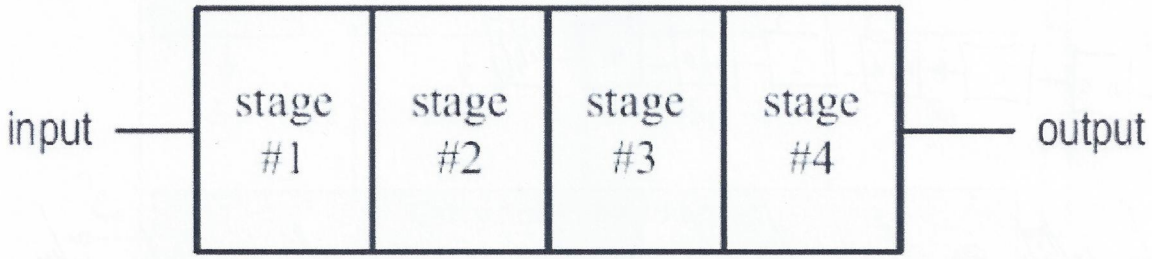
Travail écrit 2: Système Numérique 2 2016/2017

Filière : Télécommunication
Classe : T-2a, T-2d
Date : 1 juin 2017, 13h00 à 14h35
Professeur : Fabio Cunha

Nom et prénom : **Points :** 9,5 **/15** **Note :** 4.2

Problème 1 : Pipeline (4 pts) 2.5

Soit le design suivant composé de plusieurs étages combinatoires cascadés:



Avec les délais respectifs pour chacun des étages: T1 = 20 ns, T2 = 23 ns, T3 = 19 ns et T4 = 25 ns. Il s'agit d'améliorer les performances du design en pipelinant le circuit, sachant que le temps pour le setup et le temps de basculement d'une bascule sont respectivement Tsetup = 2ns et Tclock2q = 4ns.

a) Est-ce que le circuit suivant se prête bien à un design pipeline ? Justifiez votre réponse avec deux arguments au minimum.

1/1 pt

20 + 23 + 19 + 25 = 82 ns

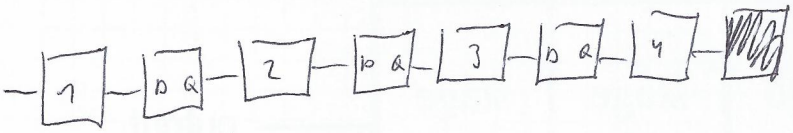
b) Calculez la fréquence d'horloge maximale pour assurer le bon fonctionnement de ce circuit **sans pipeline**.

1/ **1 pt**

$\frac{1}{87\,ns}$ (j'ai pas de calculatrice) ✓

c) Dessinez à présent le circuit en effectuant un pipeline efficace et indiquer clairement quelles sont la (les) cellule (s) qui limitent la fréquence d'horloge du circuit pipeliné.

0.5/ **1 pts**



La cellule qui limite le plus la fréquence est celle avec le plus long temps de délais donc stage #4 avec 25 ns

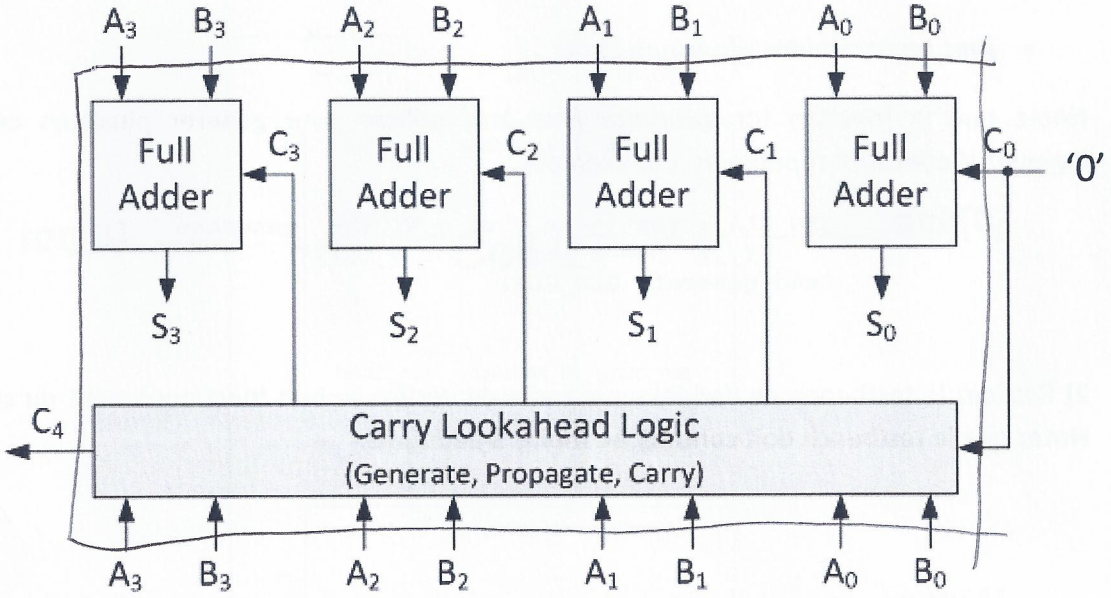
d) Calculez la fréquence maximale à laquelle ce circuit peut fonctionner **avec pipeline**.

0/ **1 pt**

$$\frac{1}{25\,ns + \underbrace{2ns \cdot 3}_{2ns \cdot 3} + 4\,ns}$$

Problème 2: Additionneur « Carry-Lookahead » (10 pts) 6.5

Afin de réduire le délai dû à la propagation du carry dans un additionneur « ripple-carry adder, il est possible d'évaluer rapidement pour chaque étage si le carry de l'étage précédemment vaut '0' ou '1'. Si une évaluation correcte peut être faite dans un temps relativement court, la performance de l'additionneur complet sera améliorée. Il s'agit en fait d'un additionneur de structure « Carry-Lookahead » :



Pour mieux comprendre la logique du block Carry Lookahead logic, la fonction du carry-out pour un étage i est :

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

Si on factorise cette expression :

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i$$

On peut réécrire l'expression sous cette forme :

$$c_{i+1} = g_i + p_i c_i$$

Où le terme generate (g) :

$$g_i = a_i b_i$$

Et le terme propagate (p) :

$$p_i = a_i + b_i$$

La fonction g_i est égale à '1' quand les deux entrées sont égales à '1', sans se préoccuper du carry d'entrée, d'où le nom generate pour la fonction puisqu'il génère le carry-out. Concernant le terme p_i est égale à '1' lorsqu'une des deux entrées vaut '1'. Toutefois, le carry est généré si le carry-in vaut '1', d'où le terme propagate puisque l'on propage le carry-in.

1) Vous devez réaliser le code VHDL du carry-lookahead adder en le rendant le plus générique possible :

3,5/7 pts

- en utilisant uniquement des mappings (4 pts)
- avec la fonction for...generate (2 pts)
- avec un paramètre générique (1 pt)

Notez que la fonction for...generate peut-être utiliser pour générer plusieurs cellules logiques si elles sont répétitives, par exemple :

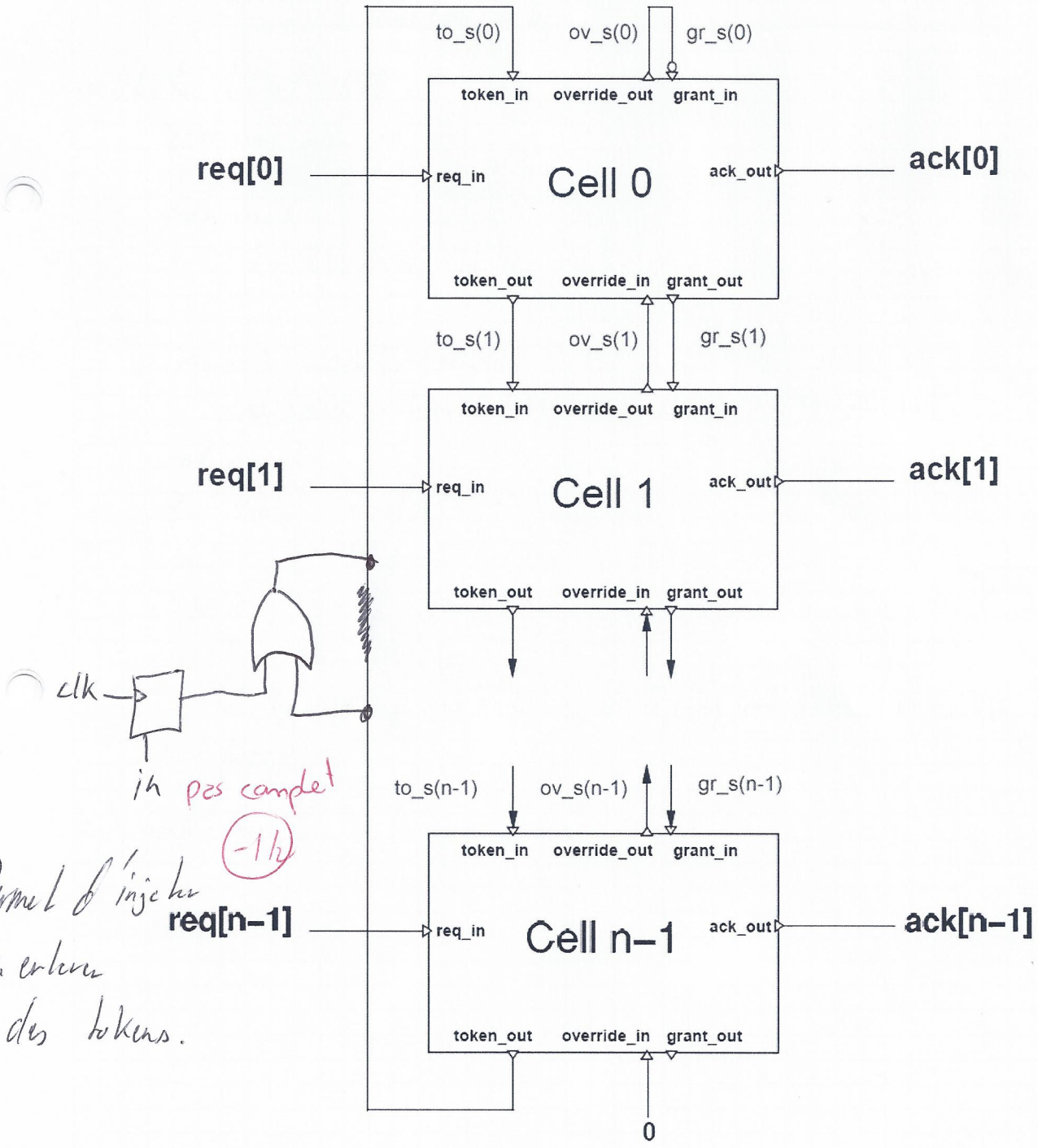
```
GEN_CLA : for jj in 0 to g_WIDTH-1 generate
    G(jj)   <= a_in(jj) xor b_in(jj);
end generate GEN_CLA;
```

2) Réalisez le testbench de l'additionneur afin de vérifier le bon fonctionnement du circuit. Notez que le testbench doit contenir au moins 3 additions.

3/3 pts

Problème 3: Arbiter (1 pt) 0,5

Soit la structure de l'arbiter que vous avez réalisé en TP. Ce schéma est incomplet et n'assure pas le bon fonctionnement de l'arbiter, notamment la règle 3. Indiquez directement sur le schéma le changement à apporter afin de s'assurer que l'arbiter fonctionnement correctement.



ih pas complet -1/2

Permet d'injecter ou enlever des tokens.

1/ a) Oui, car on peut le diviser en plusieurs étage fonctionnant en cascade. et car les délais de étages sont + ou - identiques. ✓

2) library ieee;
use ieee.std_logic_1164.all;
entity stdd is generic (n : integer);
port (Cin : in std_logic;
a, b : in std_logic_vector (n-1 downto 0);
s : out " " " " " ;
Cout : out std_logic);

architecture bc of stdd is
component adder is
port (a, b, c : in std_logic;
s : out std_logic);
end component;
signal Za, Zb, Zs, Zc : std_logic_vector (0 to n); ← pas nécessaire pour tous les vecteurs (-1/2)

begin
Q1 : for I in 0 to (n-1) generate
add : adder port map (Zc(I), Za(I), Zb(I), Zs(I), open);
end generate;
Za <= a; Zc(0) <= '0'; size mismatch (-1)
Zb <= b; Cout <= Zc(n-1); ✗ (-1/2)
S <= Zs; ou alors dans un process
for I in 0 to (n-1) ~~loop~~ generate (-1)
Zc(n+1) <= (Za(n) ~~or~~ AND Zb(n)) or (Zc(n) and (Za(n) or Zb(n)));
end loop; (-1/2)

end architecture;

Test bench

```
library ieee; use ieee.std_logic_1164.all;
entity tb is
end tb;
architecture tb of tb is
    signal Cin, Cout : std_logic;
    signal a, b, S : std_logic_vector (n-1 downto 0);   signal n : integer := 2;

    component sh11 is
    ...
    end component

begin
    map : sh11
        port map ( Cin => Cin,
                   Cout => Cout,
                   a => a,
                   b => b,
                   S => S );

    process
    begin
        Cin <= '0';
        a <= "11";
        b <= "11";
        wait for 100 ns;
        assert (Cout = '1') report ("erreur carry") severity ERROR;
        assert (S = "11") report ("faux resultat") severity ERROR;
        a <= "01";
        b <= "10";
        wait for 100 ns;

        assert (Cout = '0') report ("erreur de carry") severity ERROR;
        assert (S = "11") report ("faux resultat") severity ERROR;
        a <= "00";
        b <= "00";
        wait for 100 ns;

        assert (Cout = '0') report ("faux carry") severity ERROR;
        assert (S = "00") report ("faux resultat") severity ERROR;

    end process;
end architecture;
```