HDL et SystemVerilog

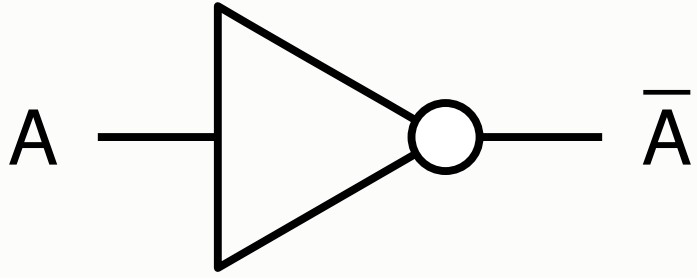
Introduction aux langages de description du matériel

Tarik Graba
tarik.graba@telecom-paristech.fr
Année scolaire 2019/2020

Plan

Représentation de fonctions numériques

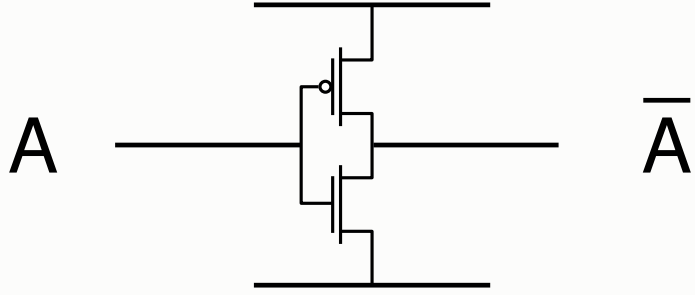
Schéma d'une fonction booléenne



Inverseur

Représentation de fonctions numériques

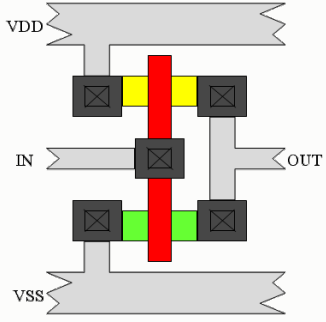
Schéma des transistors



Inverseur CMOS

Représentation de fonctions numériques

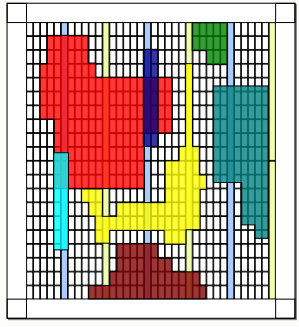
Dessin des couches physiques



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 ?

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.

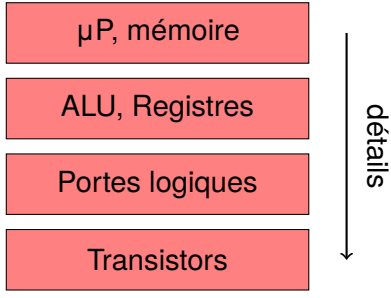
Abstraction et productivité

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

Plan

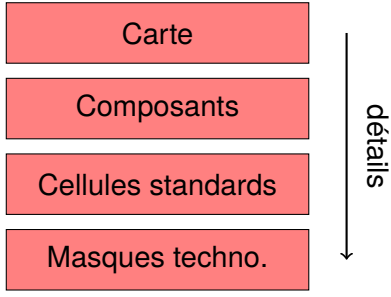
Description structurelle

- Des composants
- Des connexions
- On parle de “netlist”
- …
- Conception par assemblage



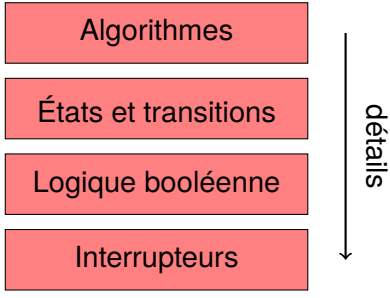
Description physique

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



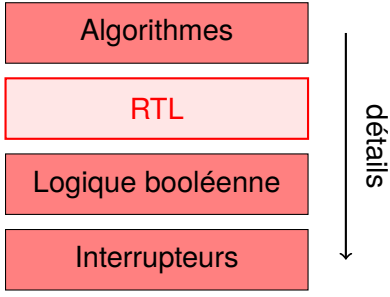
Description comportementale

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



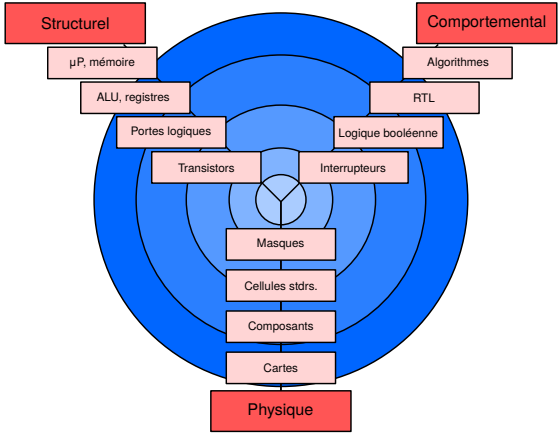
Description comportementale

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



Niveaux de description

Équivalence mais différentes finalités



Utilisation d'outils pour automatiser le passage d'une représentation à l'autre.

Plan

Le “RTL”

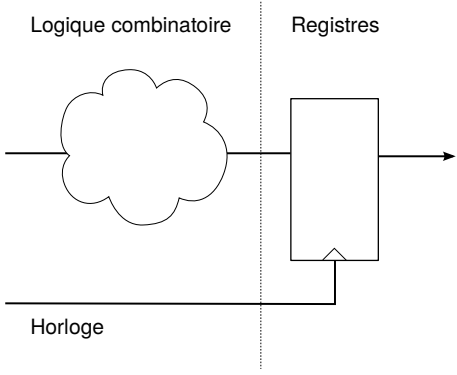
Représentation des états en logique synchrone

RTL

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

Le “RTL”

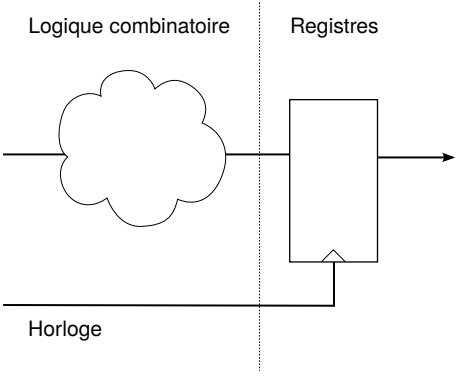
Représentation des états en logique synchrone



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

2- Attribuer des ressources

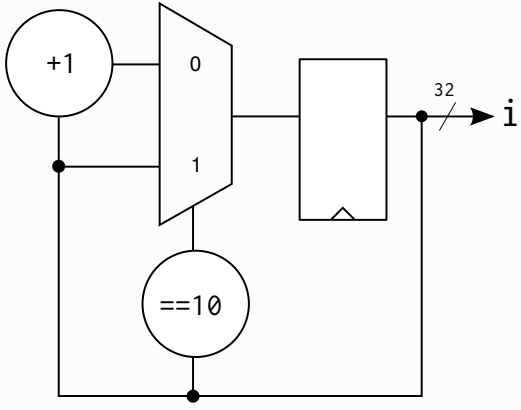
- Un registre pour stocker `i` .
  - Qui change à chaque front d’horloge.
  - Suffisamment grand (32 ou 4 bits ?)
- Un additionneur (ou incrémenteur).
- Un comparateur.

2- Description RTL

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

cond = i<10
sum  = i + 1
@(clk) i = sum[31:0]

// ou plus simplement "@(clk) i = i + 1"
```

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

2- Synthèse automatique



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 des outils automatiques pour passer à des niveaux plus bas
  - Synthèse logique puis physique.

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.

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

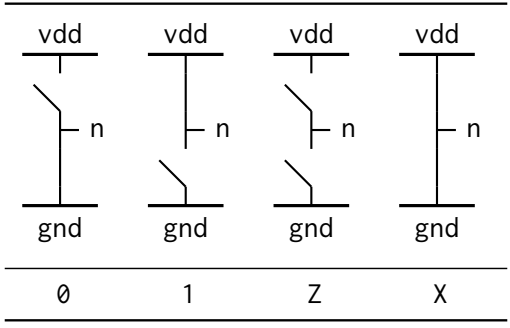
Les valeurs logiques

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 bus

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

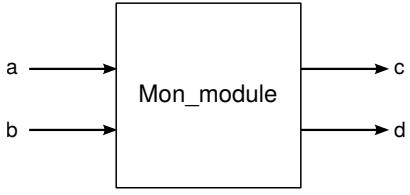
```
wire  [7:0] A      ; // déclare un vecteur de 8 bits de type wire.
wire  [1:8] B      ; // déclare un vecteur de 8 bits de type wire.

... 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



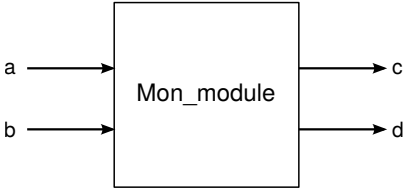
Un module doit avoir :

- 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
```

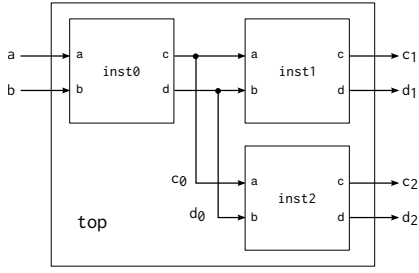
nb. Les `input` / `output` sont par défaut implicitement des `wire` .

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

On utilise des affectations continues pour représenter de la logique combinatoire.

Exemple

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

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

endmodule
```

- o change si a, b ou s change.

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

Les processus

`always` : Permet de décrire de la logique combinatoire ou sequentielle.

`initial` : Réservé à la simulation.

- Les instructions dans un processus sont exécutées les unes après les autres.
- Les processus sont exécutés en ”parallèle”.
- On ne maitrise pas l’ordre d’exécution des processus.

Les variables

Dans un processus on ne peut pas modifier de nœuds. On utilise des variables. En SystemVerilog une variable est de type `logic`. Le type `reg` est gardé par compatibilité avec Verilog.

```
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 affectations

Deux types d'affectations possibles :

- `<=` 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
```

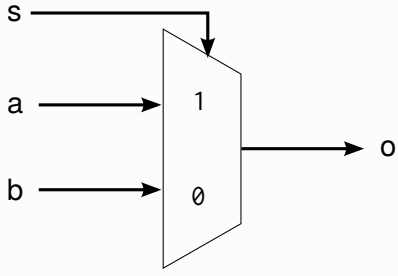
```
// a= 0, b=1, c=2
...
  b = a;
  c = b;

// 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
```



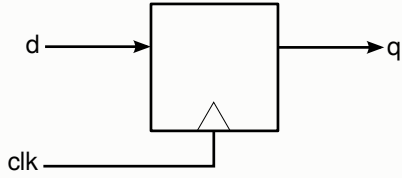
La liste de sensibilité

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).

Les affectations

Alors, différées ou immédiates ?

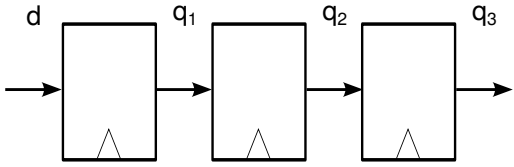
Pour faire simple :

Logique combinatoire $\rightarrow$ affectations immédiates (=)

Logique séquentielle $\rightarrow$ affectations différées (<=)

Les affectations

Alors, différées ou immédiates ?



OK

```
always @(posedge clk)
   q1 <= d;

always @(posedge clk)
   q2 <= q1;

always @(posedge clk)
   q3 <= q2;
```

PAS OK

```
always @(posedge clk)
   q1 = d;

always @(posedge clk)
   q2 = q1;

always @(posedge clk)
   q3 = q2;
```

OK

```
always @(posedge clk)
begin
   q1 <= d;
   q2 <= q1;
   q3 <= q2;
end
```

Les affectations

Alors, différées ou immédiates ?

Garantir que l'ordre d'exécution des processus, en simulation, ne change pas le résultat.
Les affectations peuvent être vues comme faites en parallèle.

OK

```
always @(posedge clk)
   q1 <= d;

always @(posedge clk)
   q2 <= q1;

always @(posedge clk)
   q3 <= q2;
```

PAS OK

```
always @(posedge clk)
   q1 = d;

always @(posedge clk)
   q2 = q1;

always @(posedge clk)
   q3 = q2;
```

OK

```
always @(posedge clk)
begin
   q1 <= d;
   q2 <= q1;
   q3 <= q2;
end
```

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.

Plan

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 (pour la simulation) :

- attendre un événement (`@`)
  - Pour décrire du matériel ne peut être que dans la liste de sensibilité
- attendre un temps (`#`)
  - Exclusivement pour la simulation.
- attendre un état (`wait`)
  - Exclusivement pour la simulation.

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) ;
```

Les opérateurs

Arithmétiques et logiques

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

Conditionnel

- `? :`

Comparaison

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

Autres

concaténation : `{}`

duplication : `{{}}`

Plus de détails section “`11.3 Operators`” de la norme

Plan

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.

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 entiers

Représentation de la forme

[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`).

Plan

Plan

- La liste de sensibilité doit contenir toutes les entrées.
- Les valeurs des sorties doivent être définie pour toutes les valeurs des entrées.

Recommandations

- Liste de sensibilité automatique.
  - `always_comb`
- Donner systématiquement une valeur par défaut aux sorties.

Exercice

Une ALU

- Les entrées sont sur 8 bits.
- Les opérations possibles, la somme, la différence, le et,ou et ou exclusif.
- La sortie est sur 8 bits plus une éventuelle retenue.
- Faire un testbench.

Plan

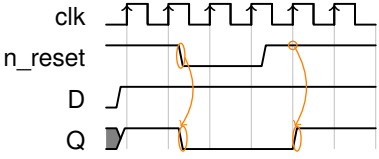
Logique séquentielle

- Dès qu'il faut mémoriser/garder un état on fait de la logique séquentielle synchrone.
- L'état initial vient d'une action extérieure de remise à zéro (reset)

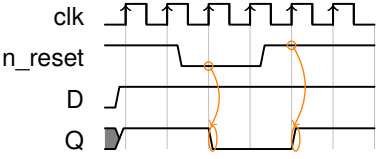
Logique séquentielle

Remise à zéro

Remise à zéro asynchrone :



Remise à zéro synchrone :



Forme générique

Avec remise à zéro synchrone :

```
always_ff @(posedge clk)
if (reset)
 begin
  // Remise à zéro synchrone des registres
  ...
 end
else
 begin
  // Que se passe-t-il à chaque front de l'horloge
  ...
 end
```

Si `reset` vaut 1 au moment du front d'horloge !

Forme générique

Avec remise à zéro asynchrone :

```
always_ff @(posedge clk or posedge reset)
if (reset)
 begin
  // Remise à zéro asynchrone des registres
  ...
 end
else
 begin
  // Que se passe-t-il à chaque front de l'horloge
  ...
 end
```

Si `reset` vaut 1 (dès qu'il passe à 1) indépendamment du front d'horloge.

Forme générique

Avec remise à zéro asynchrone :

```
always_ff @(posedge clk or negedge nreset)
if (!nreset)
 begin
  // Remise à zéro asynchrone des registres
  ...
 end
else
 begin
  // Que se passe-t-il à chaque front de l'horloge
  ...
 end
```

Si `nreset` vaut `0` (dès qu'il passe à `0`) indépendamment du front d'horloge.

Exercice

Compteur

- Un compteur modulo 256 avec remise à zéro synchrone
- Une commande d'activation (enable)
- Un testbench qui génère l'horloge et le reset