mem1 := ((‘0’,‘0’),(‘0’,‘1’),
                   (‘1’,‘0’),(‘1’,‘1’));


TYPE mem2 IS ARRAY(0 TO 3) OF bit_vector(1 DOWNTO 0);
  code2 : mem2 := ("00", "01", "10", "11");
```

- On peut manipuler des sous-tableaux

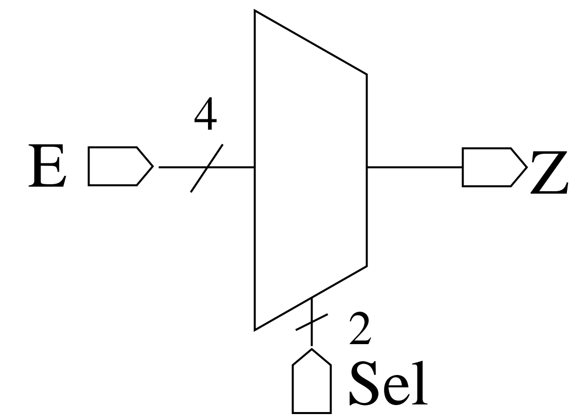
```
  SIGNAL etat : std_logic_vector(0 TO 4);
  SIGNAL sous_etat : std_logic_vector(0 TO 1);
  sous_etat <= etat (1 to 2); flag <= etat(3);
```

Les structures

```
TYPE ligneCache IS RECORD
   tag : mot(31 DOWNTO 0);
   dirty : bit;
   donnee : mot (15 DOWNTO 0);
END RECORD;
```

- On peut utiliser tous les types primitifs ou composés
- Mais en **synthèse**
  - les entiers peuvent être traduits de façons différentes
  - Les types réels ou physiques sont exclus

Exemple du *mux4*



- Un entité pour un multiplexeur à quatre entrées

```
ENTITY mux4 IS
   PORT (e : IN std_logic_vector(3 downto 0);
         sel : IN std_logic_vector(1 downto 0);
         z : OUT std_logic);
END mux4;
```

- Une équation concurrente peut contenir des opérateurs logiques (not, and, or, nand, xor, nor, …)

L'affectation conditionnelle concurrente

- Le comportement du multiplexeur peut être défini avec une seule **affectation concurrente conditionnelle (when-else)**
- Si un cas n'est pas couvert, le signal garde sa valeur (cf. others)

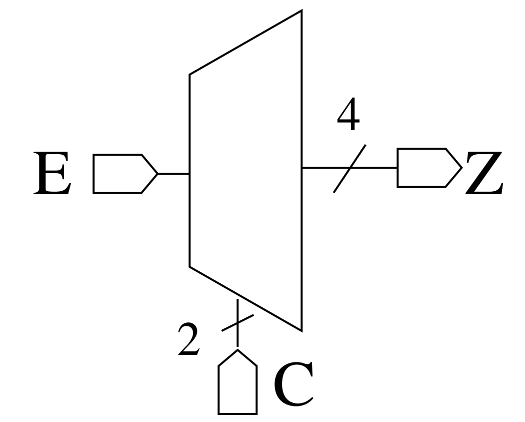
```
ARCHITECTURE flot_mux4 OF mux4 IS
BEGIN
    Z <=  E(0) WHEN (Sel = "00") ELSE
          E(1) WHEN (Sel = "01") ELSE
          E(2) WHEN (Sel = "10") ELSE
          E(3) WHEN (Sel = "11");
END flot_mux4;
```

La sélection <u>concurrente</u>

- Une autre construction concurrente est la **sélection concurrente (with-select-when)**
- Contrairement au *when-else*, **tous les cas doivent être traités** !

```
ARCHITECTURE flot2_mux4 OF mux4 IS
BEGIN
  WITH Sel SELECT
    Z <=  E(0) WHEN "00",
          E(1) WHEN "01",
          E(2) WHEN "10",
          E(3) WHEN "11",
          'X' WHEN others; -- les autres cas
END flot2_mux4;
```

Exemple du démultiplexeur 2x4




```
ENTITY demultiplexeur2x4 IS
   PORT (e : IN std_logic;
         c : IN std_logic_vector(2 downto 0);
         z : OUT std_logic_vector(4 downto 0));
END demultiplexeur2x4;
```

- Les équations logiques sont (réalisation en TP):

$$Z_0 = E \bullet \overline{C_0} \bullet \overline{C_1} \qquad Z_1 = E \bullet \overline{C_0} \bullet C_1$$

$$Z_2 = E \bullet C_0 \bullet \overline{C_1} \qquad Z_3 = E \bullet C_0 \bullet C_1$$

Les *PROCESS*

- On veut utiliser VHDL pour ne pas avoir à calculer toutes les équations
- **Les *process* sont des instructions concurrentes dont le corps s'exécute séquentiellement**
  - Dans un *process*, on ne précise pas la structure, mais le comportement (un algorithme)
- Les *process* ont une **liste de sensibilité** qui décrit la liste des signaux dont la modification déclenche l'exécution du *process*.
- Exemple : registre à décalage 8 bits
  - On pourrait calculer toutes les équations et faire un modèle flot de données

Registre à décalage - ENTITY

```
SUBTYPE data IS std_logic_vector(7 DOWNTO 0);
ENTITY decalage IS
  PORT (din : IN data;     -- entrée
        dout : OUT data; -- sortie
        charge : IN std_logic;
        clk : IN std_logic; -- décale
        reset : IN std_logic -- asynchrone);
END decalage;
```

- Un registre a **décalage synchrone** sur front montant
- Le **reset** est asynchrone actif au niveau bas
- La **charge** est synchrone active au niveau haut

Modélisation comportementale

```
ARCHITECTURE dec1 OF decalage IS
   SIGNAL qtmp : data;
BEGIN
   reg : PROCESS (Clk, Reset) -- pas charge !
   BEGIN
      -- calcul de la sortie (qtmp)
      -- le corps du process est séquentiel
   END PROCESS;

   dout <= qtmp; -- affectation concurrente
END dec1;
```

Liste de sensibilité

Corps du process

Nom du *process* (si ambiguïté)

Synchrone ou asynchrone ?

```
reg: PROCESS(Reset,Clk)
BEGIN
  IF Reset = '0' THEN   -- reset asynchrone (L)
    qtmp <= "00000000";
  ELSIF (Clk = '1' AND Clk'EVENT) THEN
    IF (Charge = '1') THEN -- charge synchrone (H)
      qtmp <= din;
    ELSE  -- décalage synchrone (garde le signe)
      qtmp <= qtmp(7) & qtmp(7 DOWNTO 1);
    END IF;
  END IF;
END PROCESS;
```

Concaténation de *std_logic* et *std_logic_vector*

Liste de sensibilité

- Le *process* reg dépend de ses deux entrées **asynchrones** (Reset, Clk)
  - Le processus est activé lorsqu'un événement se produit sur un des signaux de sa liste de sensibilité
  - La commande `charge` est **synchrone par rapport** à `Clk` !
- L'exécution d'un *process* est **« instantané »** *a priori*
  - Les instructions ne consomment pas de temps de simulation
  - Cela ne reflète pas nécessairement la réalité. Après synthèse, des délais de propagation sont appliqués en fonction de la technologie.
- La liste de sensibilité est obligatoire si il n'y a pas d'instruction qui consomme du temps de simulation (`wait for`, `after`)
  - Sinon, le processus est réactivé indéfiniment (boucle infinie)
- Lorsqu'un *process* n'est pas actif, il est suspendu

Générer une horloge ou un reset

```
SIGNAL clk, rst : bit;

horloge : PROCESS -- signal périodique
BEGIN -- exécution séquentielle dans le corps
  clk <='0';
  WAIT FOR 100ns;
  clk <= '1';
  WAIT FOR 50 ns; -- horloge asymétrique
END PROCESS;

Rst <= '1', '0' AFTER 50 ns, '1' AFTER 100ns;
```

- Le *process* `horloge` et l'affectation de `Rst` sont exécutés en concurrence !

Sémantique de simulation

- Que fait ce *process* ?

```
SIGNAL a,b : INTEGER := 0 ;
PROCESS
BEGIN
  a <= a + 1 ;
  b <= b + a ;
  WAIT FOR 100 ns;
END PROCESS;
```

- Que vaut b après la première exécution ?
- L'exécution est pourtant séquentielle !

Variables et signaux

- Les **affectations séquentielles de signaux** sont appliquées quand tous les *process* ont été exécutés !
- Les **affectations de variables** (forcément séquentielles) sont appliquées immédiatement !

```
SIGNAL a,b : INTEGER := 0 ;
PROCESS
  VARIABLE t : INTEGER;
BEGIN
  t := a + 1 ; -- affectation de variable
  a <= t ;
  b <= b + t ; -- utilise la nouvelle valeur de t
  WAIT FOR 100 ns;
END PROCESS;
```

Opérations arithmétiques

- La norme IEEE 1076.3 fournit deux types pour effectuer des opérations **signées**, ou **non signées** sur les tableaux de *std_logic* : paquetage `std_logic_arith`.

```
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

- Sur *signed* et *unsigned*, les opérations arithmétiques standards sont définies (addition, soustraction)
- Exemple :

```
-- 4 bits non signés
LIBRARY ieee; USE ieee.std_logic_arith.all;
SIGNAL a, b : unsigned (3 downto 0) ;
a <= a + b;
```

Opérateurs de conversion

- Le paquetage `std_logic_arith` définit également des opérateurs de conversion dans les deux sens

```
SIGNAL a1,a2 : std_logic_vector(3 downto 0);
SIGNAL b : unsigned (3 downto 0);

-- de std_logic_vector vers unsigned
b <= unsigned(a1);
b <= unsigned("1010");

-- de unsigned vers std_logic_vector
a2 <= std_logic_vector(b);
```

Compléments syntaxiques

- L'attribut **RANGE** permet de connaître le domaine de définition d'un tableau
  - E.g. `VARIABLE t : signed(inputA'RANGE);`
  - -- la variable t aura la même taille que le signal A
- Les attributs **HIGH** et **LOW** pour connaître les limites inférieures et supérieures d'un tableau
  - E.g. `SIGNAL tmp : signed(inputA'LOW TO 2);`
  - -- même minimum que *inputA*, 2 comme maximum
- Le mot clé **OTHERS** permet une initialisation de plusieurs membres à la fois
  - `SIGNAL a : signed(A'RANGE):=(0=>'1', OTHERS=>'0');`
  - `A:="000…001";` quelle que soit la taille de A !

For : génération automatique (concurrente)

```
COMPONENT registre IS
   PORT (…);
END COMPONENT;

 TYPE sortie IS ARRAY (1 TO 15) OF std_logic_vector(7 DOWNTO 0);
 SIGNAL sn : sortie;
 SIGNAL wrn, rdn : std_logic_vector(1 TO 15);

BEGIN
 gr : FOR i IN 1 TO 15 GENERATE
   r : registre PORT MAP (e, sn(i), clk, wrn(i), rdn(i), reset);
   wrn(i) <= '0' WHEN wr = i ELSE '1';
   rdn(i) <= '0' WHEN (rd1 = i OR rd2 = i) ELSE '1';
 END GENERATE;
```

Boucle For : séquentiel

```
SIGNAL D : std_logic_vector(0 TO 7);
BEGIN
  PROCESS (D)
        VARIABLE otmp: Boolean;
  BEGIN
        otmp := false;
        l : FOR i IN D'LOW TO D'HIGH LOOP
            IF D(i) = '1' THEN
                otmp := not otmp;
            END IF;
        END LOOP;
  END PROCESS;
```

Additionneur comportemental

- Entité

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY add4 IS
  PORT ( a,b : IN std_logic_vector(3 downto 0);
         cin : IN std_logic;
         s : OUT std_logic_vector(3 downto 0);
         cout : OUT std_logic);
END add4;
```

Additionneur comportemental

- Architecture

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
ARCHITECTURE add4_beh OF add4 IS
  SIGNAL a_uint, b_uint : unsigned(3 DOWNTO 0);
  SIGNAL res : unsigned(4 DOWNTO 0);
BEGIN
  a_uint <= unsigned(a);
  b_uint <= unsigned(b);
  res <= ('0' & a_uint) + b_uint + cin;
  s <= std_logic_vector(res(3 DOWNTO 0));
  cout <= std_logic(res(4));
END add4_beh;
```

Testbench

Bibliographie

- « VHDL – langage, modélisation, synthèse », 2nd Ed.
  - Presses polytechniques et universitaires romandes
- http://www.vhdl.org
  - Travaux en cours
- http://www.amontec.com/fix/vhdl_memo/index.html
  - Mémo sur la syntaxe VHDL
- http://www.opencores.org
  - Cœurs de processeurs en VHDL
- http://www.acc-eda.com/