C7 – Les Process synchrones

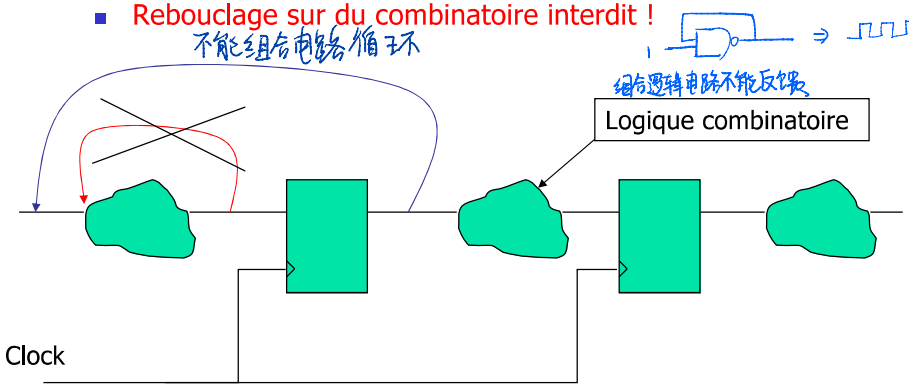
Yann DOUZE
VHDL

Design Synchrone

寄存器都在Dflip-flop中设置，只需一个时钟

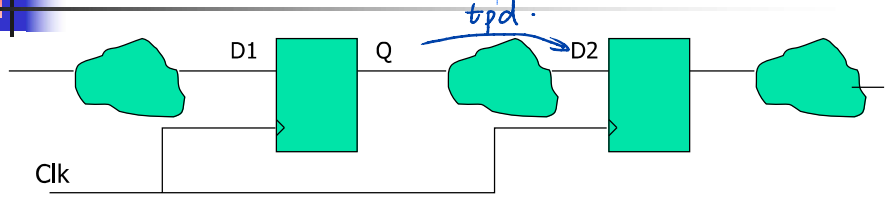
- Tous les registres se font dans des bascules D flip-flops avec une seule horloge externe.
- Rebouclage sur du combinatoire interdit !

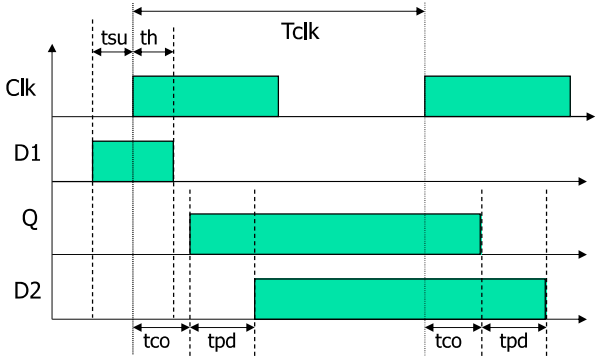
oscillateur 晶振(振荡器)



Contraintes de temps

通过逻辑电路时间





Tclk: période de l'horloge 时钟周期

tsu: time setup

th: time hold

tco: clock to output

depend Techo

tpd: propagation delay

Tclk > tco + tpd + tsu

Fmax = 1 / (tco + tsu + tpd)

1. **Tclk：时钟周期**
   - **定义**：时钟信号从一个上升沿（或下降沿）到下一个相同沿的时间间隔。
   - **作用**：时钟周期决定了电路的工作频率，它必须足够长，以确保信号从寄存器传播并稳定在下一级寄存器之前。
2. **Tsu：设置时间（Setup Time）**
   - **定义**：时钟上升沿到来之前，输入数据需要在有效状态持续的最小时间，即数据需要在时钟沿前稳定并保持足够长时间。
   - **作用**：为了保证数据能够正确被采样并传送到下一级电路，必须满足此设置时间。如果输入数据变化太快而不满足这个时间约束，会导致采样错误。
3. **Th：保持时间（Hold Time）**
   - **定义**：时钟上升沿到来之后，输入数据仍然需要保持稳定的最小时间。
   - **作用**：保持时间是为了确保时钟采样后，输入数据不会立即变化，从而避免数据错误采样。因此，时钟沿之后数据也需要在此时间内保持稳定。
4. **Tco：时钟到输出延迟（Clock to Output Delay）**
   - **定义**：时钟沿到来后，从寄存器输出信号开始变化所需的时间。
   - **作用**：这是从时钟触发到数据开始输出的延迟，表示寄存器内部逻辑将数据从输入端传输到输出端所需的时间。
5. **Tpd：传播延迟（Propagation Delay）**
   - **定义**：信号从一个逻辑电路传播到另一个逻辑电路的延迟时间。
   - **作用**：当信号从寄存器输出后，通过组合逻辑电路传递到下一级寄存器的延迟时间。此时间包括了电路中的门延迟和传输延迟。

关键时序公式：

- **时钟周期约束**：

$$T_{clk} > T_{co} + T_{pd} + T_{su}$$

意味着时钟周期必须足够长，能容纳寄存器的时钟到输出延迟、组合逻辑传播延迟以及下一级寄存器的设置时间。否则，电路将无法正常工作。

- **最大工作频率**：

$$F_{max} = \frac{1}{T_{co} + T_{su} + T_{pd}}$$

这个公式表示电路能运行的最大时钟频率，由时钟到输出延迟、设置时间和传播延迟共同决定。减少这些时间可以提高电路的工作频率。

Process synchrone

同步进程发生在边沿触发时刻.

```
process(Reset,Clk)
begin
if Reset = '1' then
    Q1 <= '0';
elsif RISING_EDGE(Clk) then
    Q1 <= D;
end if;
end process;
```

- **Un process synchrone est exécuté à chaque front montant de l'horloge.**
- Pour tester le front montant de l'horloge, on utilise la fonction rising_edge() qui est définit dans le package STD_LOGIC_1164.
- RISING_EDGE est VRAI lorsque l'horloge passe de l'état '0' à l'état '1'.
- Utile pour décrire une bascule D flip-flop (Dff).

Style de code légal utilisant 'EVENT

- S'EVENT est vrai si est seulement si il y a un évènement sur S

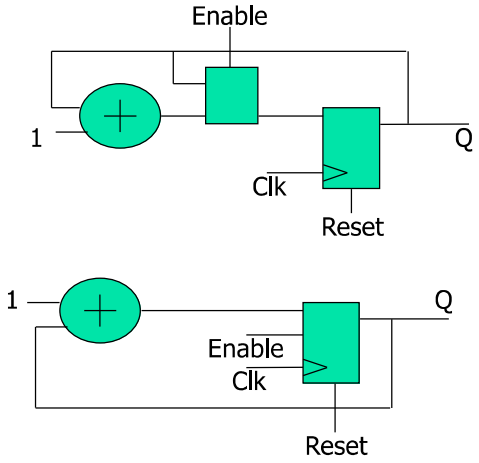
```
process(Clk, rst)
Begin
   if rst = '1' then
      Q <='0';
   elsif Clk'EVENT and Clk = '1' then
      Q <= D;
   end if;
end process;
```

= rising_edge(clk) in VHDL 1993

```
process
begin
   wait until Clk'EVENT and Clk = '1';
      ...
end process;
```

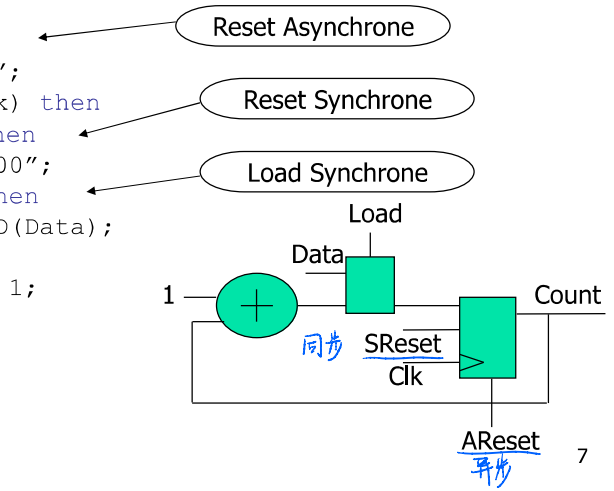

Clock Enables

```
process (Clk, Reset)
begin
  if Reset = '1' then
    Q <= "00000000";
  elsif RISING_EDGE(Clk) then
    if  Enable = '1' then
      Q <= Q + 1;
    end if;
  end if;
end process;
```



Actions Synchrones et Asynchrones

```
signal Count : unsigned(7 downto 0);
process (Clk, AReset)
begin
  if AReset = '1' then
    Count <= "00000000";
  elsif RISING_EDGE(Clk) then
    if  SReset = '1' then
      Count <= "00000000";
    elsif Load = '1' then
      Count <= UNSIGNED(Data);
    else
      Count <= Count + 1;
    end if;
  end if;
end process;
```



Compteur 8 bits

```
entity COUNTER8BIT is
  port ( CLK, RST : in     Std_logic;
         Q        : out    Std_logic_vector( 7 downto 0));
end entity;
architecture RTL of COUNTER8BIT is
  signal CNT: unsigned( 7 downto 0);
begin
  process (CLK,RST)
  begin
    if RST = '1' then
      CNT <= "00000000";
    elsif rising_edge(CLK) then
      CNT <= CNT + 1;
    end if;
  end process;
  Q <= std_logic_vector(CNT);
end architecture;
```


Compteur génériques

调用组件

```
entity COUNTER is
generic(N : integer:=8);
port ( CLK, RST : in     Std_logic;
       Q        : out    Std_logic_vector(N-1 downto 0));
end entity;
architecture RTL of COUNTER is
  signal CNT: unsigned(N-1 downto 0);
begin
  process (CLK,RST)
  begin
    if RST = '1' then
      CNT <= (others => '0');
    elsif rising_edge(CLK) then
      CNT <= CNT + '1';
    end if;
  end process;
  Q <= std_logic_vector(CNT);
end architecture;
```


可配置计数器.

Instanciation d'un composant générique

端口说明

```
-- l'entité du composant COUNTER
entity COUNTER is
  generic(N : integer:=8);
  port (CLK, RST : in  Std_logic;
        Q        : out   Std_logic_vector(N-1 downto 0));
end entity;
```

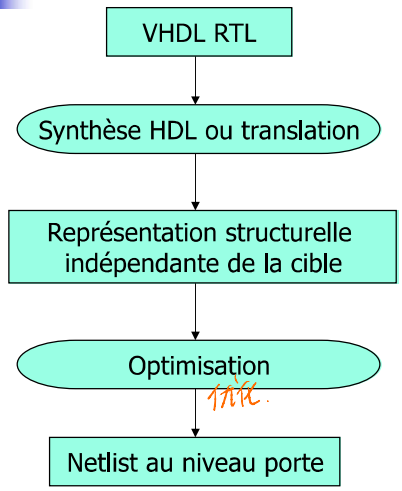
可通过改变N来调整计数值位宽.

引用

```
-- Utilisé dans l'architecture STRUCT d'un composant BLOK
architecture STRUCT of BLOK is
  signal Count4: std_logic_vector(3 downto 0);
  signal Count6: std-logic_vector(5 downto 0);
begin
-- association par position
  U1: entity work.COUNTER generic map(4)
  port map(CLK , RST, Count4);
-- association par nom
  U2: entity work.COUNTER generic map (N => 6)
  port map (CLK => CLK, RST => RST, Q => Count6);
end architecture;
```

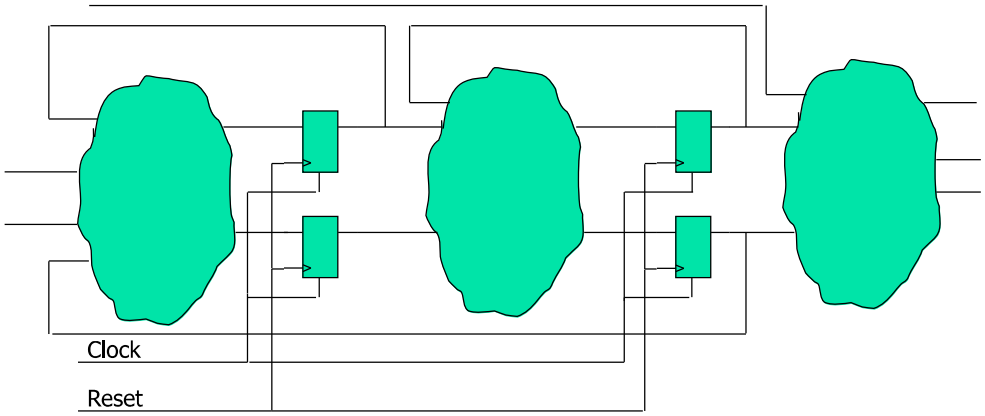
推荐用这个引用方式.

Fonctionnement des outils de synthèse



- Pas d'optimisation
- Structure inféré par le code VHDL
- Optimisation de la logique combinatoire et des contraintes de temps entre registre
- Dépend de la cible

Synthèse RTL



- La synthèse RTL n'ajoute, ne supprime ou ne bouge pas de registre.
- La synthèse RTL optimise uniquement la logique combinatoire.

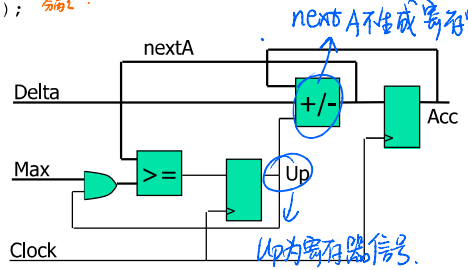
VHDL生成寄存器规则

Règles : Les registres à la synthèse

- Les outils de synthèse RTL inférent les registres directement à partir du code VHDL suivant certaine règle élémentaire :
  1. Des **registres** sont **inférés** à la **synthèse** que dans les **process synchrone.**
  2. **Signaux:** Tout les signaux assignés dans un **process synchrone** sont synthétisés par des registres.
  3. **Variables** : les variables assignés dans un **process synchrone** peuvent être synthétisées soit par un fil, soit par un registre.
     - Les variables sur lesquelles sont assignées une nouvelle valeur avant d'être lues sont synthétisées par un fil. (**assigné avant d'être lu => fil)**
     - Les variables qui sont lues avant d'être assignées sont synthétisées par un registre. (**lu avant d'être assigné => registre**)

Les registres à la synthèse

```
signal Acc, Delta, Max: signed(11 downto 0);
process (Clock)
   variable Up : std_logic;
   variable nextA : signed(11 downto 0);
begin
   if RISING_EDGE(Clock) then
      if Up = '1' then
         nextA := Acc + Delta;
         if nextA >= Max then
            Up := '0';
         end if;
      else
         nextA := Acc - Delta;
         if nextA < 0 then
            Up := '1';
         end if;
      end if;
      Acc <= nextA;
   end if;
end process;
```



Exercice 1

```
signal reg: std_logic_vector(3 downto 0);
begin
process(clk, rst)
begin
  if rst='1' then
    reg <= (others =>'0');
  elsif Rising_edge(clk) then
    reg(3) <= D;
    reg(2 downto 0) <= reg(3 downto 1);
    Q <= reg(0);
  end if;
end process;
```

Combien de flip-flops ?

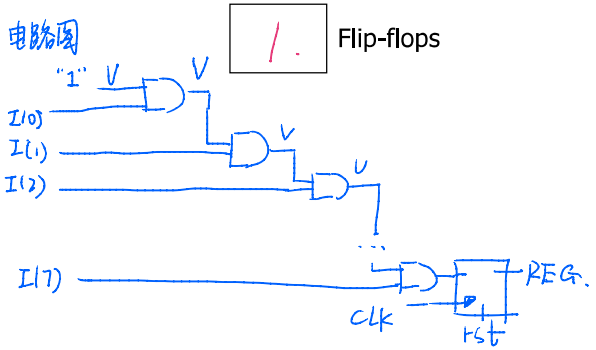
5 Flip-flops

Exercice 2

```
Signal INPUT : std_logic_vector(7 downto 0)
Signal REG : std_logic;
process (clk,rst)
variable V : STD_LOGIC;
begin
  if rst='1' then
    REG <= '0';
  elsif RISING_EDGE(clk) then
    V := '1';
    for I in 0 to 7 loop
      V := V and INPUT(I);
    end loop;
    REG <= V;
  end if;
end process;
```

Combien de bascule D flip-flops ?

1 Flip-flops



Exercice 3

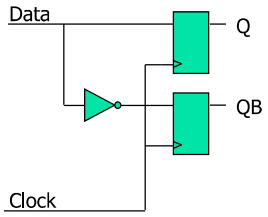
```
Signal OUTPUT : std_logic;
COUNTER : process (CLK, RST)
  variable COUNT : UNSIGNED(7 downto 0);
Begin
  if RST = '1' then
    COUNT := "00000000";
  elsif RISING_EDGE(CLK) then
    COUNT := COUNT + 1;
    OUTPUT <= COUNT(7);
  end if;
end process;
```

Combien de flip-flops ?

9 Flip-flops

La synthèse n'optimise pas les registres !

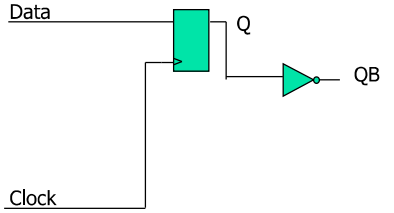
```
process (clk, rst)
begin
  if rst = '1' then
    Q <= '0';
    QB<='0';
  elsif RISING_EDGE(clk) then
    Q <= Data;
    QB <= not Data;
  end if;
end process;
```



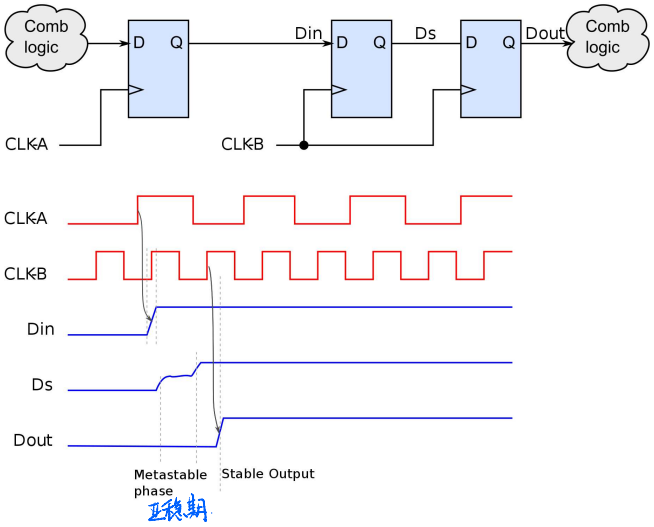
Comment faut il réécrire le code pour s'assurer qu'il n'y est qu'une seule bascule D ?

Solution !

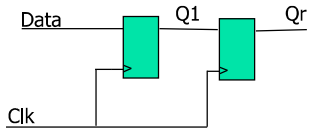
```
process (clk, rst)
begin
  if rst = '1' then
    Q <= '0';
  elsif RISING_EDGE(clk) then
    Q <= Data;
  end if;
end process;

QB <= not Q;
```



Problèmes de métastabilité



Solution : Resynchronisation



- Avantage : sûreté de fonctionnement
- Inconvénient : introduit du délai (pipeline)

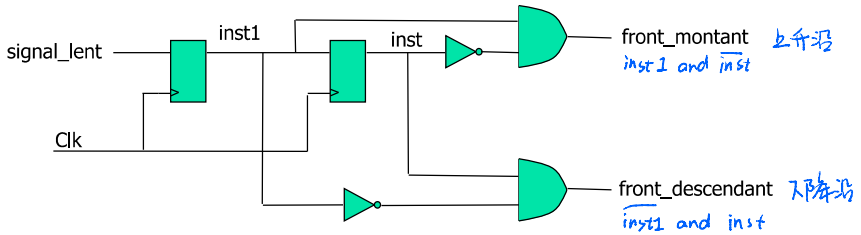
Code : re-synchronisation des entrées

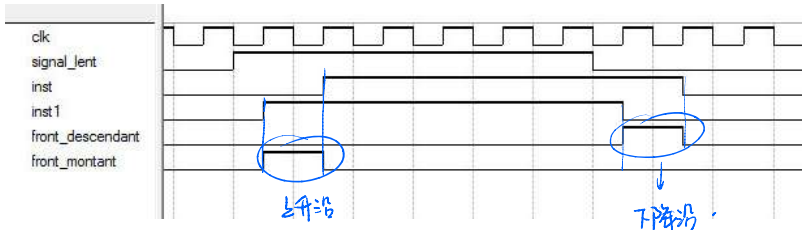
```
entity resynchro is
port(
  clk, rst, Data: in std_logic;
  Qr : out std_logic);
end entity;
architecture RTL of resynchro is
  Signal Q1 : std_logic;
begin
  process (clk,rst)
  begin
     if (rst='1') then
       Q1 <= '0'; Qr <= '0';
     elsif rising_edge(clk) then
       Q1 <= Data;
       Qr <= Q1;
     end if;
  end process;
end architecture;
```


Détection de fronts

- Les opérateurs de détection de fronts (Rising_edge(clk) et Clk'event and clk='1') ne doivent être utilisé que pour tester le front d'une horloge (clk).
- Chaque fois que l'on fait un test de front, l'outil de synthèse comprend qu'il s'agit d'une horloge.
- Horloge = le signal le plus rapide du circuit.

Exercice : Détection des fronts d'un signal lent





Détection de front lent : code

```
Entity detect_fronts is
port(
  clk, rst, signal_lent : in std_logic;
  front_descendant : out std_logic;
  front_montant : out std_logic);
end entity;
architecture RTL of detect_fronts is
  signal inst,inst1 : std_logic;
begin
  PROCESS ( clk , rst)
  BEGIN
    if rst ='1' then
      inst <= '0'; inst1 <= '0';
    elsif rising_edge (clk) then
      inst1 <= signal_lent;
      inst <= inst1;
  end if;
  end process;
  front_montant <= inst1 and (not inst);
  front_descendant <= inst and (not inst1);
end architecture;
```


Exemple d'utilisation : compteur d'évenements

```
entity cnt_evt is
port( clk, rst, sig_lent : std_logic;
      cnt : std_logic_vector(7 downto 0));
end entity;
architecture RTL of cnt_evt is
  signal Q : unsigned (7 downto 0);
  signal fm : std_logic;
Begin
U1 : entity work.detect_front port map (
     clk => clk, rst => rst, signal_lent=>sig_lent,
     front_montant => fm);
Process (clk,rst)
begin
  if (rst='1') then
    Q <= (others => '0');
  elsif rising_edge(clk) then
    if (fm='1') then
      Q<= Q +'1';
    end if;
  end if;
end process;
cnt <= std_logic_vector(Q);
End architecture;
```