



INSTITUT  
Mines-Télécom

# SystemVerilog

Comment décrire du matériel

Tarik Graba

Année scolaire 2016/2017



# Plan

La logique combinatoire

Exemples

La généralité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

## Rappel

- La sortie d'un bloc combinatoire ne dépend que de la valeur de ses entrées.

Dit autrement, pour les mêmes valeurs des entrées on doit **toujours** avoir les mêmes valeurs de sortie.

En SystemVerilog on peut utiliser `always_comb` .

- Équivalent à “`always @(*)`” .
- Le designer précise qu’il veut décrire de la logique combinatoire et les outils le vérifient.

# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

Machines de Mealy

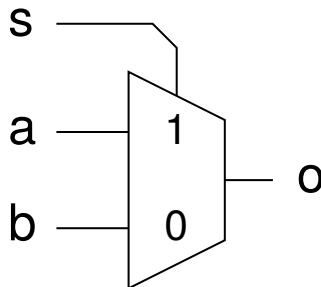
Machines de Mealy resynchronisées

Modélisation des mémoires

# Processus always pour décrire la logique combinatoire

Exemple: Un multiplexeur 2 → 1

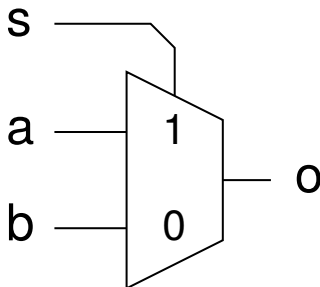
```
module mux21( s, a, b, o );  
  
  input  s;  
  input  a, b ;  
  output reg o ;  
  
  always @(a or b or s)  
    if (s) o = a;  
    else o = b;  
  
  /* Pourrait être  
     o = b;  
     if (s) o = a;  
  * ou  
     o = s? a : b;  
  */  
endmodule
```



# Processus always pour décrire la logique combinatoire

Exemple: Un multiplexeur 2  $\rightarrow$  1

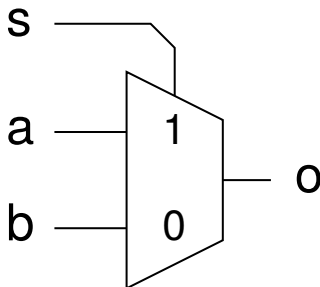
```
module mux21( s, a, b, o );  
  
  input  s;  
  input  a, b ;  
  output reg o ;  
  
  always @(*)  
    if (s) o = a;  
    else  o = b;  
  
  /* Pourrait être  
     o = b;  
     if (s) o = a;  
     * ou  
     o = s? a : b;  
     */  
endmodule
```



# Processus always pour décrire la logique combinatoire

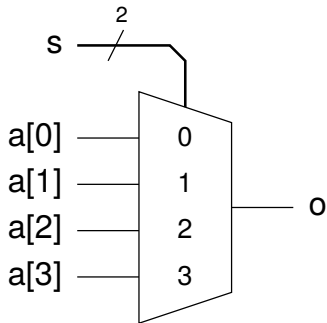
Exemple: Un multiplexeur 2  $\rightarrow$  1

```
module mux21( s, a, b, o );  
  
  input s;  
  input a, b ;  
  output logic o ;  
  
  always_comb  
    if (s) o = a;  
    else o = b;  
  
  /* Pourrait être  
    o = b;  
    if (s) o = a;  
  * ou  
    o = s? a : b;  
  */  
endmodule
```





# Un mux 4→1



```
module mux41( s, a, o );  
  
input [1:0] s;  
input [3:0] a;  
output reg o ;  
  
always @(*)  
    case(s)  
        2'b00: o = a[0];  
        2'b01: o = a[1];  
        2'b10: o = a[2];  
        2'b11: o = a[3];  
    endcase  
  
/* Pourrait être  
   o = a[s];  
   * ou  
   if      (a == 2'd0) o = a[0];  
   else if (a == 2'd1) o = a[1];  
   else if (a == 2'd2) o = a[2];  
   else if (a == 2'd3) o = a[3];  
   */  
  
endmodule
```

# Un mux incomplet

```
module mux_il( s, a, o );  
  
input [1:0] s;  
input [3:0] a;  
output reg o ;  
  
always @( * )  
    case(s)  
        2'b00: o = a[0];  
        2'b01: o = a[1];  
        2'b10: o = a[2];  
    endcase  
  
endmodule
```

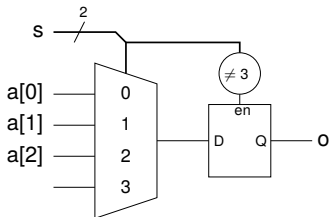
# Un mux incomplet

Que se passe-t-il si  $s = 3$ ?

```
module mux_il( s, a, o );  
  
input [1:0] s;  
input [3:0] a;  
output reg o ;  
  
always @( * )  
    case(s)  
        2'b00: o = a[0];  
        2'b01: o = a[1];  
        2'b10: o = a[2];  
    endcase  
  
endmodule
```

# Un mux incomplet

On mémorise la valeur précédente!!

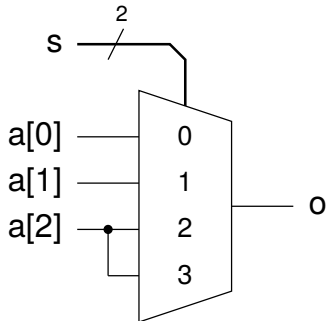


Erreur avec `always_comb`

```
module mux_il( s, a, o );  
  
  input [1:0] s;  
  input [3:0] a;  
  output reg o ;  
  
  always @( * )  
    case(s)  
      2'b00: o = a[0];  
      2'b01: o = a[1];  
      2'b10: o = a[2];  
    endcase  
  
endmodule
```

# Un mux incomplet

Valeurs des sorties toujours définies



```
module mux_i( s, a, o );  
  
input [1:0] s;  
input [3:0] a;  
output reg o ;  
  
always @(*)  
begin  
    // la valeur par défaut  
    o = a[2];  
    case(s)  
        2'b00: o = a[0];  
        2'b01: o = a[1];  
    endcase  
end  
  
/* Pourrait être  
case(s)  
    2'b00: o = a[0];  
    2'b01: o = a[1];  
    default: o = a[2];  
endcase  
*/  
endmodule
```

# Règles pour décrire la logique combinatoire

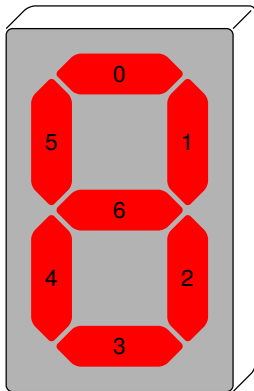
- La liste de sensibilité doit contenir toutes les entrées.
- Les valeurs des sorties doivent être définie pour toutes les valeurs des entrées.

## Recommandations

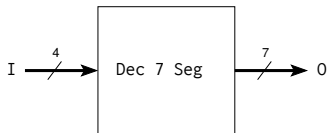
- Liste de sensibilité automatique.
- Donner systématiquement une valeur par défaut aux sorties.

## Exercice

- Écrire le code SystemVerilog d'un décodeur 7 segments
- Écrire le code SystemVerilog d'un décodeur 7 segments qui ne décode que les nombres de 0 à 9



# Un décodeur 7 segments



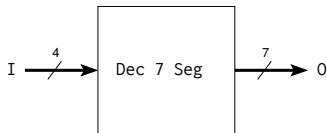
```
module dec7seg ( I, O );
  input  [3:0] I;
  output [6:0] O;
  logic  [6:0] O;

  always_comb
  case(I)
    4'h0: O = 7'b0111111 ;
    4'h1: O = 7'b0000110 ;
    4'h2: O = 7'b1011011 ;
    4'h3: O = 7'b1001111 ;
    4'h4: O = 7'b1100110 ;
    4'h5: O = 7'b1101101 ;
    4'h6: O = 7'b1111101 ;
    4'h7: O = 7'b0000111 ;
    4'h8: O = 7'b1111111 ;
    4'h9: O = 7'b1100111 ;
    4'ha: O = 7'b1110111 ;
    4'hb: O = 7'b1111100 ;
    4'hc: O = 7'b0111001 ;
    4'hd: O = 7'b1011110 ;
    4'he: O = 7'b1111001 ;
    4'hf: O = 7'b1110001 ;
  endcase
endmodule
```



# Un décodeur 7 segments

utilisation d'une table



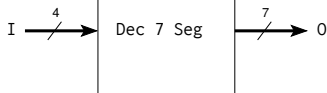
LUT: LookUp Table

```
module dec7segT ( I, O );
  input  [3:0] I;
  output [6:0] O;
  logic  [6:0] O;

  logic [6:0] Tab [0:15] = '{
                                'b0111111 ,
                                'b0000110 ,
                                'b1011011 ,
                                'b1001111 ,
                                'b1100110 ,
                                'b1101101 ,
                                'b1111101 ,
                                'b0000111 ,
                                'b1111111 ,
                                'b1100111 ,
                                'b1110111 ,
                                'b1111100 ,
                                'b0111001 ,
                                'b1011110 ,
                                'b1111001 ,
                                'b1110001 };

  always_comb O = Tab[I];
endmodule
```

# Un décodeur 7 segments incomplet



```
module dec7segI ( I, O );
  input  [3:0] I;
  output [6:0] O;
  logic  [6:0] O;

  always_comb
  case(I)
    4'h0 : O = 7'b0111111 ;
    4'h1 : O = 7'b0000110 ;
    4'h2 : O = 7'b1011011 ;
    4'h3 : O = 7'b1001111 ;
    4'h4 : O = 7'b1100110 ;
    4'h5 : O = 7'b1101101 ;
    4'h6 : O = 7'b1111101 ;
    4'h7 : O = 7'b0000111 ;
    4'h8 : O = 7'b1111111 ;
    4'h9 : O = 7'b1100111 ;
    4'd10,4'd11,
    4'd12,4'd13,
    4'd14,4'd15
      : O = 7'b0000000 ;
  endcase
endmodule
```

# Un décodeur 7 segments incomplet



```
module dec7segI ( I, O );  
  input  [3:0] I;  
  output [6:0] O;  
  logic  [6:0] O;  
  
  always_comb  
  case(I)  
    4'h0 : O = 7'b0111111 ;  
    4'h1 : O = 7'b0000110 ;  
    4'h2 : O = 7'b1011011 ;  
    4'h3 : O = 7'b1001111 ;  
    4'h4 : O = 7'b1100110 ;  
    4'h5 : O = 7'b1101101 ;  
    4'h6 : O = 7'b1111101 ;  
    4'h7 : O = 7'b0000111 ;  
    4'h8 : O = 7'b1111111 ;  
    4'h9 : O = 7'b1100111 ;  
    default: O = 7'b0000000 ;  
  endcase  
  
endmodule
```

# Un décodeur 7 segments incomplet



```
module dec7segI ( I, O );
input  [3:0] I;
output [6:0] O;
logic  [6:0] O;

always_comb
begin
    O = 7'b0000000 ; // valeur par défaut
    case(I)
        4'h0 : O = 7'b0111111 ;
        4'h1 : O = 7'b0000110 ;
        4'h2 : O = 7'b1011011 ;
        4'h3 : O = 7'b1001111 ;
        4'h4 : O = 7'b1100110 ;
        4'h5 : O = 7'b1101101 ;
        4'h6 : O = 7'b1111101 ;
        4'h7 : O = 7'b0000111 ;
        4'h8 : O = 7'b1111111 ;
        4'h9 : O = 7'b1100111 ;
    endcase
end
endmodule
```

# Un décodeur 7 segments incomplet

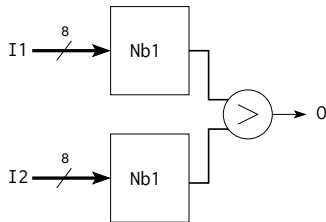
Utilisation de casez



```
module dec7segI ( I, O );  
  input  [3:0] I;  
  output [6:0] O;  
  logic  [6:0] O;  
  
  always_comb  
  casez(I)  
    4'h0 : O = 7'b0111111 ;  
    4'h1 : O = 7'b0000110 ;  
    4'h2 : O = 7'b1011011 ;  
    4'h3 : O = 7'b1001111 ;  
    4'h4 : O = 7'b1100110 ;  
    4'h5 : O = 7'b1101101 ;  
    4'h6 : O = 7'b1111101 ;  
    4'h7 : O = 7'b0000111 ;  
    4'h8 : O = 7'b1111111 ;  
    4'h9 : O = 7'b1100111 ;  
    4'b101?,  
    4'b11??: O = 7'b0000000 ;  
  endcase  
  
endmodule
```

# Utiliser des fonctions

Pour “mutualiser” du code on peut utiliser des modules.



# Utiliser des fonctions

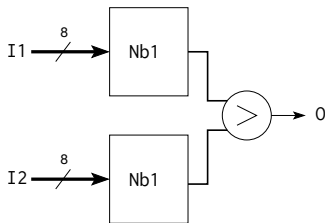
Pour “mutualiser” du code on peut aussi utiliser des fonctions.

```
module foo( I1,I2, O );
input  [7:0] I1,I2;
output logic O;

// Nombre de 1 dans un mot de 8bits
function [3:0] nbr1 (input [7:0] N);
int tmp;
begin
tmp = 0;
for (int i = 0; i<8; i++)
if (N[i]) tmp++;
return tmp;
// On aurait pu écrire
// nbr1 = tmp ;
end
endfunction

always_comb
O = nbr1(I1) > nbr1(I2);

endmodule
```



# Utiliser des fonctions

## différence entre tâches et fonctions

En SystemVerilog il y deux type de sous-programmes:

**Les fonctions:** Exécution en temps nul

- Affectation bloquantes
- Pas de synchronisation (#,@,...)

**Les tâches:** Ne renvoient pas de valeurs

- Peuvent avoir des output

Plus d'informations et des exemples *Section 13* de la norme.



# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

# Faire un code paramétrable

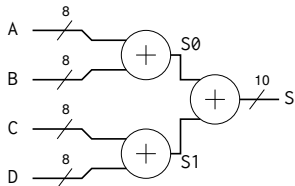
Les modules peuvent être paramétrables:

- Définir leur comportement/structure en fonction de certains paramètres

Le code peut ainsi être réutilisé dans des conditions différentes.

# Faire un code paramétrable

parameter



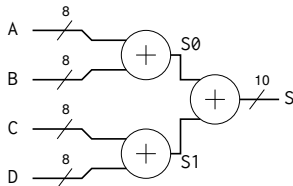
# Faire un code paramétrable

parameter

```
module adder #(parameter WIDTH = 8)
    (input [WIDTH-1:0] A,B,
     output [WIDTH :0] S );

    assign S = A + B;

endmodule
```



# Faire un code paramétrable

parameter

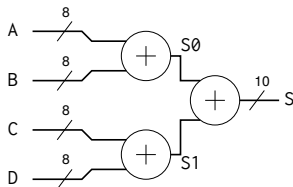
```
module truc ( input [7:0] A,B,C,D,
              output [9:0] S );

  wire [8:0] S0,S1;

  adder #(.WIDTH(8)) add1 (.A(A),.B(B),.S(S0));
  adder #(.WIDTH(8)) add2 (.A(C),.B(D),.S(S1));

  adder #(.WIDTH(9)) add3 (.A(S0),.B(S1),.S(S));

endmodule
```



# Faire un code paramétrable

## localparam

Parfois on a besoin de paramètres non modifiables à l'instanciation.

```
module Table (clk, index, valeur_e, valeur_s);  
  
    parameter SIZE = 256;  
    parameter WIDTH = 8;  
    localparam I_WIDTH = $clog2(SIZE);  
  
    input          clk;  
    input  [I_WIDTH-1:0] index;  
    input  [WIDTH-1:0] valeur_e;  
    output [WIDTH-1:0] valeur_s;  
    logic  [WIDTH-1:0] valeur_s;  
  
    logic [WIDTH-1:0] Tab [0:SIZE-1];  
  
    always_ff @(posedge clk)  
    begin  
        Tab[index] <= valeur_e;  
        valeur_s <= Tab[index];  
    end  
  
endmodule
```

Un paramètre local est:

- Une constante
- Calculé à partir d'autres constantes

Il ne sont pas modifiables à l'extérieur du module.



# Faire un code paramétrable

Comment changer le code en fonction de ces paramètres?

# Faire un code paramétrable

generate

De façon conditionnelle:

```
module adder #( parameter generic = "YES" )
    ( input [7:0] A,B,
      output [7:0] S);

    generate
    if ( generic == "YES")
    begin
        assign S = A + B ;
    end
    else
    begin
        optimised_adder o_adder (A,B,S);
    end
    endgenerate

endmodule
```

Remplace dans ce cas des directives de préprocesseur (`ifdef )



# Faire un code paramétrable

generate

En répétant le comportement

```
// Extrait de la norme section 27.4
module gray2bin #(parameter W = 8)
    ( input [W-1:0] G,
      output [W-1:0] B
    );

    genvar i;
    generate
        for ( i=0; i<W; i++ )
            begin:position

                // Le Xor des bit i à W-1
                assign B[i] = ^G[W-1:i];

            end
        endgenerate
    endmodule
```

# Faire un code paramétrable

generate

En répétant la structure

```
module struct_adder #( parameter W = 8 )
    ( input [W-1:0] A,B, input Ci,
      output [W-1:0] S, output Co,m );

    wire [W:0] c;

    assign c[0] = Ci;
    assign Co = c[W];

    genvar i;
    generate
        for ( i=0; i<W; i++ )
            begin:position
                // ces noeuds seront dupliqués
                wire s, e0, e1;
                xor xor0 ( s , A[i], B[i] );
                xor xor1 ( S[i] , s , c[i] );
                and and0 ( e0 , A[i], B[i] );
                and and1 ( e1 , s , c[i] );
                or or0 ( c[i+1], e0 , e1 );
            end
        endgenerate

    assign m = position[W/2].s;
endmodule
```

# Plan

La logique combinatoire

Exemples

La généricité

**Logique séquentielle synchrone**

Exemples

Machines à états finis

Machines de Moore

Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

# Processus always

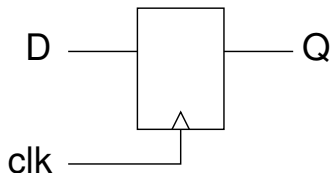
pour de la logique séquentielle synchrone

## La bascule D

```
module Dff ( input clk,
             input D ,
             output reg Q );

always @(posedge clk)
    Q <= D;

endmodule
```



- A chaque front montant (**posedge**) de l'horloge on mémorise la valeur de l'entrée.
- Entre les fronts d'horloge la sortie conserve sa valeur.

En SystemVerilog on peut utiliser `always_ff` .

- Équivalent à “`always`”.
- Le designer précise qu’il veut décrire de la logique séquentielle et les outils le vérifient.

## 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
  end
```

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

## 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.

## 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.



# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

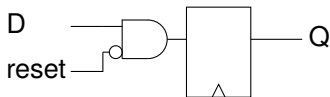
Machines de Moore

Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

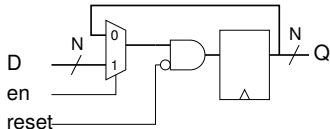
## Une bascule D avec reset synchrone:



```
module Dff ( clk, reset, D, Q );  
    input clk, reset ;  
    input D ;  
    output reg Q ;  
  
    always_ff @(posedge clk)  
        if (reset)  
            Q <= 'b0;  
        else  
            Q <= D;  
  
endmodule
```

# Un registre :

Avec reset synchrone et enable

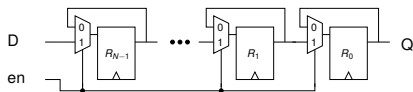


```
module Reg( input clk, reset, en,
            input [7:0] D,
            output logic [7:0] Q );

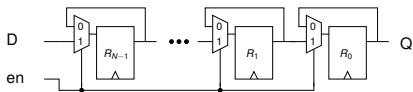
always_ff @(posedge clk)
    if (reset)
        Q <= 8'd0;
    else
        if (en) Q <= D;

endmodule
```

# Un registre à décalage:



# Un registre à décalage:



```
module SftReg # ( parameter N = 8 )
    ( input clk, en ,
      input D ,
      output Q );

    logic [N-1:0] R;

    assign Q = R[0];

    always_ff @(posedge clk)
        if (en) R <= { D, R[N-1:1] };

endmodule
```

# Astuce

mettre la même valeur à tous les bits d'un vecteur

## Opérateur de duplication

```
reg [N-1:0] A,B,C;  
  
initial  
begin  
  A = {N{1'b1}}; // tous les bits à 1  
  B = {N{1'b0}}; // tous les bits à 0  
  C = {N{1'bz}}; // tous les bits à z  
  ...  
end
```

# Astuce

mettre la même valeur à tous les bits d'un vecteur

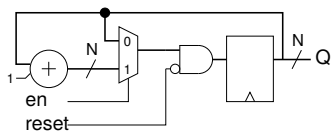
## Opérateur de duplication

```
reg [N-1:0] A,B,C;  
  
initial  
begin  
  A = {N{1'b1}}; // tous les bits à 1  
  B = {N{1'b0}}; // tous les bits à 0  
  C = {N{1'bz}}; // tous les bits à z  
  ...  
end
```

## Magie!

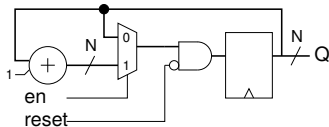
```
logic [N-1:0] A,B,C,D;  
  
initial  
begin  
  A = '1; // tous les bits à 1  
  B = '0; // tous les bits à 0  
  C = 'z; // tous les bits à z  
  ...  
  D = 'd1 // 1 en décimal adapté  
           // à la taille de D!end
```

# Un compteur :





# Un compteur :



```
module Cpt ( clk, reset, en, Q );  
  
parameter N = 8;  
  
input clk, reset, en ;  
output logic [N-1:0] Q ;  
  
always_ff @(posedge clk)  
    if (reset)  
        Q <= '0;  
    else  
        if (en)  
            Q <= Q + 1;  
  
endmodule
```

# Affectations bloquantes/différées

Quelle est la différence entre ces deux codes:

```
logic a,b,r,q;

always_ff @(posedge clk)
begin
  r = a & b;
  q <= r;
end
```

```
logic a,b,r,q;

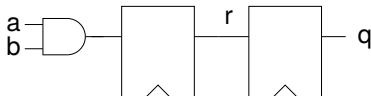
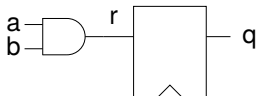
always_ff @(posedge clk)
begin
  r <= a & b;
  q <= r;
end
```

# Affectations bloquantes/différées

Quelle est la différence entre ces deux codes:

```
logic a,b,r,q;  
  
always_ff @(posedge clk)  
begin  
    r = a & b;  
    q <= r;  
end
```

```
logic a,b,r,q;  
  
always_ff @(posedge clk)  
begin  
    r <= a & b;  
    q <= r;  
end
```



# Affectations bloquantes/différées

Et ici ?

```
logic [N:0] R;

always_ff @(posedge clk)
begin: loop
  int i;
  for (i=0; i<N; i++)
    R[i+1] = R[i];
end
```

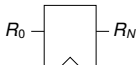
```
logic [N:0] R;

always_ff @(posedge clk)
begin: loop
  int i;
  for (i=0; i<N; i++)
    R[i+1] <= R[i];
end
```

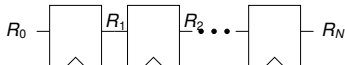
# Affectations bloquantes/différées

Et ici ?

```
logic [N:0] R;  
  
always_ff @(posedge clk)  
begin: loop  
  int i;  
  for (i=0; i<N; i++)  
    R[i+1] = R[i];  
end
```



```
logic [N:0] R;  
  
always_ff @(posedge clk)  
begin: loop  
  int i;  
  for (i=0; i<N; i++)  
    R[i+1] <= R[i];  
end
```



# Affectations bloquantes/différées

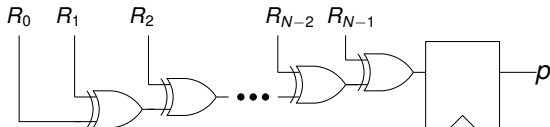
Que fait ce code?

```
logic [N-1:0] R;  
logic p;  
  
always_ff @(posedge clk)  
begin: loop  
  int i;  
  logic t;  
  t = 0;  
  for (i=0; i<N; i++)  
    t = t ^ R[i];  
  p <= t;  
end
```

# Affectations bloquantes/différées

Que fait ce code ?

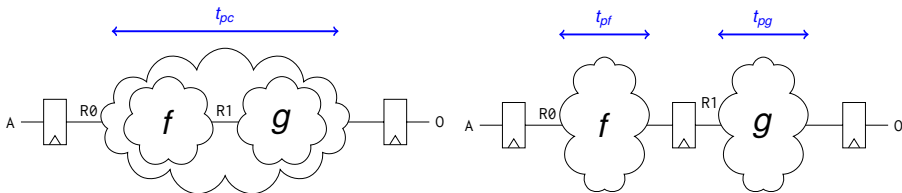
```
logic [N-1:0] R;  
logic p;  
  
always_ff @(posedge clk)  
begin: loop  
  int i;  
  logic t;  
  t = 0;  
  for (i=0; i<N; i++)  
    t = t ^ R[i];  
  p <= t;  
end
```



# Chemin critique/Pipeline

```
always_ff @(posedge clk)
begin
  R0 <= A;
  ...
  R1 = f(R0);
  ...
  O <= g(R1);
  ...
end
```

```
always_ff @(posedge clk)
begin
  R0 <= A;
  ...
  R1 <= f(R0);
  ...
  O <= g(R1);
  ...
end
```





# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

**Machines à états finis**

Machines de Moore

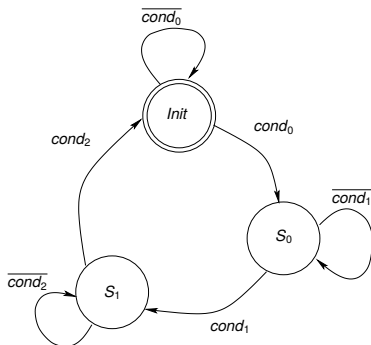
Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

# Machines à états finis

- Méthode pour concevoir des automates.
- A partir d'un graphe d'états.
- Système synchrone.



# Déclaration du registre d'état et des états

## En Verilog 95

```
`define INIT 2'b00
`define S0 2'b01
`define S1 2'b10

reg [1:0] state, n_state;
//...Le code

`undef INIT
`undef S0
`undef S1
```

# Déclaration du registre d'état et des états

## En Verilog 2001

```
localparam INIT = 2'b00;  
localparam S0  = 2'b01;  
localparam S1  = 2'b10;  
  
reg [1:0] state, n_state;  
//...Le code
```

# Déclaration du registre d'état et des états

## En SystemVerilog

```
enum logic[1:0] { INIT, S0, S1 } state, n_state;  
//...Le code
```

# Modification de l'état

## Synchrone

```
always_ff @(posedge clk)
if (reset)
    state <= INIT ;
else
    state <= n_state ;
```

- L'état initial au reset doit être explicite.
  - Si l'état initial n'est pas connu le comportement n'est pas déterministe.
- Le changement d'état se fait de façon synchrone.

# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

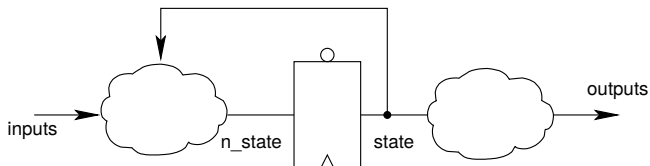
Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

# Machine de Moore

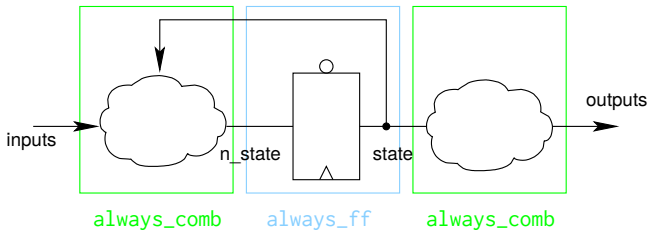
- Il faut mémoriser l'état (registre).
- Le prochain état dépend de l'état actuel et des entrées.
- Les sorties dépendent combinatoirement de l'état courant.





## 3 processus

- Un processus séquentiel pour sauvegarder l'état.
- Deux processus combinatoires:
  - Calcul de l'état futur.
  - Calcul des sorties.
- `n_state` doit être un signal.

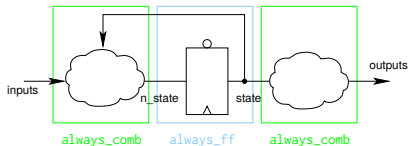


# 3 processus

```
always_comb
begin
    // par défaut on reste
    // dans l'état courant
    n_state = state ;
    case (state)
        INIT: if (cond0)
            n_state = S0;
        S0  : if (cond1)
            n_state = S1;
        S1  : if (cond2)
            n_state = INIT;
    endcase
end
```

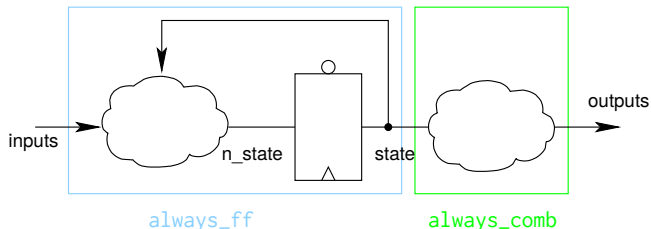
```
always_ff @(posedge clk)
if (reset)
    state <= INIT ;
else
    state <= n_state ;
```

```
always_comb
begin
    if (state == INIT) begin
        output1 = ...
    end
    else if (state == S0) begin
        output1 = ...
    end
    else if (state == S0) begin
        output1 = ...
    end
    else begin // Par défaut
        output1 = ...
    end
end
```



## 2 processus

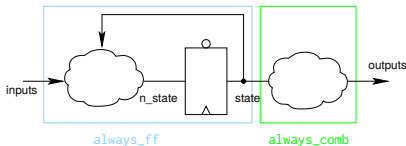
- Un processus séquentiel pour modifier l'état.
- Un processus combinatoire pour le calcul des sorties en fonction de l'état.
- `n_state` disparaît.



## 2 processus

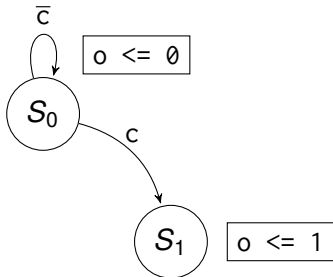
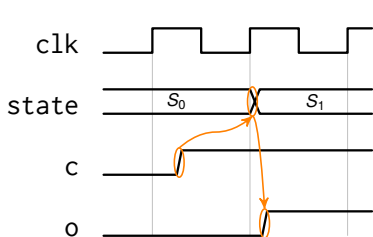
```
always_ff @(posedge clk)
if (reset)
    state <= INIT ;
else
    case (state)
        INIT: if (cond0)
            state <= S0;
        S0  : if (cond1)
            state <= S1;
        S1  : if (cond2)
            state <= INIT;
    endcase
// Sinon on reste dans
// l'état courant
```

```
always_comb
begin
    if (state == INIT) begin
        output1 = ...
    end
    else if (state == S0) begin
        output1 = ...
    end
    else if (state == S0) begin
        output1 = ...
    end
    else begin // Par défaut
        output1 = ...
    end
end
```



# Inconvénient

Un changement de sortie nécessite un changement d'état et donc au moins un cycle de latence.



# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

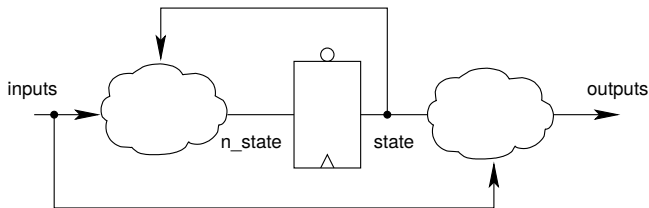
Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

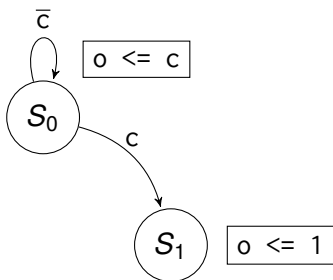
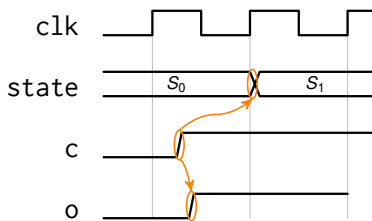
# Machines de Mealy

- Il faut mémoriser l'état (registre).
- Le prochain état dépend de l'état actuel et des entrées.
- Les sorties dépendent combinatoirement de l'état courant et des entrées.



# Inconvénient

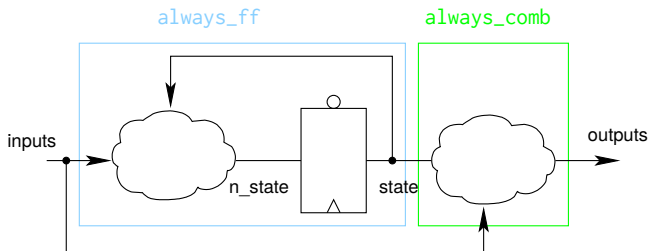
Un changement d'entrée peut être propagé immédiatement sur une sortie.





## 2 processus

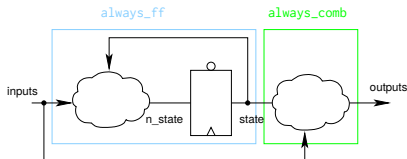
- Un processus séquentiel pour modifier l'état.
- Un processus combinatoire pour le calcul des sorties en fonction de l'état et des entrées.



## 2 processus

```
always_ff @(posedge clk)
if (reset)
    state <= INIT ;
else
    case (state)
        INIT: if (cond0)
            state <= S0;
        S0  : if (cond1)
            state <= S1;
        S1  : if (cond2)
            state <= INIT;
    endcase
// Sinon on reste dans
// l'état courant
```

```
always_comb
begin
    if (state == INIT) begin
        output1 = f1(inputs ...)
    end
    else if (state == S0) begin
        output1 = f2(inputs ...)...
    end
    else if (state == S0) begin
        output1 = f3(inputs ...)
    end
    else begin // Par défaut
        output1 = f4(inputs ...)
    end
end
```



# Inconvénient

On relie les entrées et les sorties par un chemin combinatoire:

- Le chemin critique n'est pas maîtrisé.
- La modification de la MAE modifie les performances du reste du circuit.

# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

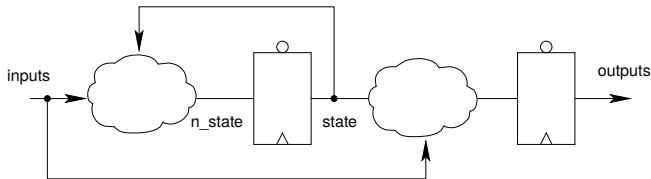
Machines de Mealy

Machines de Mealy resynchronisées

Modélisation des mémoires

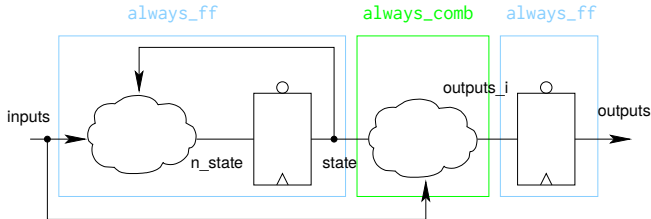
# Machines de Mealy resynchronisées

- C'est une machine de Mealy pour la quelle les sorties sont resynchronisées pour éviter les chemins combinatoires.
- Il faut ajouter des registres sur les sorties



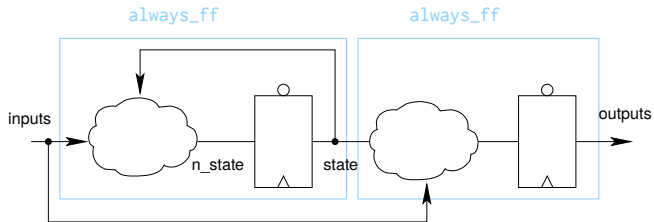
## 3 processus

- Un processus séquentiel pour modifier l'état.
- Un processus combinatoire pour le calcul des sorties en fonction de l'état et des entrées.
- Faire apparaître un signal interne pour les sorties avant resynchronisation.
- Un processus séquentiel pour resynchroniser les sorties.



## 2 processus

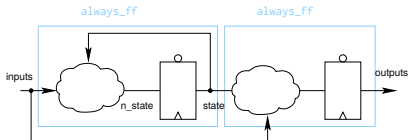
- Un processus séquentiel pour modifier l'état.
- Un séquentiel pour le calcul des sorties



## 2 processus

```
always_ff @(posedge clk)
if (reset)
    state <= INIT ;
else
    case (state)
    INIT: if (cond0)
        state <= S0;
    S0  : if (cond1)
        state <= S1;
    S1  : if (cond2)
        state <= INIT;
    endcase
// Sinon on reste dans
// l'état courant
```

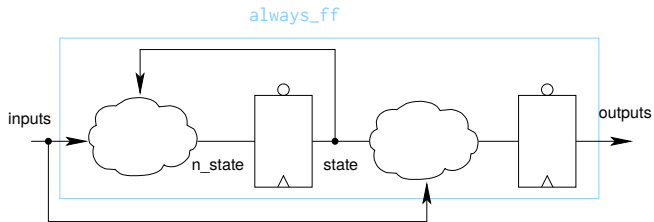
```
always_ff @(posedge clk)
if (reset)
    begin
        // initialiser les sorties
        output1 <= ...
    end
else
    begin
        if (state == INIT) begin
            output1 <= f1(inputs ...)
        end
        else if (state == S0) begin
            output1 <= f2(inputs ...)...
        end
        else if (state == S1) begin
            output1 <= f3(inputs ...)
        end
    end
end
```





# 1 processus

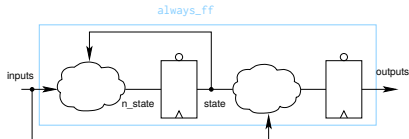
- Un processus séquentiel pour modifier l'état et les sorties.



# 1 processus

```
always_ff @(posedge clk)
if (reset)
begin
// Initialiser l'état
state <= INIT ;
// Initialiser les sorties
output1 <= ...
end
else
begin
// Les transitions
case (state)
INIT: if (cond0)
state <= S0;
S0 : if (cond1)
state <= S1;
...
end
```

```
...
S1 : if (cond2)
state <= INIT;
endcase
// Les sorties
if (state == INIT) begin
output1 <= f1(inputs ...)
end
else if (state == S0) begin
output1 <= f2(inputs ...)...
end
else if (state == S0) begin
output1 <= f3(inputs ...)
end
end
```



# Plan

La logique combinatoire

Exemples

La généricité

Logique séquentielle synchrone

Exemples

Machines à états finis

Machines de Moore

Machines de Mealy

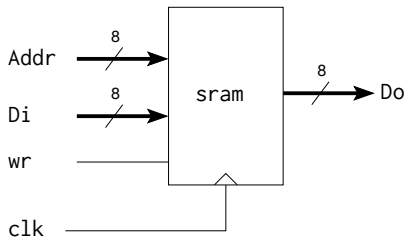
Machines de Mealy resynchronisées

Modélisation des mémoires

# Mémoire synchrone

## mémoire simple port

- un bus d'adresse
- 2 bus pour les données:
  - écriture
  - lecture
- des signaux de contrôle
- une **horloge**
- **PAS DE RESET**
- On ne peut accéder qu'à un seul élément dans le même cycle!



# Mémoire synchrone

## mémoire simple port

- un bus d'adresse
- 2 bus pour les données:
  - écriture
  - lecture
- des signaux de contrôle
- une horloge
- **PAS DE RESET**
- On ne peut accéder qu'à un seul élément dans le même cycle!

```
module sram(input clk, wr,
            input [7:0] Addr,
            input [7:0] Di,
            output logic [7:0] Do );

    logic[7:0] mem [0:255];

    always_ff @(posedge clk)
    begin
        if (wr)
            mem[Addr] <= Di;
        Do <= mem[Addr];
    end

endmodule
```

# Mémoire synchrone

## mémoire simple port

- un bus d'adresse
- 2 bus pour les données:
  - écriture
  - lecture
- des signaux de contrôle
- une horloge
- **PAS DE RESET**
- On ne peut accéder qu'à un seul élément dans le même cycle!

```
module sram(input clk, wr,
            input [7:0] Addr,
            input [7:0] Di,
            output [7:0] Do );

    logic[7:0] mem [0:255];
    logic[7:0] Addr_r;

    always_ff @(posedge clk)
    begin
        if (wr)
            mem[Addr] <= Di;
            Addr_r <= Addr;
    end

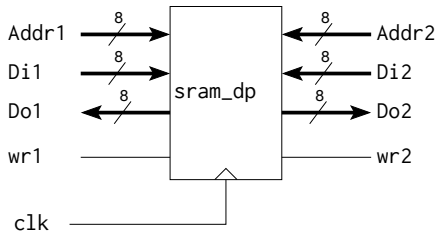
    assign Do = mem[Addr_r];

endmodule
```

# Mémoire synchrone

## mémoire double ports

- permet un accès double
- l'écriture et la lecture à la même adresse dans le même cycle n'est pas prédictible
- pourrait avoir deux horloges



# Mémoire synchrone

## mémoire double ports

- permet un accès double
- l'écriture et la lecture à la même adresse dans le même cycle n'est pas prédictible
- pourrait avoir deux horloges

```
module sram_dp(input clk, wr1, wr2,
               input [7:0] Addr1, Addr2,
               input [7:0] Di1, Di2,
               output logic [7:0] Do1, Do2 );

logic[7:0] mem [0:255];

always_ff @(posedge clk)
begin
    if (wr1)
        mem[Addr1] <= Di1;
    Do1 <= mem[Addr1];
end

always_ff @(posedge clk)
begin
    if (wr2)
        mem[Addr2] <= Di2;
    Do2 <= mem[Addr2];
end

endmodule
```



# Mémoire synchrone

## Initialisation du contenu

- Possible seulement pour les FPGA
- **initial** est normalement exclusivement réservé à la simulation
- **\$readmemh** (ou **\$readmemb**) permet d'initialiser une table à partir d'un fichier

```
module sram(input clk, wr,
           input [7:0] Addr,
           input [7:0] Di,
           output logic [7:0] Do );

logic[7:0] mem [0:255];

initial
    $readmemh("init.txt", mem);

always_ff @(posedge clk)
begin
    if (wr)
        mem[Addr] <= Di;
    Do <= mem[Addr];
end

endmodule
```

# Mémoire synchrone

## ROM synchrone

- Possible seulement pour les FPGA
- il suffit d'enlever la possibilité d'écrire

```
module rom (input clk,
            input [7:0] Addr,
            output logic [7:0] Do );

logic[7:0] mem [0:255];

initial
    $readmemh("init.txt", mem);

always_ff @(posedge clk)
    Do <= mem[Addr];

endmodule
```