LE GUIDE DU ROOTARD VHDL

Patrice Nouel
MAJ pk 28/03/17

1. AVANT PROPOS

Le langage VHDL pose des problèmes car il est double. Il y a l'excellent descripteur de comportements logico-temporels dont la finalité est la construction de modèles validés par la simulation et il y a le descripteur de circuits qui est en fait un spécificateur pour le synthétiseur. Si le code synthétisable est toujours simulable, l'inverse n'est pas vrai et là réside la principale difficulté.

Cet ensemble de questions-réponses s'adresse à ceux qui veulent construire des circuits de qualité. Après plus de 10 ans de pratique auprès de débutants, c'est toujours les mêmes questions qui posent problème. Cela oblige à rappeler avec force quelques principes simples.

S'imposer quelques règles rigides lorsqu'on débute donnera de bonnes habitudes et l'on pourra toujours avec l'expérience assouplir ces principes.

Il n'est pas question pour moi de réécrire mon cours de VHDL. Chaque fois que cela sera nécessaire, j'y ferai cependant référence.

2. CONCEVOIR UN CIRCUIT OU CE QUE VHDL NE SAIT PAS FAIRE

Comme dans tout projet, la qualité provient avant tout de l'analyse qu'on a pu faire du problème posé. Plus celle-ci est approfondie, plus la solution technique devient évidente. Cela s'appelle structurer le projet c'est à dire le décomposer jusqu'à voir apparaître des fonctions bien connues. Dans le cas du VHDL cette granularité est bien représentée par un compteur ou une mémoire.

Ce que ne peut pas faire VHDL à votre place, c'est structurer le circuit. Malheureusement les synthétiseurs essaient d'être de plus en plus "intelligents" et bien souvent vont trouver une solution à partir des fichiers sources qui leur est proposé.
Pour un circuit pas ou peu structuré, une solution pourra être trouvée mais qu'en est-il de sa qualité (surface, vitesse, documentation, réutilisation) résultante ?

==> IL FAUT STRUCTURER

3. CONCEVOIR SYNCHRONE

3.1. Écriture standard

L'horloge choisie doit être le signal le plus rapide du système, c'est à dire celui qui donnera le temps minimal d'échantillonnage des autres signaux (qualifiés de lent).

==> **UNE SEULE HORLOGE**

Chaque fois que l'on écrit `h'event AND`..., le synthétiseur comprend que `h` est une horloge.

Si le circuit est synchrone, tous les processus avec mémorisation par registres devront s'écrire :

```
synchrone: PROCESS
      BEGIN
            WAIT UNTIL rising_edge (clk); -- c'est donc synchrone de clk
            IF clear = '1' THEN
                 q <=  OTHERS => '0';     -- remise à zéro synchrone
            ELSIF  condition_logique THEN -- Entrée Enable des bascules D
                q <= entree_particuliere;
            END IF;
      END PROCESS;
```

Remarques :

- Préférer le `WAIT UNTIL` plus explicite à la liste de sensibilité (`clk`) suivie d'un `IF rising_edge(clk)`.
- Autant de `rising_edge()` sur des signaux différents autant d'horloges. Dans les cas simple, une seule horloge suffit.
- Si l'on doit traiter des signaux lents par rapport à l'horloge, ils interviennent au niveau de l'*ENABLE*, c'est à dire de la condition logique du `PROCESS`. Par exemple, l'horloge est à 50 MHz, on veut compter un signal qui provient touts les millisecondes. La condition logique sera vraie pendant une seule période de base toutes les millisecondes (voir aussi exemple suivant).

3.2. Traiter le front d'un signal lent

On a souvent besoin de traiter l'événement que constitue un front (montant ou descendant) d'un signal quelconque (donc lent par rapport à l'horloge). Pour rester conforme à la contrainte du synchronisme, il faut donc créer pour le signal lent une détection de front. On échantillonne le signal, on mémorise sa valeur, une succession 0..1 détecte un front montant, une succession 1..0 détecte un front descendant.

```
Detection: PROCESS
      VARIABLE detect : std_logic_vector(1 DOWNTO 0);
        BEGIN
            WAIT UNTIL rising_edge (clk); -- c'est donc synchrone de clk
             front_montant <= '0';  front_descendant <= '0' ;
             detect(1) := detect(0);
             detect(0) := signal_lent;
             IF detect = "01" THEN
                         front_montant <= '1';
             END IF;
             IF detect = "10" THEN
                        front_descendant <= '1';
            END IF;
        END PROCESS;
```

Observations :

- Les signaux `front_montant` ou `front_descendant` vont servir de condition logique (*ENABLE*) dans un PROCESS de traitement.
- On n'a toujours qu'une seule horloge qui est `clk`.

4. SEPARER AU MAXIMUM LE COMBINATOIRE ET LE SEQUENTIEL

Si dans un processus, on constate un grand nombre d'imbrications de `IF` (plus de 3) il faut se poser la question de la simplification éventuelle de cette écriture.

```
WAIT UNTIL rising_edge (clk);
  IF condition1 THEN ...
        IF condition2 THEN ...
            IF condition3 THEN ...
```

On a ici associé à une fonction de mémorisation une fonction combinatoire de `condition1`, `condition2`...

Si cette fonction est combinatoire, a-t-on décrit toutes les combinaisons possibles ? Cette fonction, si elle est trop complexe, va créer des retards importants et faire chuter la fréquence maximale du circuit.
Enfin, pour la lisibilité de la description, n'aurait-on pas eu intérêt à séparer cette partie combinatoire de la partie purement séquentielle ?

4.1. Un circuit combinatoire

On privilégie toujours l'écriture en instruction concurrente. C'est plus économique en nombre de lignes et c'est surtout plus lisible.

```
S1 <=   e1 OR (e2 AND NOT e3);  -- une équation
```

4.2. Un contre exemple

```
faux_combi : PROCESS(e1) --ERREUR
      BEGIN
      IF (e1 OR (e2 AND NOT e3)) = '1' THEN
            s1 <= '1';
          ELSE
            s1 <= '0';
          END IF;
     END PROCESS;
```

Observations :

- La liste de sensibilité est équivalente à `WAIT ON`. Si vous êtes débutant, préférer le `WAIT ON` explicite, vous vous apercevez tout de suite que le processus décrit est faussement combinatoire. En effet si `e2` ou `e3` changent, la sortie n'est pas affectée. Elle ne le sera que lorsqu'`e1` changera.
- On a ici affaire à un type de circuit inconnu en conception synchrone. Il y aurait ici un effet mémoire asynchrone.

On doit mettre en liste de sensibilité tous les signaux d'entrée si le circuit est combinatoire.

L'écriture correcte est alors la suivante :

```
vrai_combi : PROCESS
      BEGIN
      WAIT ON e1, e2, e3;  -- toutes les entrées puisque c'est combinatoire
          IF (e1 OR (e2 AND NOT e3)) = '1' THEN
            s1 <= '1';
          ELSE
            s1 <= '0';
          END IF;
      END PROCESS;
```

5. LE CHOIX DES BIBLIOTHEQUES ET DES TYPES

Cette question me semble d'une grande importance pour le débutant. Dans ce domaine, la facilité d'utilisation peut entraîner de grandes déconvenues.

Après consultation du site IEEE, je conseille les bibliothèques suivantes :

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
```

Au niveau des interfaces `ENTITY`, on n'utilise que des types `std_ulogic`, `std_logic` ou `std_logic_vector`. Ceci pour des raisons de compatibilité avec les différents aperçus du circuit.

Ces bibliothèques contrairement à d'autres ne permettent pas d'additionner, de soustraire ou de comparer des `std_logic_vector` entre eux.
Ceci est heureux car il est souhaitable de savoir à priori si les nombres traités doivent être considérés comme signés ou non signés. En interne dans l'architecture, on définira des signaux avec des types `signed` ou `unsigned` et toutes les opérations arithmétiques

pourront alors se traiter correctement y compris avec des types différents (`signed` et `integer`, `unsigned` et `natural`)

Le passage entre les signaux internes et les signaux visibles au niveau de l'entité se fera par un *cast* comme dans l'exemple suivant :

```
SIGNAL   niveau_entity : std_logic_vector(31 DOWNTO 0);
SIGNAL   niveau_architecture : signed(31 DOWNTO 0);

BEGIN
      niveau_entity <= std_logic_vector(niveau_architecture - 212);
```

Par défaut, choisir `std_ulogic` plutôt que `std_logic` comme le font tous les outils. En effet, `std_logic` est un `std_ulogic` avec fonction de résolution (donc qui permet de construire les bus). Les outils considèrent que « qui peut le plus peut le moins » et proposent `std_logic` comme type par défaut. Cela ne fait pas l'affaire du débutant qui préférera être alerté suffisamment tôt lorsqu'il fera des bus sans le savoir.

Exemple d'erreur souvent rencontrée :

```
un: PROCESS
      WAIT until rising_edge(clk);
      IF condition1 THEN
            s <=  une_certaine_equation;  -- premiere affectation de s
      END IF;
END PROCESS;

deux: PROCESS
      WAIT until rising_edge(clk);
      IF condition2 THEN
            s <=  une_autre_equation;     -- deuxieme affectation de s
      END IF;
END PROCESS;
```

Observations :

- Si le type de `s` est `std_ulogic`, une erreur va apparaître lors de la synthèse (compilation) du circuit. Si le type est `std_logic`, il n'y a pas d'erreur car on a décrit un bus (avec un seul fil).
- Mais le concepteur a-t-il vraiment voulu réaliser un bus sur `s` (connecter 2 sorties entre elles) ?
- N'a-t-il pas oublié qu'un même signal doit être affecté dans un même `PROCESS` ?

Ne voulait-il pas écrire la chose suivante (ou quelque chose d'approchant) ? :

```
correct: PROCESS
      WAIT until rising_edge(clk);
      IF condition1 THEN
            s <=  une_certaine_equation;
      ELSIF condition2 THEN
            s <=  une_autre_equation;
      END IF;
END PROCESS;
```

==> **UTILISER `std_ulogic`. RESERVER `std_logic` EXCLUSIVEMENT AUX BUS.**