INP ENSEEIHT

Partiel d'Architecture des Ordinateurs

Documents autorisés : planches du Cours, notes Cours/TD/TP/Sujets projets

Bien lire le sujet en entier avant de se lancer.

Durée : 1 Heure 30

1 Questions de cours

1. Quels sont les avantages et les inconvénients respectifs des caches à correspondance direct et des caches associatifs par ensemble ?
2. Le mécanisme de bypass permet-il de limiter l'impact d'un alea de contrôle d'un pipeline ? Pourquoi ?
3. Dans un many-coeurs mettant en oeuvre le wormhole switching, la taille des buffers en entrée des routeurs peut-elle avoir un impact sur le délai de bout en bout des paquets ? Pourquoi ?

2 Un générateur d'horloge spéciale

Avertissement :

Cet exercice a pour objectif de vérifier que vous êtes capable d'écrire en VHDL un composant (**entité, architecture**) et de décrire son fonctionnement en utilisant les possibilités du langage. Il n'y a pas beaucoup de code à écrire (le corps de l'architecture peut faire moins d'une vingtaine de lignes)

Exercice : Écrire **l'entité et l'architecture** du composant générique `generateur` dont le fonctionnement est présenté dans la suite.

Nous voulons définir un composant générique `generateur` capable à partir d'une horloge de base `clk`, de générer un signal qui restera à '1' pendant un certain nombre de périodes de cette horloge `clk` et à '0' pendant un autre nombre de périodes et ainsi de suite.

Il y aura aussi un `reset` permettant de ré-initialiser le composant.

Ce type de matériel est intéressant dans le cas de traitements périodiques complexes.

Les chronogrammes de la Figure 1 montrent deux signaux `ph1` et `ph2` générés à partir de **deux composants `generateur` différents.**

Vous devez écrire une architecture **SANS UTILISER D'OPÉRATION ARITHMÉTIQUE.**

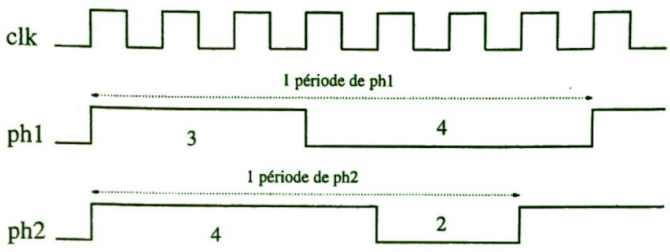

FIGURE 1 – Deux signaux générés par 2 composants `generateur`

Pour compter les périodes, vous avez à votre disposition un composant générique `compteur` qui pourra être instancié comme sous-composant de `generateur`.

Le composant générique a l'interface suivante :

```
entity compteur is
  generic(size : natural := 8);
  port (clk, reset : in std_logic;
        val : out natural);
end compteur;
```

et permet de compter de 0 à `size - 1` indéfiniment (quand le reset n'est pas actif).

Ce n'est pas la peine d'écrire l'architecture de ce compteur.

3 Une décomposition plus fonctionnelle du composant `er_1octet`

On désire reprendre l'architecture du composant `er_1octet` développé lors du mini-projet. Pour rappel, l'architecture découlant des réflexions du TD ressemblait à celle donnée en annexe : un seul process nous permettait de générer `sclk`, envoyer un octect bit à bit sur `mosi` et réceptionner un octet, bit à bit, depuis `miso`.

Une **différence notable** par rapport à la version du TD est l'utilisation de deux registres : un registre `reg_mosi` pour envoyer l'octet sur `mosi` et un registre `reg_miso` pour collecter l'octet reçu sur `miso` [1].

On souhaite reprendre cette architecture en la décomposant en unités fonctionnelles **(on se contentera de traduire ces unités par des process VHDL sans en faire des sous-composants)**, chaque unité étant chargée de réaliser une tâche.

Voici quelques questions qui peuvent vous guider dans votre développement :

1. Quel est le nombre d'unités envisageable au vue de ce que doit réaliser `er_1octet` ?
2. Les unités d'émission et de réception peuvent être cadencées par quelle horloge ? Pour chacune d'entre elles, sur quel front ?

À réaliser :

1. répondez aux 2 questions
2. développer une architecture où chaque tâche sera réalisée par un process

Vous ferez attention à

— ce qu'un signal ne soit mis à jour que dans un seul process

— ne pas utiliser (== consulter la valeur) des signaux en sortie du composant mais passer par un signal auxiliaire si nécessaire

1. Cela va vous permettre de décomposer plus facilement le fonctionnement du composant (objet de l'exercice).

Annexe : Entité et Architecture du composant er_1octet

```
entity er_1octet is
  port (
    rst  : in std_logic;
    clk  : in std_logic;
    en   : in std_logic;
    din  : in std_logic_vector(7 downto 0);
    miso : in std_logic;
    --
    sclk : out std_logic;
    mosi : out std_logic;
    dout : out std_logic_vector(7 downto 0);
    busy : out std_logic);
end er_1octet;

architecture behavioral_4 of er_1octet is
  type t_etat is ( idle, bit_emis, bit_receptionne );
  signal etat : t_etat;
begin
  traitement : process(clk, rst)
    -- variables locales
    -- compteur des bits (reçus et envoyés)
    variable cpt: natural;
    -- registre contenant la donnée à envoyer
    variable reg_miso : std_logic_vector (7 downto 0);
    -- registre où est rangée la donnée reçue
    variable reg_mosi : std_logic_vector (7 downto 0);
  begin

    if ( rst = '0' ) then
      etat <= idle;
      busy <= '0';
      reg_mosi := (others => '0');
      reg_miso := (others => '0');
      cpt := 7;
      mosi <= '0';
      -- sclk au repos à '1'
      sclk <= '1';
      dout <= (others => '0');

    elsif ( rising_edge(clk) ) then
```

```
      case etat is
        when idle => -- état d'attente

          -- un échange est demandé
          if ( en = '1' ) then
            -- on charge le registre avec la donnée à envoyer
            reg_mosi := din;
            busy <= '1';
            -- on envoie le bit de poids fort sur front descendant de sclk
            cpt := 7;
            sclk <= '0';
            mosi <= reg_mosi(cpt);
            -- on passe à l'état suivant
            etat <= bit_emis;
          end if;

        when bit_emis => -- état de réception
                         -- (nom de l'état correspond à ce que l'on a fait
                         -- en y arrivant)
          -- on réceptionne le bit cpt sur un front montant de sclk
          sclk <= '1';
          reg_miso(cpt) := miso;

          -- si on réceptionne le bit de poids faible, on a fini l'échange
          if ( cpt = 0 ) then
            dout <= reg_miso;
            busy <= '0';
            etat <= idle;
          else
            etat <= bit_receptionne;
          end if;

        when bit_receptionne => -- état d'émission
          -- on passe au bit suivant
          cpt := cpt - 1;
          -- on envoie le bit sur front descendant de sclk
          sclk <= '0';
          mosi <= reg_mosi(cpt);
          -- après un bit émis, il y a toujours un bit à réceptionner
          etat <= bit_emis;
      end case;
    end if;
  end process traitement;
end behavioral_4;
```