

# UE VLSI

## cours 7: Machines à état, détail de DECOD

Jean-Lou Desbarbieux  
UPMC 2017

# Sommaire

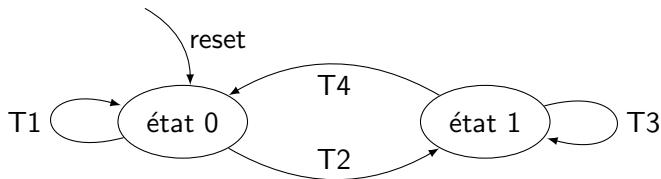
① Machines à état

② Étage DECOD

# Principe d'une machine à état

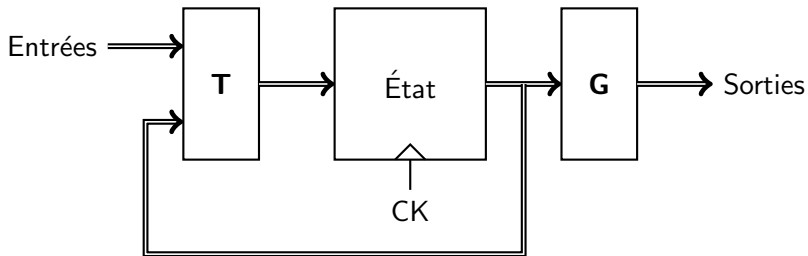
Une machine à état c'est :

- **états** : souvenir du passé.
- **transitions** : comment passer d'un état à un autre.
- **entrées**
- **sorties**



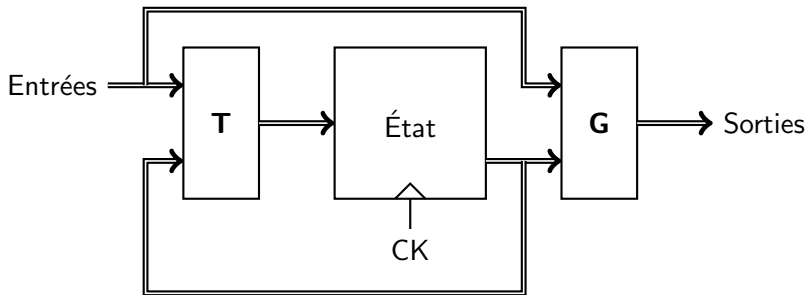
# Machine de Moore

Les sorties ne dépendent que de l'état, mieux pour le contrôle des temps de propagation !



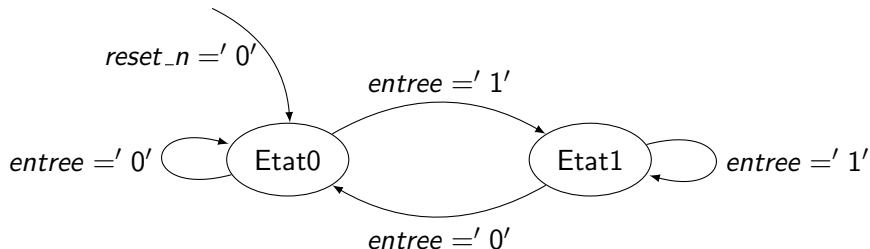
# Machine de Mealy

Les sorties dépendent de l'état et des entrées.



## Exemple

On veut concevoir une machine à état qui détecte un front montant sur son entrée et génère un créneau sur sa sortie.



## Description *VHDL* d'une machine à état

Définition du type des états et déclaration des signaux :

```
type state_type is (Etat0 , Etat1);  
signal cur_state , next_state : state_type;
```

Description du registre d'état :

```
process (ck)  
begin  
  if (rising_edge(ck)) then  
    if (reset_n = '0') then  
      cur_state <= Etat0;  
    else  
      cur_state <= next_state;  
    end if;  
  end if;  
end process;
```

## Description *VHDL* d'une machine à état (2)

Définition de la fonction de transition :

```
process (entree)
begin
    case cur_state is
    when Etat0 =>
        if entree = '1' then
            next_state <= Etat1;
        else
            next_state <= Etat0;
        end if;
    when Etat1 =>
        if entree = '0' then
            next_state <= Etat0;
        else
            next_state <= Etat1;
        end if;
    end case;
end process;
```



## Description *VHDL* d'une machine à état (3)

La génération des sorties, peut être dans le même process que la transition.  
Cas d'une machine de Moore :

```
case cur_state is  
when Etat0 =>  
    sortie <= '0';  
    if entree = '1' then  
        next_state <= Etat1;  
    else  
        next_state <= Etat0;  
    end if;
```

...

## Description *VHDL* d'une machine à état (4)

Cas d'une machine de Mealy :

```
case cur_state is  
when Etat0 =>  
    if entree = '1' then  
        next_state <= Etat1;  
        sortie <= '1';  
    else  
        next_state <= Etat0;  
        sortie <= '0';  
    end if;
```

...

# Fonctionnalités de DECOD

En plus d'héberger le banc de registre, le bloc/étage DECOD doit assurer deux principales fonctionnalités :

- Décoder les instructions pour permettre leur exécution par les étages EXEC et MEM.
- Assurer le séquençement du pipeline, gestion des aléas et traitement des instructions multi cycles.

# Décodage des instructions

L'instruction reçue de la mémoire est codée de façon compacte (32 bits), il va falloir la traduire en un mot très large (127 bits) qui va être interprété par EXEC et/ou MEM. Beaucoup de signaux de DECOD vont être utiles :

**signal** cond : Std\_Logic;

**signal** condv : Std\_Logic;

**signal** operv : Std\_Logic;

**signal** regop\_t : Std\_Logic;

**signal** mult\_t : Std\_Logic;

**signal** swap\_t : Std\_Logic;

**signal** trans\_t : Std\_Logic;

**signal** mtrans\_t : Std\_Logic;

**signal** branch\_t : Std\_Logic;

## Décodage des instructions(2) regop

```
signal and_i    : Std_Logic;  
signal eor_i    : Std_Logic;  
signal sub_i    : Std_Logic;  
signal rsb_i    : Std_Logic;  
signal add_i    : Std_Logic;  
signal adc_i    : Std_Logic;  
signal sbc_i    : Std_Logic;  
signal rsc_i    : Std_Logic;  
signal tst_i    : Std_Logic;  
signal teq_i    : Std_Logic;  
signal cmp_i    : Std_Logic;  
signal cmn_i    : Std_Logic;  
signal orr_i    : Std_Logic;  
signal mov_i    : Std_Logic;  
signal bic_i    : Std_Logic;  
signal mvn_i    : Std_Logic;
```

## Décodage des instructions(3)

— *trans instruction*

**signal** ldr\_i : Std\_Logic;

**signal** str\_i : Std\_Logic;

**signal** ldrb\_i : Std\_Logic;

**signal** strb\_i : Std\_Logic;

— *mtrans instruction*

**signal** ldm\_i : Std\_Logic;

**signal** stm\_i : Std\_Logic;

— *branch instruction*

**signal** b\_i : Std\_Logic;

**signal** bl\_i : Std\_Logic;

# Commande de EXEC

— *Exec operands*

```
signal op1      : Std_Logic_Vector(31 downto 0)
signal op2      : Std_Logic_Vector(31 downto 0)
signal alu_dest  : Std_Logic_Vector(3 downto 0)
signal alu_wb   : Std_Logic;
signal flag_wb  : Std_Logic;
```

## Commande de EXEC(2)

— *Shifter command*

**signal** shift\_lsl : Std\_Logic;

**signal** shift\_lsr : Std\_Logic;

**signal** shift\_asr : Std\_Logic;

**signal** shift\_ror : Std\_Logic;

**signal** shift\_rrx : Std\_Logic;

**signal** shift\_val : Std\_Logic\_Vector(4 **downto** 0)

**signal** cy : Std\_Logic;

— *Alu operand selection*

**signal** comp\_op1 : Std\_Logic;

**signal** comp\_op2 : Std\_Logic;

**signal** alu\_cy : Std\_Logic;

— *Alu command*

**signal** alu\_cmd : Std\_Logic\_Vector(1 **downto** 0)



# Commande de MEM

— *Decod to mem via exec*

```
signal mem_data : Std_Logic_Vector(31 downto 0);  
signal ld_dest  : Std_Logic_Vector(3  downto 0);  
signal pre_inde : Std_logic;
```

```
signal mem_lw   : Std_Logic;  
signal mem_lb   : Std_Logic;  
signal mem_sw   : Std_Logic;  
signal mem_sb   : Std_Logic;
```

# Séquencement/contrôle du pipeline

```
signal dec2if_push    : Std_Logic;  
signal dec2exe_push   : Std_Logic;  
signal if2dec_pop     : Std_Logic;
```

# Une machine à état pour DECOD

