HDL et SystemVerilog

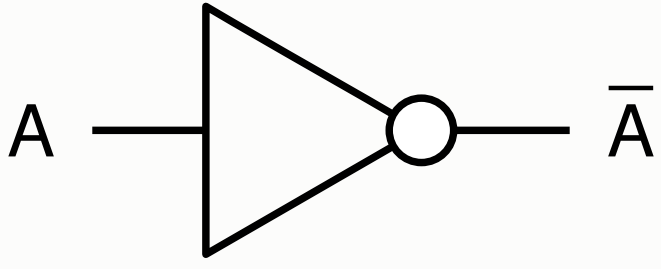
Introduction aux langages de description du matériel

Tarik Graba
tarik.graba@telecom-paristech.fr
Année scolaire 2020/2021

Plan

Représentation de fonctions numériques

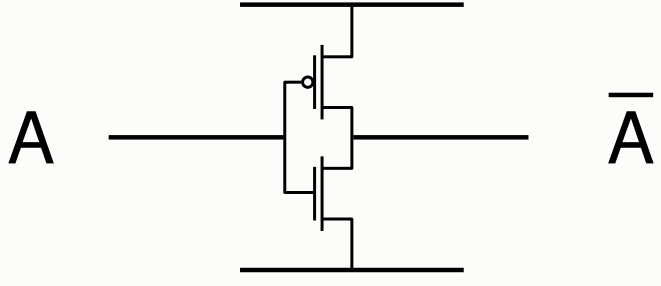
Schéma d'une fonction booléenne



Inverseur

Représentation de fonctions numériques

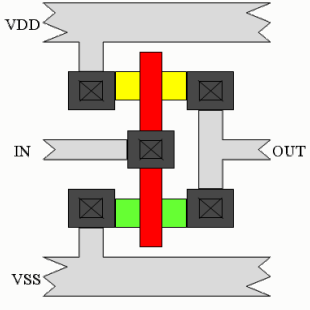
Schéma des transistors



Inverseur CMOS

Représentation de fonctions numériques

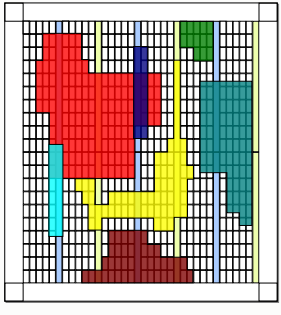
Dessin des couches physiques



Inverseur CMOS intégré

Représentation de fonctions numériques

Et dans le FPGA ?



Programmer des cellules

Représentation de fonctions numériques

Comment représenter ?

- Dessins/schémas ?
  - Pas facile…
- Équations ?
  - Comment représenter la structure physique ?
  - Comment représenter le temps ?
- Autres ?

Quel niveau de représentation ?

- Logique ? Structurel ? Physique ?

Représentation de fonctions numériques

Autres niveaux d'abstraction ?

Augmenter la productivité

- Abstraire
- Réutiliser

Quel niveau de représentation ?

- Algorithme ?
- autres…

HDL

HDL

- **HDL : H**arware **D**escription **L**anguage.
- Langage informatique de description du matériel.

HDL

Ces langages doivent permettre deux choses

- Concevoir/Réaliser
  - Implémenter
  - Fabriquer
- Modéliser/Simuler
  - Tester la fonctionnalité

HDL : Une représentation textuelle

Qui permet :

de représenter la structure

L'inverseur

```
not(nA, A)
```

une porte logique avec une entrée et une sortie

HDL : Une représentation textuelle

Qui permet :

de représenter son comportement

L'inverseur

```
nA = !A
```

son comportement sous forme d'une équation

HDL : Une représentation textuelle

Qui permet :

de représenter son comportement

L'inverseur

```
if(A) nA = 0 else nA = 1
```

son comportement par une séquence (fonctionnellement)

HDL : Les plus

Automatisation

- **EDA : E**lectronic **D**esign **A**utomation.
- Conception électronique automatisée.
- Utilisation de l'outil informatique pour générer les autres représentations.

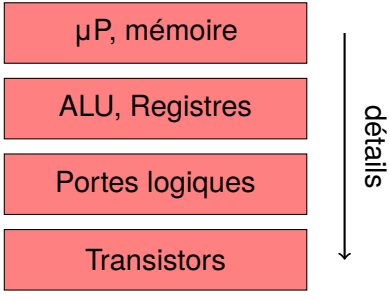
Abstraction et productivité

- S'abstraire de la cible technologique.
- Utiliser des représentations de plus haut niveau.
  - Ne pas se limiter à des équations logiques.

Plan

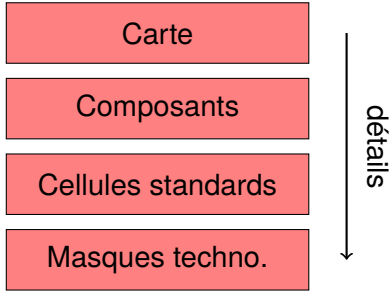
Description structurelle

- Des composants
- Des connexions
- On parle de «netlist»
- …
- Conception par assemblage



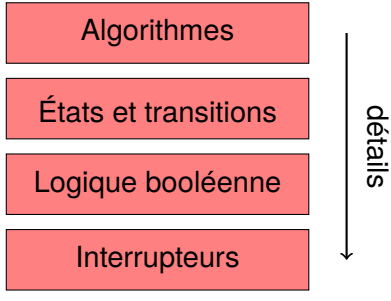
Description physique

- Les matériaux
- Les dimensions
- …
- Nécessaires pour la fabrication



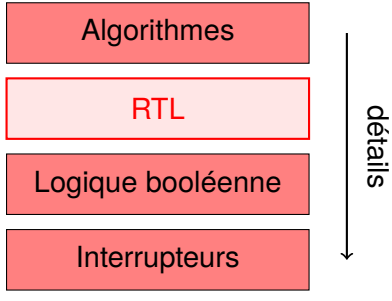
Description comportementale

- Décrire la fonction réalisée.
- …
- C'est ce qui nous intéresse ici !



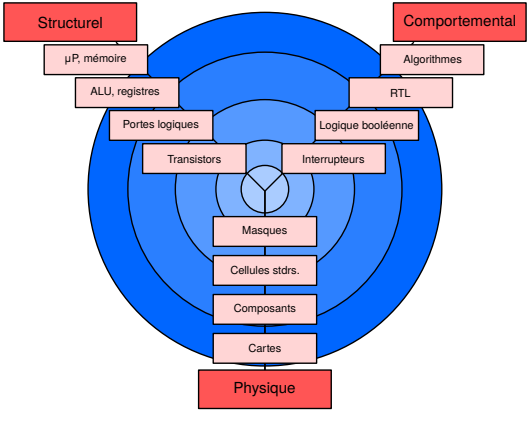
Description comportementale

- Décrire la fonction réalisée.
- …
- C'est ce qui nous intéresse ici !

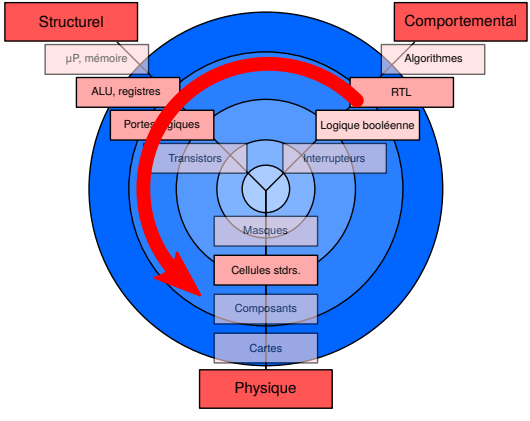


Niveaux de description

Équivalence mais différentes finalités

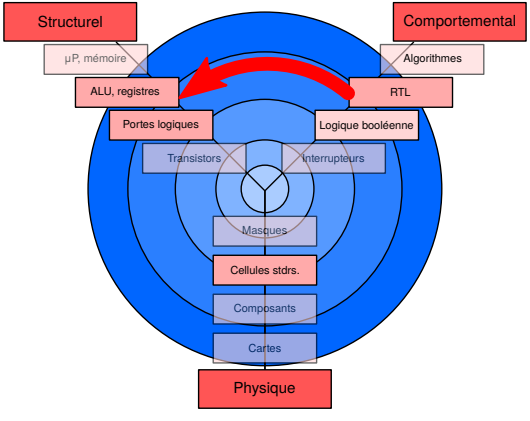


Niveaux de description
Synthèse automatique



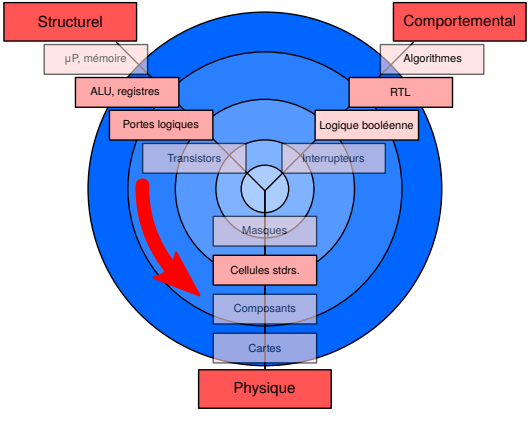
Passage d'une représentation comportementale à la fabrication/programmation

Niveaux de description
Synthèse automatique



Une étape de synthèse logique générique

Niveaux de description
Synthèse automatique



Une étape de synthèse physique spécifique

Plan

Le «RTL»

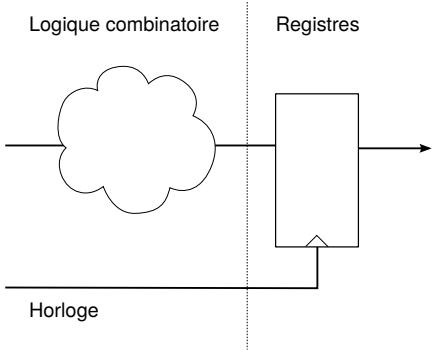
Représentation des états en logique synchrone

RTL

- **RTL : R**egister **T**ransfer **L**evel.
- Le niveau « transfert entre registres ».

Le «RTL»

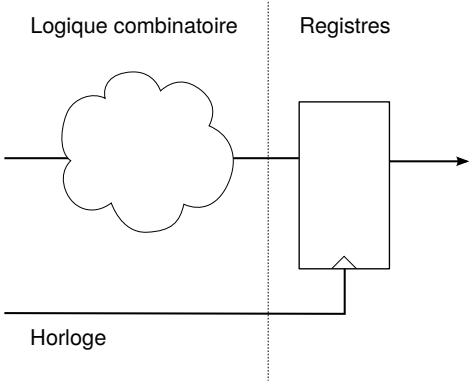
Représentation des états en logique synchrone



Un registre (ou une bascule) est un élément mémorisant dont le changement d'état est déclenché par un signal d'horloge.

Le «RTL»

Représentation des états en logique synchrone



- Une horloge explicite !
- Il faut décrire ce qui ce passe à chaque coup d'horloge.

Les algorithmes doivent être transformés pour exprimer ce qui se passe à chaque cycle d'horloge.

Le «RTL»

Représentation des états en logique synchrone

1- L'algorithme

```
int i;

for (i = 0; i< 10; i++)
{
   ...
}

...
// puis on utlise i
```

- pas de notion de temps ou d'horloge

2- L'algorithme + hypothèses d'architecture

```
int i; // <- valeur signée sur 32 bits

for (i = 0; i< 10; i++)   // <--- une itération par cycle?
{
   ...
}

...
// puis on utlise i // <- et ainsi de suite
```

- Faire des hypothèses sur l'architecture

3- Attribuer des ressources

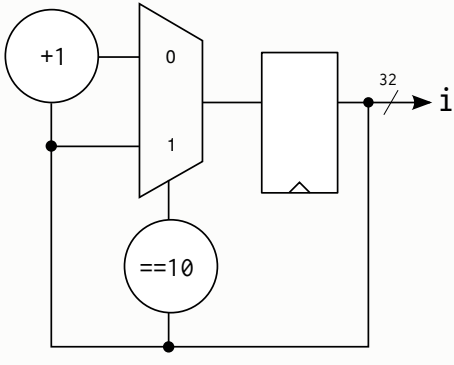
- Un registre pour stocker `i` .
  - Qui change à chaque front d'horloge.
  - Suffisamment grand (32 ou 4 bits ?)
- Un additionneur (ou incrémenteur).
- Un comparateur.

3- Description RTL

```
i[32]
sum[33]
cond[1]

// logique combinatoire
cond = i<10
sum  = i + 1

// registres
@(clk) i = sum[31:0]
// ou plus simplement "@(clk) i = i + 1"
```

- Décrire la logique combinatoire.
- Décrire l'évolution des registres.

4- Synthèse automatique

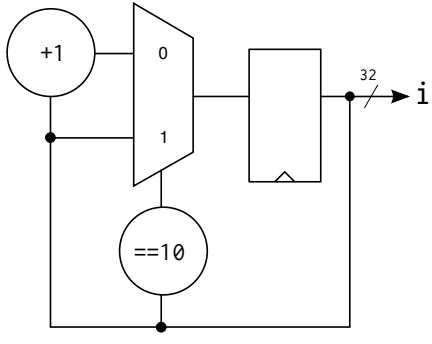


Le «RTL»

Représentation des états en logique synchrone

```
i[32]
sum[33]
cond[1]

// logique combinatoire
cond = i<10
sum  = i + 1

// registres
@(clk) i = sum[31:0]
// ou plus simplement "@(clk) i = i + 1"
```

→



Pourquoi le niveau RTL ?

- Suffisamment haut niveau pour représenter «simplement» tout système numérique synchrone.
  - chemin de données
  - contrôle, MAE
  - …
- Abstraction de la technologie.
- Il existe depuis longtemps des outils automatiques fiables pour la synthèse
  - Synthèse logique puis physique.

Plan

Simuler le comportement du matériel

Simuler efficacement une représentation RTL

- D'un côté,
  - le matériel est intrinsèquement parallèle,
  - tous les composants d'un circuit sont actifs et fonctionnent en même temps,
  - le nombre d'éléments peut être nombreux.
- D'un autre côté,
  - les simulations sont exécutées sur des machines séquentielles (le processeur de votre PC) ;
  - il nous est plus simple de décrire des séquences.

Simuler le parallélisme

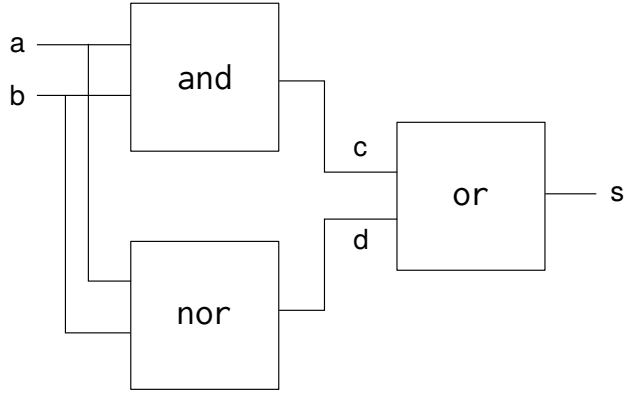
- Comment simuler efficacement du matériel ?
- On ne veut pas simuler ce qui se passe physiquement ! Uniquement le comportement (logique/arithmétique) à un niveau RTL.

Simuler le parallélisme

Exemple : de la logique combinatoire

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x



Simuler le parallélisme

Pour de la logique combinatoire

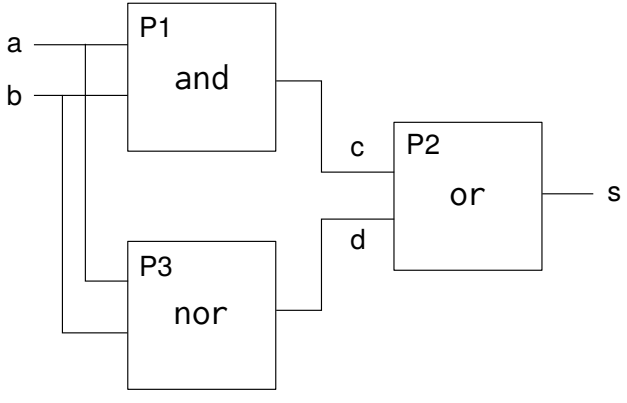
- Une fonction par bloc combinatoire
  - qu'on appellera **processus**
  - les instructions dans un processus sont exécutées séquentiellement
- On agit sur des variables globales vues par tous les processus
- On exécute les processus dans un ordre quelconque
- On re-exécuter tant qu'il y a des changements
- Ne fonctionne plus s'il y a une boucle

Simuler le parallélisme

Exemple : de la logique combinatoire

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x

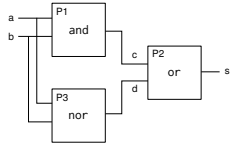


Simuler le parallélisme

Pour de la logique combinatoire

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x



	0				1				2				3
		P1	P2	P3		P1	P2	P3		P1	P2	P3	
c	x	0	–	–	0	0	–	–	0	0	–	–	0
d	x	–	–	0	0	–	–	0	0	–	–	0	0
s	x	–	x	–	x	–	0	–	0	–	0	–	0

Simuler le parallélisme
Le temps symbolique

- Chaque cycle d'exécution du simulateur est appelé **temps symbolique** (ou delta Δ).
- Ça ne représene pas un temps «physique' !
- Tant qu'un processus a besoin d'être exécuté, on est dans le même delta.

Simuler le parallélisme

Le temps symbolique

					fin				fin				fin
	0ns	0ns			0ns	0ns			0ns	0ns			0ns
	init	Δ_1	$\rightarrow$		Δ_1	Δ_2	$\rightarrow$		Δ_2	Δ_3	$\rightarrow$		Δ_3
		P1	P2	P3		P1	P2	P3		P1	P2	P3	
c	x	0	-	-	0	0	-	-	0	0	-	-	0
d	x	-	-	0	0	-	-	0	0	-	-	0	0
s	x	-	x	-	x	-	0	-	0	-	0	-	0

Simulation événementielle
évènements et liste de sensibilité

Pour accélérer la simulation, il ne faut exécuter que les processus dont les entrées ont changé.

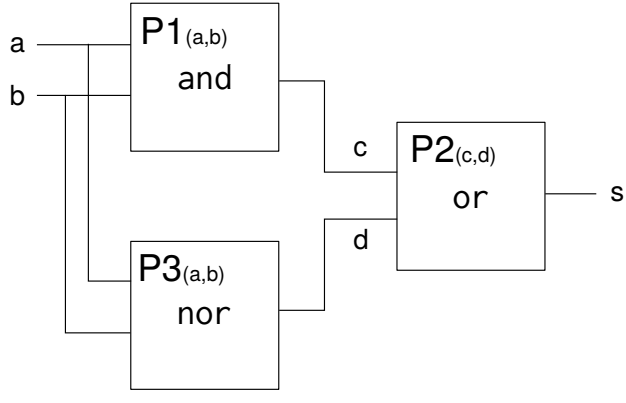
On ajoute alors deux notions :

- Les évènements sur les entrées
  - Si une entrée change
- La liste de sensibilité
  - La liste des évènements qui nécessitent l'exécution d'un processus

Simulation événementielle
évènements et liste de sensibilité

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x

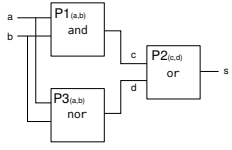


Simulation événementielle

évènements et liste de sensibilité

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x



				fin		fin
	0ns	0ns		0ns	0ns	0ns
	init	Δ_1	$\rightarrow$	Δ_1	Δ_2	Δ_2
		P1	P3		P2	
c	x	0	-	0	-	0
d	x	-	0	0	-	0
s	x	-	-	x	0	0

On a réduit le nombre d'itérations de simulation et de fonctions exécutées.

Simulation événementielle
La gestion du temps

Le simulateur maintient un compteur pour le **temps physique** indépendant du temps symboliques.
À un évènement sont attachés 2 informations de temps :

- Le temps symbolique Δ
  - ou le cycle de simulation
- Le temps physique
  - celui qu'on voudrait observer

Ceci permet d'ordonner les événements dans **l'échéancier** du simulateur ou de préciser le temps physique auquel il doit être pris en compte.

On parle de **notification**.

Simulation événementielle
Exemple

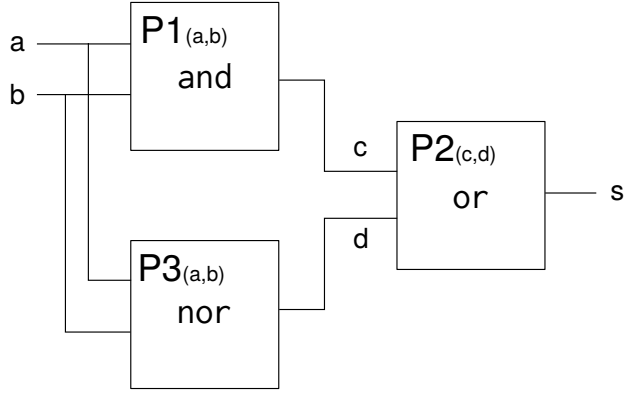
- a = x
- b = x
- c = x
- d = x
- s = x

@t=0ns

- a = 1
- b = 0

@t=10ns

- a = 1
- b = 1



Simulation événementielle

Exemple

- a = x
- b = x
- c = x
- d = x
- s = x

@t=0ns

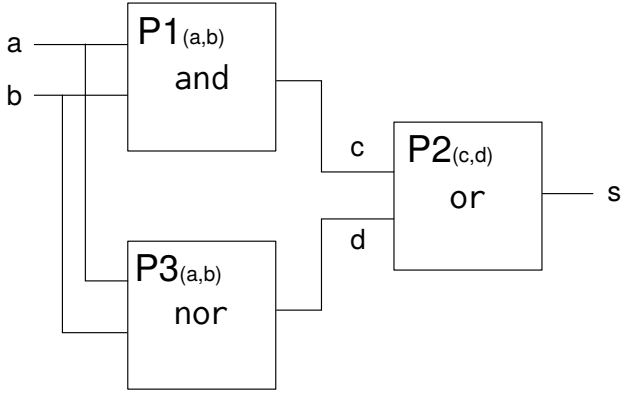
- a = 1
- b = 0

@t=10ns

- a = 1
- b = 1

@

P0



Simulation événementielle
le temps physique

		fin			fin		fin		fin			fin		fin
		0ns			0ns	0ns	0ns	→	10ns	10ns		10ns	10ns	10ns
	init	Δ_0	Δ_1	→	Δ_1	Δ_2	Δ_2		Δ_0	Δ_1	→	Δ_1	Δ_2	Δ_2
		P0	P1	P3		P2			P0	P1	P3		P2	
a	x	1	-	-	1	-	1		1	-	-	1	-	1
b	x	0	-	-	0	-	0		1	-	-	1	-	1
c	x	x	0	-	0	-	0		-	1	-	1	-	1
d	x	x	-	0	0	-	0		-	-	0	0	-	0
s	x	x	-	-	x	0	0		-	-	-	0	1	1

Le temps physique avance quand il n'y a plus d'évènements déclenchant un processus.

Simulation événementielle
Fonctionnement des processus

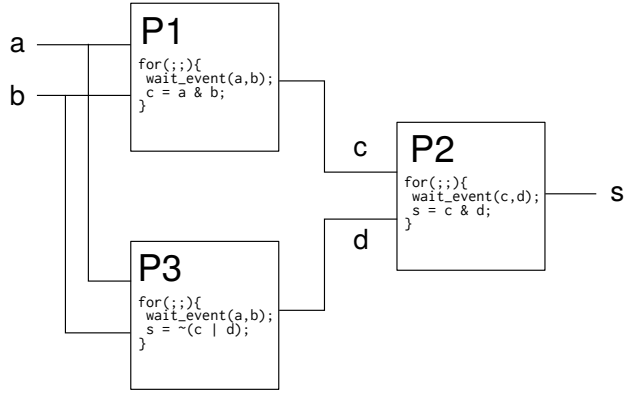
Les processus sont en pratique des **boucles infinies**.

Ils sont en attente d'un événement :

- Implicite due à la liste de sensibilité
- Explicite à l'aide d'instructions de synchronisation

Simulation événementielle
Fonctionnement des processus

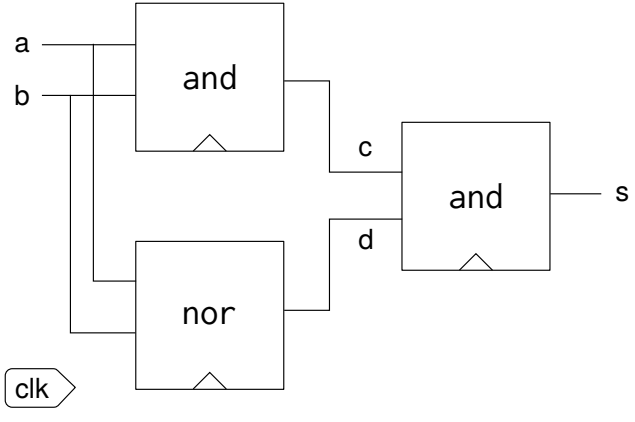
```
     for(;;){
     wait_for(0,ns);
     a = 1;
     b = 0;
P0   wait_for(10,ns);
     a = 1;
     b = 1;
     wait_forever();
     }
```



Simulation événementielle
et logique séquentielle ?

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x



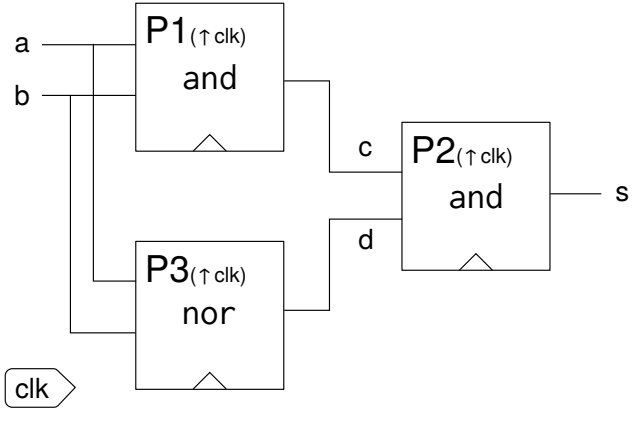
Simulation événementielle
et logique séquentielle ?

- Un évènement unique sur une entrée particulière, l'horloge.

Simulation événementielle
et logique séquentielle ?

Initialement :

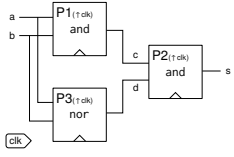
- a = 1
- b = 0
- c = x
- d = x
- s = x



Simulation événementielle
et logique séquentielle ?

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x

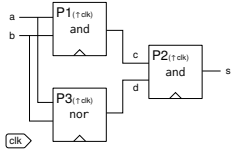


							fin							fin
		0ns	→	↑ clk				→	↑ clk					
	init	Δ_0		Δ_1	→	→	Δ_1		Δ_1	→	→	Δ_1		
		P0		P1	P2	P3			P1	P2	P3			
a	x	1		–	–	–	–		–	–	–	–		
b	x	0		–	–	–	–		–	–	–	–		
c	x	x		0	–	–	0		0	–	–	0		
d	x	x		–	–	0	0		–	–	0	0		
s	x	x		–	0	–	0		–	0	–	0		

Simulation événementielle
et logique séquentielle ?

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x



							fin						fin
		0ns	→	↑ clk				→	↑ clk				
	init	Δ_0		Δ_1	→	→	Δ_1		Δ_1	→	→	Δ_1	
		P0		P2	P3	P1			P2	P3	P1		
a	x	1		–	–	–	1		–	–	–	1	
b	x	0		–	–	–	0		–	–	–	0	
c	x	x		–	–	0	0		–	–	0	0	
d	x	x		–	0	–	0		–	0	–	0	
s	x	x		x	–	–	x		0	–	–	0	

Simulation événementielle
et logique séquentielle ?

Problème

- Tous les processus sont déclenchés en même temps.
- En modifiant instantanément les sorties, le résultat n'est pas tout le temps le même, il dépend de l'ordre d'exécution.

Simulation événementielle
Les signaux

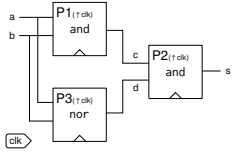
- Un **signal** est une structure de données contenant :
  - la valeur **courante** avant le Δ
  - la valeur **future** qu'il aura après le Δ
- Une affectation sur un signal ne modifie que sa valeur future
- Lire un signal renvoie sa valeur courante
- À la fin du Δ la valeur du signal est mise à jour
  - ie. la valeur future remplace la valeur courante

La valeur d'signal est maintenue tant que tous les processus actifs à un «Δ» n'ont pas été exécutés. On parle d'**affectations différées**.

Simulation événementielle
et logique séquentielle ?

Initialement :

- a = 1
- b = 0
- c = x
- d = x
- s = x



	m.à.j					m.à.j				m.à.j
	0ns	→	↑ clk			→	↑ clk			
init	Δ_0		Δ_1	→	→	Δ_1	Δ_1	→	→	Δ_1
	P0		P1	P2	P3		P1	P2	P3	
a (x,x)	(x,1)	(1,1)	-	-	-	-	-	-	-	-
b (x,x)	(x,0)	(0,0)	-	-	-	-	-	-	-	-
c (x,x)	(x,x)	(x,x)	(x,0)	-	-	(0,0)	(0,0)	-	-	(0,0)
d (x,x)	(x,x)	(x,x)	-	-	(x,0)	(0,0)	-	-	(0,0)	(0,0)
s (x,x)	(x,x)	(x,x)	-	(x,x)	-	(x,x)	-	(x,0)	-	(0,0)

Simulation événementielle (SPOILER)

Les affectations en Verilog/SystemVerilog

En Verilog/SystemVerilog, il n'y pas de différence de déclaration entre une variable et un signal. C'est le symbole utilisé pour l'affectation qui permet de faire la différence :

- `a <= b` : affectation différée (donc signal)
- `a = b` : affectation immédiate (donc variable)

Simulation événementielle (SPOILER)

Les affectations en Verilog/SystemVerilog

`a <= b` : affectation différée

Doivent être utilisées pour modéliser de la logique séquentielle de façon deterministe.

`a = b` : affectation immédiate

Peuvent être utilisées pour modéliser de la logique combinatoire.

Simuler le parallélisme
Résumons

- Décrire sous la forme d'un ensemble de fonctions : **les processus**
- La communication entre ces processus se faisant en modifiant des variables globales de type **signal** pour garantir le déterminisme.
- On définit la liste des **évènements** sur les **signaux** qui nécessitent de relancer des processus.

Simuler le parallélisme
principes

1. Exécuter les processus dans un ordre arbitraire.
2. Si on demande à modifier la valeur d'un signal, mémoriser sa valeur **future**.
   Les variables sont modifiées immédiatement.
3. Quand tous les processus ont été exécutés (à la fin du Δ), modifier vraiment la valeur des signaux.
4. Refaire les étapes 1,2,3 si on a déclenché un nouvel évènement.
5. S'il n'y a aucun évenement, alors, faire avaner le temps physique

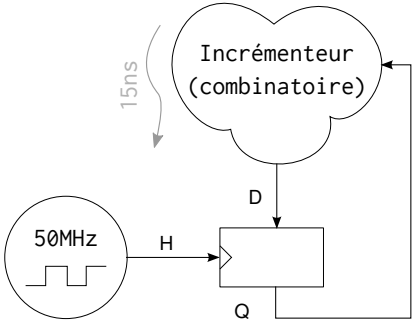
Simulation événementielle
Exercice à faire

On veut simuler un système composé :

- un générateur d'horloge,
- un registre,
- un incrémenteur.

Avec

- `50MHz` de fréquence d'horloge,
- `15ns` de temps de propagation.



Simulation événementielle

Exercice à faire

Simulez «à la main» la description suivante :

Notation :

`@(x)`	:	attente d'un évènement sur `x`
`#10`	:	processus stoppé pour `10ns`
`x <= y`	:	affectation différée à la fin du Δ
`x <= #10 y`	:	affectation différée de `10ns`

P1

```
H <= 0;
#10;
H <= 1;
#10;
```

P2

```
@(H);
if H ==1
 Q <= D;
```

P3

```
@(Q);
D <= #15 Q+1;
```

À mettre dans votre dépôt git dans un dossier **`ExoSim`**!

Plan

Plan

Historique

- 1990: Cadence Design System rend public Verilog HDL.
- 1995: devient la standard IEEE 1364-1995.
- 2001: amélioration IEEE 1364-2001.
- 2005: amélioration IEEE 1364-2005.
- 2005: l'extension SystemVerilog est standardisée IEEE 1800-2005
- 2009: standard unique SystemVerilog IEEE 1800-2009
- 2012: dernière révision du standard IEEE 1800-2012

SystemVerilog

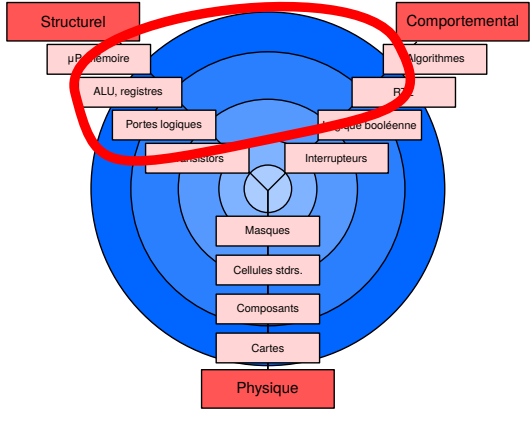
HDVL

- **HDVL : H**ardware **D**escription and **V**erification **L**anguage
- Langage de description et de vérification du matériel.

Ce cours ne couvre pas les aspects avancés de la vérification.

SystemVerilog

- Historiquement Verilog permet de représenter la logique du niveau RTL au niveau transistor !
- Les ajouts pour la vérification permettent de monter au niveau algorithmique.



Plan

Syntaxe

- Fichiers texte d'extension `.sv`
  - (`.v`) pour le Verilog
- Les commentaires sont les même qu'en C
  - `//` pour commenter une ligne.
  - `/* ... */` pour commenter un bloc.
- Les instructions se terminent par un point-virgule (;)
- un bloc est délimité par `begin ... end`

Oui c'est un langage informatique …

Fichier texte `hello.sv`

```
module foo ( );

initial
begin
   // $display est une tache système
   $display("hello world");
end

endmodule
```

- `vlib work`
- `vlog hello.sv`
- `vsim -c foo`

- `qverilog hello.sv`

Plan

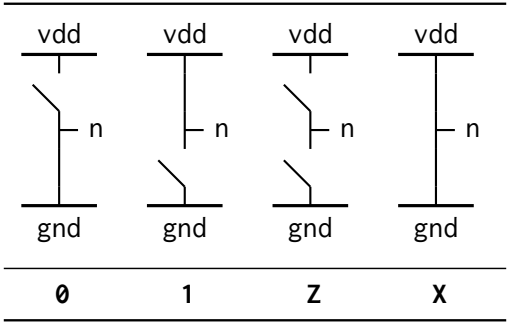
Les valeurs logiques

Pour représenter les différents états dans un circuit électronique, en SystemVerilog on utilise les 4 états suivant :

- `0` : l'état logique `0`/faux.
- `1` : l'état logique `1`/vrai.
- `x/X` : l'état inconnu ou conflit.
- `z/Z` : haute impédance (nœud flottant).

Les valeurs logiques

Schématiquement, pour un nœud :



L'état initial, inconnu, d'un registre sera aussi `X` .

Les nœuds

Les nœuds servent à décrire les interconnexions entre différents éléments d'une représentation structurelle.
On utilise le type **`wire`** .

```
wire a;          // déclare un noeud, a, sur 1 bit
wire b, c, d;    // déclare trois noeuds, b, c, et d, sur 1 bit chacun
wire [7:0] data; // déclare un bus de 8 noeuds data.
```

Les nœuds (`wire`) ne sont pas modifiables dans un processus.

Les variables

Les variables sont utilisées dans les processus.
En Verilog le type **`reg`** était utilisé, en SystemVerilog il a été renommé en **`logic`** pour éviter d'être assimilé à un registre au sens électronique (bascules D).

```
logic a;             // déclare une variable, a, sur 1 bit
logic b, c, d;       // déclare trois variables, b, c, et d, sur 1 bit chacun
logic [7:0] result; // déclare un mot de 8 bits.
```

Les bus

Des nœuds ou des variables logiques peuvent être regroupés dans un bus (vecteur de plusieurs bits).

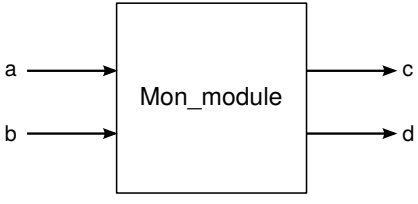
```
logic [7:0] A; // déclare un vecteur de 8 bits de type logic.
wire  [1:8] B; // déclare un vecteur de 8 bits de type wire.

A[4] = ...   ; // le bit n° 4 de A
B[0] = ...   ; // Attention le bit 0 n'existe pas

A[7:4] = ... ; // Le demi-octet de poids fort de A
B[1:4] = ... ; // Le demi-octet de poids fort de B
A[0:3] = ... ; // Erreur ne correspond pas à l'ordre de déclaration

A[5 -: 4] = ... ; // 4 bits de A à partir de la position 5 (5,4,3,2)
B[5 +: 4] = ... ; // 4 bits de B à partir de la position 5 (5,6,7,8)

// Plus de détails section 11.5.1 de la norme
//              "Vector bit-select and part-select addressing"
```

Plan

Le module

- L'élément de base de tout code SystemVerilog.
- Il représente le circuit ou l'un de ses sous blocs.
- Tout code SystemeVerilog décrivant un circuit doit appartenir à un module.

Le module

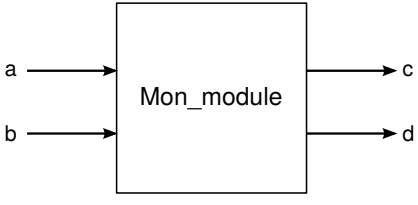


Un module doit avoir :

- Un nom pour l'identifier.
- Une interface décrivant ses entrées et sorties.
- Une description.

Le module
Syntaxe

Commence par **`module`** et se termine par **`endmodule`**




```
module Mon_module ( /* interface */);

   // description

endmodule
```

Le module
L'interface

Deux façons de faire :

```
// style verilog 2001
module mon_module ( input        a,
                    input  [7:0] b,
                    output [7:0] c,
                    output logic d
                    );

    // description
    ....

endmodule
```

```
// style verilog 95
module mon_module ( a,b, c, d );

    input        a;
    input  [7:0] b;
    output [7:0] c;
    output       d;
    logic        d;

    //description
    ....

endmodule
```

nb. Les **`input`** / **`output`** sont par défaut implicitement des **`wire`** .

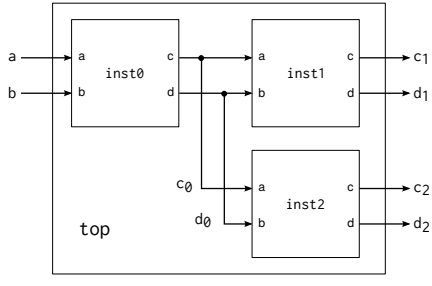
Plan

Les instances

On peut décrire un module structurellement en :

- instanciant des sous modules.
- définissant les interconnexions

```
module top (
            input a, b,
            output c1,d1,
            output c2,d2
            );
// interconnexions
wire c0, d0;

// structure
mon_module inst0 (.a(a),  .b(b),
                  .c(c0), .d(d0));
mon_module inst1 (.a(c0), .b(d0),
                  .c(c1), .d(d1));
mon_module inst2 (.a(c0), .b(d0),
                  .c(c2), .d(d2));

endmodule
```



Pourquoi faire du structurel ?

- Pour découper une module complexe en sous modules plus simple.
  - Chemin de données/Contrôle.
  - Découpage fonctionnel
- Pour réutiliser un module existant.
  - qu'on a conçu.
  - qu'on nous a donné.
- Pour se partager le travail en équipe.
  - Comme pour un développement logiciel.

Plan

Les affectations continues

Parfois on veut affecter le résultat d'un calcul à un nœud (`wire`). On utilise pour cela une affectation continue.

Exemple

```
module mux (
            input a,b,s,
            output o
            );

   // affectation continue
   assign o = s? a : b;

endmodule
```

- o est réévalué si a, b ou s change.

On ne peut modéliser que de la logique combinatoire avec les affectations continues.

Les processus

`always` : Processus exécuté de façon continue. Nécessite une liste de sensibilité ou des points d'arrêt explicites.

`initial` : Processus exécuté qu'une fois en début de **simulation** (temps `0`).

Les instructions dans un processus sont exécutées de façon séquentielle.
L'ordre d'exécution des processus n'est pas spécifié.

Les affectations

- `<=` Affectation différée
- `=` Affectation immédiate

Exemple

```
// a= 0, b=1, c=2
...
  b <= a;
  c <= b;
// à la prochaine synchro.
// explicite @/# ou implicite
// a=0, b=0, c=1
```

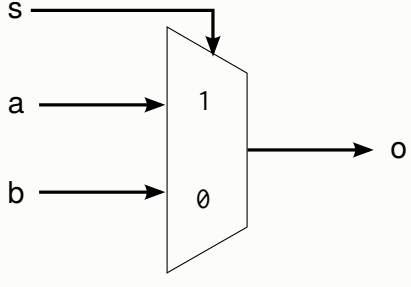
```
// a= 0, b=1, c=2
...
  b = a;
  c = b;

// à la prochaine instruction
// a=0, b=0, c=0
```

La liste de sensibilité

La liste de sensibilité est la liste des signaux (nœuds et variables) dont la modification déclenche un processus.

Exemple

```
module mux21(
              input  s,
              input  a, b ,
              output logic o
             );

always @(s,a,b)
  if (s) o = a;
  else   o = b;

endmodule
```



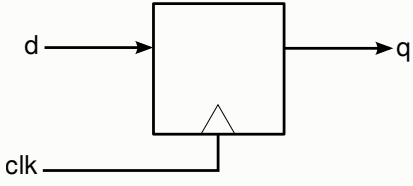
La liste de sensibilité

On peut aussi préciser le type d'événement :

- passsage de `0` à `1` (posedge)
- passsage de `1` à `0` (negedge)

Exemple

```
module mux21(
             input  clk,
             input  d,
             output logic q
            );

always @( posedge clk )
  q <= d;

endmodule
```



La liste de sensibilité
Importance

Une liste incomplète peut entrainer un comportement non désiré.

```
module mux21(
             input  s,
             input  a, b ,
             output logic o
            );

always @(a,b)
  if (s) o = a;
  else   o = b;

endmodule
```

- Si l'entrée s est la seule à changer de valeur, la sortie o gardera sa valeur.
- Ce n'est plus un multiplexeur.

La liste de sensibilité
Liste de sensibilité automatique

Pour éviter d'oublier des éléments de la liste de sensibilité, on peut utiliser la liste de sensibilité automatique « **@*** ».

```
module mux21(
             input  s,
             input  a, b ,
             output logic o
            );

// équivalent à @(s,a,b)
always @(*)
  if (s) o = a;
  else   o = b;

endmodule
```

- La liste de sensibilité contient automatiquement tous les signaux utilisés (lus).

Spécialisation des processus

Pour que le designer précise son intention au moment où il écrit le code, trois versions de `always` existent :

- `always_comb` : pour décrire de la logique combinatoire.
- `always_ff` : pour décrire de la logique séquentielle synchrone.
- `always_latch` : pour décrire des latchs.

Spécialisation des processus

`always@*` *vs.* `always_comb`

Comme **`always@*`** , **`always_comb`** définit aussi automatiquement la liste de sensibilité.

Attention cependant, pour **`always_comb`** , sont exclues de cette liste :

- Les variables qui sont modifiées dans le processus.
- Les variables locales au processus.

Les structures de contrôle

Dans un processus on peut utiliser des structures de contrôle classiques :

- de test (**`if`** , **`else`** , **`case`**)
- de boucle (**`for`** , **`repeat`** , **`while`** , **`forever`**)

Des structures de synchronisation :

- attendre un événement (**`@`**)
- attendre un temps (**`#`**)
- attendre un état (**`wait`**)

Les if

```
...
if (A == 0)
  //<une instruction>
else
  //<une autre>
...
if (A == 0)
begin
  // plusieurs instructions
  // ...
end
```

Les case

```
int V;
...
case (V)
 3 :
     // unse instruction
 4 : begin
     // plusieurs instructions
     ...
 end
 default:
     // Si aucun des cas prévus
endcase
```

La synchronisation

```
...
// attandre un front montant de clk
@(posedge clk) ;
...
// attendre 10 ns
#10ns ;
// attendre 23 unités de temps
#23 ;
// attendre 10 font descendant de clk
repeat(10) @(negedge clk) ;
```

Ceci ne peut être utilisé qu'en simulation.

Les opérateurs

Arithmétiques et logiques

- `+, *, -, /, **, ++, --`
- `&, |, ^, &&, ||`
- `>>, <<`
- `>>>, <<<`

Conditionnel

- `? :`

Comparaison

- `==, != <,<=,>,>=`
- `===, !==`

Autres

concaténation : `{}`

duplication : `{{}}`

Plus de détails section «`11.3 Operators`» de la norme

Plan

Les types de données
Les valeurs binaires

Si on n'a pas besoin de l'état inconnu ou haute impédance on peut utiliser des types à seulement 2 états.

Ils ont l'avantage de rendre les simulations plus rapides et de permettre l'interface avec d'autres langages informatiques.

Mais ils ne permettent pas de vérifier si l'initialisation du système se fait correctement.

On les réserve donc à la simulation.

Les types

shortint	type à 2 états, entier signé 16 bits
int	type à 2 états, entier signé 32 bits
longint	type à 2 états, entier signé 64 bits
byte	type à 2 états, entier signé 8 bits ou caractère ASCII
bit	type à 2 états, taille variable
logic	type à 4 états, taille variable
reg	type à 4 états, taille variable (≡`logic`)
integer	type à 4 états, entier signé 32 bits

Le Signe

Les entiers peuvent être interprétés comme signés ou non signés :

```
int unsigned A;       // entier non signé sur 32bits
logic signed [7:0] B; // entier signé sur 8 bits
```

Le Signe

`shortint`	**signé** par défaut
`int`	**signé** par défaut
`longint`	**signé** par défaut
`byte`	**signé** par défaut
`bit`	**non signé** par défaut
`logic`	**non signé** par défaut
`reg`	**non signé** par défaut
`integer`	**signé** par défaut

Représentation des constantes

Les constantes entières ont la forme suivante :

[signe] [taille ' [s] base] <valeur>

```
22          // entier sur 32 bits
5'd22       // entier de 5bits en décimal
5'b10110    // entier de 5bits en binaire
5'b1_0110   // On peut mettre des _
5'h16       // entier de 5bits en héxadécimal
 'd22       // entier d'une certaine taille  en décimal
...

5'sd22       // entier sur 5 bits qui est interprété comme signé
             // ici -10 !
6'sd22       // entier sur 6 bits qui est interprété comme signé
             // ici +22 !
```

Les tableaux

```
logic [7:0] A [0:255];    // Tableau de 256 mots de 8 bits
logic [7:0] B [256];      // Tableau de 256 mots de 8 bits
logic [7:0] C [0:7][0:7]; // Matrice 8x8 mots de 8 bits
...
logic [31:0] V [0:255];      // Tableau de 256 mots de 32 bits

logic [3:0][7:0] W [0:255];  // Tableau de 256 mots de 32 bits
                             // chaque mot est composé de 4 octets

 W[0]       ...              // le 1e mot de 32 bits
 W[0][3]    ...              // l'octe de poids fort de W[0]
 W[0][3][7] ...              // le bit poids fort de W[0]
```

- Les indices à gauche sont dits pacqués (ceux des bus)
- Les indices à droite sont dits non pacqués (tableaux)

Les tableaux

Affectations pacquées/non pacquées

```
logic [7:0] X [0:3];
...
// affecter des élément de la table
X = '{2,5,6,7};


logic [3:0][7:0] Y;
...
// Concaténer des valeurs
Y = {8'd1,8'd2,8'd2,8'd2};
```

Notez la subtile différence entre l'opérateur de concaténation et l'opérateur pour l'affectation des valeurs d'un tableau.

Les énumérations

SystemVerilog dispose de types énumérés. On les déclare en utilisant le mot clé **enum** .

```
// Sans préciser le type les valeur prises sont des int
// Par défaut rouge=0, vert=1, bleu=2
enum {rouge, vert, bleu} couleur;
...
couleur = vert;
couleur = 3; /* ERREUR !*/
...
if( couleur == vert )
...

// Sur 2 bits 4 valeurs possibles
enum logic[1:0] {HAUT,BAS,GAUCHE,DROITE} dir;
```

Contrairement au C, les **enum** sont fortement typées.

Les types personnalisés

Le mot clef **typedef** permet de définir des types personnalisés.

```
typedef logic[31:0] word;
word a, b;
```

Il n'est cependant pas toujours obligatoire.
Par exemple avec une **enum** .

```
typedef enum {T, F} bool;

enum {IDLE, START, GO, END} state_t;
state_t state, n_state;
```

Les structures et les unions

```
// définition de la structure vec
struct
      {
         logic [3:0] x;
         logic [3:0] y;
      }
      vec1, vec2 ;

// modification du champ a de toto ;
vec1.x = 4'd10;
// affectation directe d'une donnée sous forme de structure
vec2   = vec1;

// v puet être interprété comme un réel (v.r)
//                  ou  comme un entier (v.i)
union {shortreal r; int i;} v ;
```

Les structures «pacquées»

Une structure pacquée est équivalente à un vecteur dont la taille est la somme des tailles de ses membres. Le mot clé **`packed`** doit suivre **`struct`** .

```
// définition de la structure paquée
struct packed {logic [3:0] x;logic [3:0] y;} vec;

logic [7:0] V = 8'h0F;
...
vec = V; // vec.x = 4'h0, vec.y = 4'hF
...
vec = vec + 1;
```

Les membres doivent être des entiers ou des bus (`bit`, `logic`).