TP FPGA

1 Introduction

Un FPGA (Field Programmable Gate Array, qu'on peut traduire par matrice de portes programmables sur site) est, pour aller vite, un *circuit logique programmable*, ou configurable, contenant quelques milliers à quelques dizaines de milliers de blocs logiques. Chaque bloc logique peut contenir une fonction combinatoire de quelques entrées (4 à 8) et de 1 à 4 bascules D, selon les générations.

On en trouve dans beaucoup de systèmes pour assurer des tâches secondaires, typiquement l'interconnexion entre des processeurs et circuits intégrés spécialisés (ASIC) : on parle de *glue logic*.

Ils sont aussi très utilisés pour décharger les processeurs de certaines tâches très coûteuses en calcul : le codage/décodage canal, la compression/décompresion vidéo, le filtrage numérique, ou de manière générale, les fonctions de traitement du signal (en particulier en télécoms).

Une autre application majeure des FPGA est le prototypage des ASIC : les concepteurs implantent dans un FPGA tout ou partie de leur architecture (avec un fonctionnement à fréquence réduite). Ils peuvent ainsi le tester et affiner le design.

En pratique, l'utilisateur peut *décrire* le circuit logique qu'il souhaite synthétiser à l'aide de portes et bascules de base (synthèse graphique) ; ou – et bien sûr c'est beaucoup plus puissant – à l'aide d'un *langage de description matérielle*. Dans les deux cas, c'est l'outil logiciel qui se charge de traduire le design, de le mapper aux ressources disponibles en fonction du composant (les blocs logiques), et de faire l'interconnexion.

Les langages de description les plus utilisés sont Verilog et VHDL. VHDL signifie **V**ery high speed integrated circuit **H**ardware **D**escription **L**anguage. Le langage a été développé à partir de 1983 ; il s'agissait au départ d'un projet du ministère de la défense des États Unis. Puis, il a été normalisé par l'IEEE en 1987. Le langage a été mis à jour en 1993, 2000, 2002 et 2008. Les modifications apportées en 2000 et 2002 sont mineures ; en revanche, celles de 1993 et 2008 sont importantes. Comme celle de 2008 n'est pas encore entièrement supportée par les outils, nous présentons ici le VHDL-1993.

Pour terminer, soulignons que la conception d'un circuit intégré numérique (ASIC) se fait également à l'aide d'un langage de description matérielle, qui peut être le VHDL.

Ce TP n'est qu'une introduction, et de nombreux aspects ne pourront pas être traités.

Rappel : circuit logique combinatoire, circuit logique séquentiel

Un circuit logique combinatoire est constitué de portes logiques placées les unes derrière les autres, sans qu'il n'y ait de rebouclage d'une sortie vers l'entrée. Dans un circuit combinatoire, les sorties ne dépendent donc que des entrées (pour les mêmes entrées, j'ai toujours les mêmes sorties). Un multiplexeur, un décodeur, un additionneur, sont des exemples classiques de circuits combinatoires.

Dans un circuit séquentiel, au contraire, il y a un rebouclage d'une ou plusieurs sorties vers les entrées. Les sorties dépendent donc des entrées, mais également de leurs propres valeurs aux instants précédents (pour les mêmes entrées, les sorties peuvent avoir des valeurs différentes). Le circuit séquentiel possède une mémoire, ou un état.

La plupart du temps, les circuits séquentiels possèdent une entrée d'horloge, et leur fonctionnement est *synchrone* : les sorties ne peuvent changer que sur front d'horloge. Les bascules (flip-flops), les compteurs, les registres ou bancs de registres, sont des exemples de circuits séquentiels.

TP1 : Une porte XOR

L'objectif de ce premier TP est la prise en main des outils. Nous utilisons un FPGA Altera Cyclone V (Altera est l'un des deux leaders du marché avec Xilinx). Celui-ci est placé sur une carte de développement.

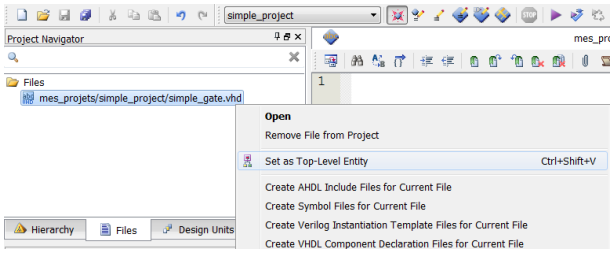
Lancer `Quartus II` (Windows) et choisir `New Project Wizard`. Définir un répertoire de travail (sur votre disque personnel), un nom pour le projet (par exemple `simple_project`) et un nom de `Top-Level entity`, par exemple `simple_gate`.

Dans la page suivante, cliquer sur Next sans ajouter de fichier. Dans Family, choisir Cyclone `V, et dans Devices, Cyclone V GX Extended Features. Notre FPGA est le 5CGXFC 5C6F27C7N.` Cliquer sur `Next` deux fois, puis sur `Finish`.

Une fois le projet créé, faire `File > New` et dans `Design Files`, choisir `VHDL File`. Enregistrer tout de suite le fichier dans le répertoire du projet (vous pouvez garder le nom proposé par défaut, et l'extension doit être `.vhd`).

Un projet peut contenir plusieurs fichiers. La liste des fichiers apparaît dans le `Project Navigator` (fenêtre de gauche), onglet `Files`. Chaque fichier contient une *entité* (entity). Le fichier qui sera compilé en premier est appelé *l'entité de plus haut niveau* (top-level entity), et il apparaît tout en haut dans l'onglet `Hierarchy`.

Si l'on veut promouvoir un fichier comme top-level, faire un clic droit sur le fichier en question (onglet `Files`), et choisir `Set as Top-Level Entity`.



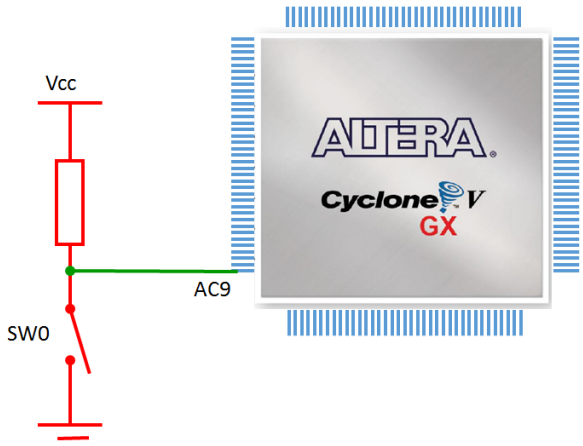
Une fois que vous êtes certain que votre fichier est l'entité de plus haut niveau, copier le code ci-dessous (l'objectif ici n'est pas de comprendre le code VHDL).

```
library ieee;
use ieee.std_logic_1164.all;

entity simple_gate is
        port (
                a, b : in bit;
                c : out bit
        );
end;

architecture rtl of simple_gate is
begin
        c <= a xor b;
end;
```

Noter que le nom de l'`entity` (ici `simple_gate`) doit forcément être le nom du fichier.

Dans le menu `Processing`, aller dans `Start > Start Analysis & Synthesis` (ou `CTRLE+K`). Il s'agit de la première étape de compilation, qui vérifie les erreurs éventuelles.

L'étape suivante permet d'indiquer sur quelles pins du FPGA se situeront ces entrées/sorties. Le FPGA est situé sur une carte de développement, et certaines de ses pins sont connectées à un interrupteur, un bouton poussoir, une Led, etc. Par exemple, on voit ci-dessous que l'interrupteur (ou switch) SW0 permet d'appliquer un 1 ou un 0 sur la patte AC9. Pour connaître les numéros des pattes utiles dans cette séance, consulter le user manual du Cyclone V GX.



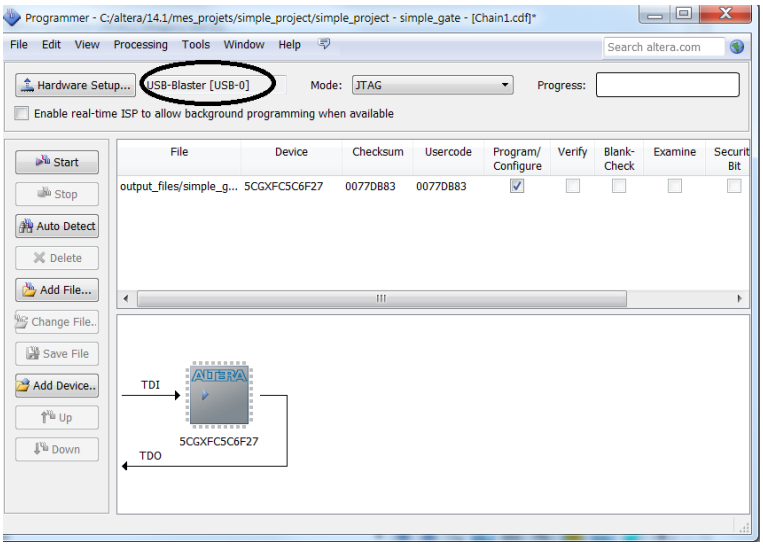
Pour faire l'association entre un port d'entrée ou sortie du schéma et un numéro de pin, aller dans `Assignment > Pin Planner`. Tout en bas, on trouve un tableau avec une ligne par entrée/sortie. Compléter la colonne Location pour les 3 entrées/sorties, en utilisant le user manual (sur le schéma ci-dessous, le travail a été fait seulement pour l'entrée `a`).

Node Name	Direction	Location	I/O Bank
a	Input	PIN_AC9	3B
b	Input		
c	Output		
<<new node>>			

Fermer la fenêtre (pas la peine de sauvegarder) et compiler en allant dans `Processing > Start Compilation` (ou `CTRLE+L`). Le processus est plus long que précédemment, car il y a plus d'étapes. En particulier, c'est dans l'étape `Fitter` qu'est effectué le mapping de votre design dans les milliers de blocs logique du FPGA.

Une fois la compilation terminée, un fichier de programmation a été généré. Pour l'envoyer au FPGA :

- alimenter la carte, puis la connecter au PC. Attention, il y a deux ports USB sur la carte, utiliser le `USB BLASTER`. Mettre la carte sous tension (bouton rouge) ;
- faire `Tools > Programmer`. Dans la fenêtre qui s'ouvre, vérifier que `USB-Blaster` apparaît en haut, cliquer sur `Add File...` et dans le répertoire `output_files` choisir le fichier portant le nom du projet, avec l'extension `.sof` ;



— sur la carte, vérifier que le switch situé à côté du slot pour carte SD est bien sur `RUN`.
— cliquer sur `Start`. Normalement, en quelques secondes, la programmation se termine (en cas d'échec, cliquer à nouveau sur `Start`).

Repérer les switchs et la Led que vous avez utilisés, et vérifier que vous avez bien implanté une porte XOR.

Couper l'alimentation (bouton rouge) et la remettre. Que se passe-t-il ?

Le FPGA est un circuit *volatile* qui perd son contenu lorsque l'alimentation est coupée. Pendant la phase de développement, ce n'est pas un problème, mais ça l'est quand on passe en production. C'est pourquoi le FPGA est toujours accompagné d'une mémoire non volatile (EEPROM) où il va charger sa configuration au Reset.

L'EEPROM présente dans notre carte est une EPCQ256.

2 Introduction au VHDL

Le VHDL est un langage conçu à la fois pour la simulation et pour la synthèse de circuits logiques. Par synthèse, on entend que l'outil utilisé va traduire le code en un circuit numérique – du hardware.

Un code VHDL peut être exécuté en simulation. On fournit des signaux d'entrées, et l'exécu-

tion du code, selon les règles que nous allons étudier, fournit des signaux en sortie. Le langage permet de faire en sorte que les signaux de sortie soient les mêmes que s'ils étaient fournis par des circuits numériques ; c'est ce qui permet à l'outil d'effectuer la synthèse.

Nous allons écrire du VHDL pour la synthèse seulement. On s'intéressera surtout au langage VHDL proprement dit ; le circuit numérique résultant de la synthèse d'un morceau de code, ou, réciproquement, le choix du code VHDL adapté à la synthèse d'un certain circuit numérique, ne seront pas ou peu abordés.

2.1 Design unit, entité, architecture

Un projet est composé de modules organisés de façon hiérarchique. Chaque module est construit à partir d'un couple de *design units* (la traduction française n'étant utilisée par presque personne, je m'en tiens en général aux termes anglais). Il peut s'agir :

- d'une déclaration d'entité (entity declaration) ;
- du corps de l'architecture (architecture body) ;
- d'une déclaration de package (package declaration) ;
- d'un corps de package (package body).

Le corps d'architecture se rapporte forcément à une entité, et de même le corps de package à un package.

Une *déclaration d'entité* décrit un module (un circuit numérique) d'un point de vue extérieur, en donnant son nom, ses entrées/sorties, ainsi que des paramètres éventuels.

Les entrées sorties s'appellent des *ports*, les paramètres des *generics*. Une déclaration d'entité a toujours la structure suivante :

```
entity entity_name is
        generic(...);
        port (...);
end;
```

La liste des ports spécifie leur nom, leur direction et leur type. Par exemple, on déclare ci-dessous deux entrées nommées `reset` et `clock`, une sortie nommée `count` sur 8 bits :

```
port (
        clock : in std_logic;
        reset : in std_logic;
        count : out std_logic_vector(7 downto 0)
);
```

Les directions les plus utilisées sont `in` et `out`. Les sorties déclarées comme `out` possèdent une particularité : on ne peut lire leur valeur depuis l'architecture. Ce défaut a été corrigé par la mise à jour de 2008, mais malheureusement cette correction n'est pas supportée par les outils actuels. Ceci a un impact sur la description de certains circuits (par exemple, le compteur, que nous verrons plus loin).

Les paramètres sont optionnels ; on en utilise souvent si l'entité est instanciée par une autre entité de niveau supérieur, et ils sont alors précisés lors de cette instanciation. On peut ainsi faire varier des caractéristiques du circuit que l'on synthétise. De manière générale, l'utilisation de paramètres rend le code réutilisable.

Par exemple, on pourrait déclarer une mémoire appelée `fifo`, avec deux paramètres permettant de faire varier la taille des mots stockées, et la capacité de la mémoire en nombre de mots :

```
entity fifo is
        generic(
        -- nombre de mots
        DEPTH : natural := 8;
        -- taille des mots
        WIDTH : natural := 32
        );
        port (
        input_data : in std_logic_vector(WIDTH-1 downto 0);
        ...
        );
end;
```

La liste des paramètres spécifie leur nom, leur type, et éventuellement, leur affecte une valeur par défaut par le symbole `:=`

Le *corps d'architecture*, ou *architecture*, décrit ce qui se passe *à l'intérieur* de l'entité. On distingue souvent deux façons d'effectuer cette description :

- comportementale (behavourial) : on décrit l'évolution des sorties en fonction des entrées ;
- structurelle (structural) : on décrit explicitement les entités à utiliser en les instanciant (celles-ci ayant en général donné lieu à une description comportementale). C'est typiquement le cas de la top-level entity d'un projet découpé en modules.

Une architecture a la structure suivante :

```
architecture architecture_name of entity_name is
-- ici des déclarations (typiquement de signaux)
begin
-- ici des assignations simples ou instructions concurrentes
end;
```

Un *package*, constitué d'une déclaration de package et d'un corps de package, est un mécanisme permettant de réutiliser des parties de code dans le projet. On met dans un package des entités, des types, ou des sous-programmes.

En pratique, un package appelé `standard` est inclus par défaut dans tous les projets. Il contient des types comme `boolean`, `bit`, `integer`. Un autre package très utile, fourni par tous les outils, est le package `std_logic_1164`. Nous ne créerons pas de package dans ce cours, car on peut réutiliser (instancier) des entités depuis une autre entité directement (sans passer par un package).

2.2 Bibliothèques et types les plus utilisés

Une *bibliothèque*, ou *design library*, regroupe des entités ou packages (donc des design units) compilés. Trois bibliothèques à connaître :

- `std`, qui contient notamment le package `standard` ;
- `ieee`, qui contient par exemple le package `std_logic_1164` ;
- `work`, qui contient les design units et packages du projet.

Les types couramment utilisés en VHDL sont définis dans des packages présents dans les deux premières bibliothèques ci-dessus.

Pour utiliser un package autre que le package `standard`, par exemple le `std_logic_1164`, situé dans la bibliothèque `ieee`, on ajoute au début du fichier :

```
library ieee;
use ieee.std_logic_1164.all;
```

La première ligne déclare le nom de la bibliothèque ; la seconde rend le package et tout son contenu visibles.

Les principaux types de *signaux* (les entrées/sorties, par exemple, sont des signaux) sont donnés dans le tableau page suivante.

Type	Valeurs	Défini dans...
`bit`	'0', '1'	package `standard` (library `std`)
`std_logic`	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	package `std_logic_1164` (library `ieee`)
`std_logic_vector`	`array` of `std_logic`	package `std_logic_1164` (library `ieee`)
`integer`	$-2^{31}+1$ à $+2^{31}-1$	package `standard` (library `std`)
`signed`, `unsigned`	`array` of `std_logic`	package `numeric_std` (library `ieee`)

Le type `std_logic` est en général utilisé à la place du type `bit` : il permet en particulier de prendre en compte la logique trois états, en offrant la possibilité pour un signal d'être en haute impédance ('Z'). Nous l'utiliserons car c'est l'usage, mais nous nous contenterons des valeurs '0'et '1'.

Pour représenter un groupe (un bus) de bits, on peut utiliser le type `std_logic_vector`, le type `integer`, les types `signed` ou `unsigned`.

Le type `integer` permet de représenter un entier, et par défaut, utilise 32 bits pour cela. Il est conseillé de restreindre cet intervalle lors de la déclaration, par exemple :

```
variable count_value: integer range 0 to 9;
```

Dans ce cas, la variable `count_value` n'utilise que 4 bits. On peut faire des opérations arithmétiques sur des opérandes de type `integer` sans qu'il soit nécessaire de déclarer de package supplémentaire.

Les types `std_logic_vector`, `signed` et `unsigned` sont construits à partir du type `array` – un type composite permettant d'associer plusieurs éléments du même type (ici des `std_logic`).
A la différence du type `integer`, ces trois types permettent d'accéder individuellement à chacun des bits constituant le bus.

Il y a toutefois une différence importante entre eux : on ne peut faire d'opérations arithmétiques sur des signaux de type `std_logic_vector`. S'il est nécessaire de faire de telles opérations, on utilisera plutôt le type `signed` ou `unsigned` (la bonne pratique étant d'utiliser `signed`, pour pouvoir coder des nombres négatifs), en déclarant le package `numeric_std`.

On doit obligatoirement préciser la taille d'un signal de type `std_logic_vector`, `signed` ou `unsigned` lors de sa déclaration. Par exemple, pour déclarer une sortie `sum` sur 8 bits, on écrira :

```
sum : out signed(7 downto 0);
```

Le bit de poids fort (à gauche) a alors l'indice 7 ; il aurait l'indice 0 si l'on avait utilisé `signed(0 to 7)`. En pratique, on utilise toujours `signed(7 downto 0)`.

Si l'on veut accéder au bit d'indice 5, il suffira d'écrire `sum(5)` ; si l'on veut accéder aux 4 bits de poids faible, on écrira `sum(3 downto 0)`.

Le tableau suivant résume les caractéristiques des types utilisables pour représenter un entier ou un groupe de bits.

Type	Caractéristiques
`std_logic_vector`	Bits accessibles individuellement ; préciser la taille obligatoirement (`N-1 downto 0` pour N bits) ; pas d'opérations arithmétiques
`signed, unsigned`	Déclarer le package `numeric_std` ; bits accessibles individuellement ; préciser la taille obligatoirement (`N-1 downto 0` pour N bits) ; opérations arithmétiques possibles
`integer`	Le plus simple pour des entiers dont on n'a pas à accéder aux bits ; restreindre la taille avec `range` ; opérations arithmétiques possibles

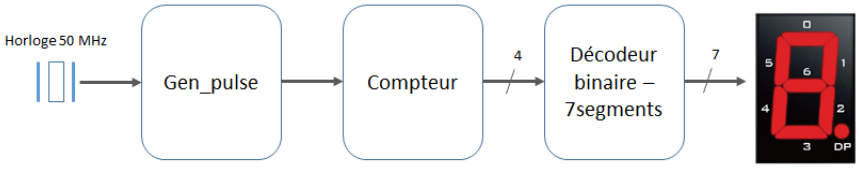
TP2 : un décodeur binaire-7 segments

Dans la suite de ce TP, on prend comme exemple la synthèse d'un circuit comptant de 0 à 9. La sortie doit être affichée sur un afficheur 7 segments, et être incrémentée à chaque seconde.

Quand on conçoit un circuit numérique, la première chose à faire est de décomposer le circuit en modules aussi petits que possible. Chaque module ne doit faire qu'une tâche à la fois.

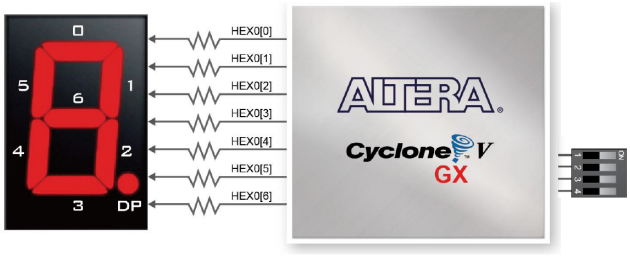
Ici, on utilisera une horloge de fréquence 50 MHz. Les trois modules proposés sont :

- un module `gen_pulse` qui, à partir de l'horloge à 50 MHz, génère chaque seconde des impulsions de durée 20 ns ;
- un module `compteur` qui compte (en binaire) de 0 à 9, la sortie étant incrémentée à chaque impulsion ;
- un décodeur binaire-7 segments qui, en fonction du chiffre, allume les bons segments de l'afficheur.



Ici, on s'intéresse au décodeur binaire-7 segments. Noter que ce circuit est combinatoire.

Il sera possible de le tester en entrant un chiffre compris entre 0 et 9, codé sur 4 bits, avec des switchs. On utilisera l'afficheur HEX0 pour la visualisation.



Créer un nouveau fichier vhdl, avec le nom `dec_7seg` (ce nom devra être celui de l'entité), et le définir comme entité de plus haut niveau.

Puis, saisir la description ci-dessous. Certaines lignes de l'architecture seront sans doute obscures, mais le sens global est facile à deviner. Vous comprendrez tout dans le détail à la fin de ce TP.

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity dec_7seg is
        port(
                digit : in integer range 0 to 15;
                segments : out std_logic_vector(6 downto 0)
        );
end;

architecture behave of dec_7seg is
begin
        with digit select
                segments <=
                        "1000000" when 0,
                        "1111001" when 1,

                        -- à compléter


                        "1111111" when others;
end;
```

La compléter pour traiter les cas où l'entrée vaut 2, 3, ..., jusqu'à 9. Noter que c'est un 0 qui allume le segment correspondant. Puis suivre la procédure du TP1 : analyse et synthèse (`CTRLE+K`), `Pin Planner`, compilation (`CTRLE+L`), chargement du programme dans le FPGA, test.

2.3 Instructions concurrentes

Considérons à nouveau le décodeur binaire-7 segments. Supposons qu'on veuille une sortie supplémentaire, sur un seul bit, qui passe à 1 en cas d'erreur sur l'entrée, c'est-à-dire si celle-ci est supérieure à 9. Pour cela, on déclarerait une sortie appelée par exemple `error`, de type `std_logic`, et on ajouterait dans l'architecture quelques lignes (NE PAS LE FAIRE ICI, c'est un exemple purement pédagogique) :

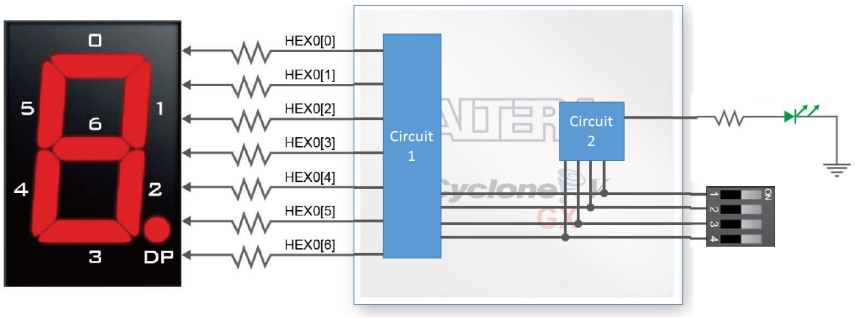
```
architecture behave of dec_7seg is
begin
        with digit select
                segments <=
                        "1000000" when 0,
                        "1111001" when 1,
                        ...
                        "1111111" when others;

        error <=
                '1' when digit > 9 else
                '0';
```

Les deux instructions (celle qui calcule `digit` et celle qui calcule `error`) sont des exemples d'instructions *concurrentes*.

Dans une architecture, des instructions concurrentes s'exécutent :

- en permanence ;
- en même temps.

Dans le FPGA, les instructions de notre décodeur seront chacune implantées sous forme d'un bloc logique, et pourront éventuellement être localisées dans des zones distinctes :



Il est fondamental de bien comprendre la différence entre la programmation classique d'un CPU (software programmation), et la description d'un FPGA (hardware description). Dans le premier cas, on a affaire à circuit intégré dont la structure matérielle est plus ou moins complexe, mais figée, gravée dans le silicium : une unité de calcul (ALU), des registres, un circuit de contrôle, de la mémoire, etc. La programmation aboutit (après compilation) à la modification de la mémoire programme. Celle-ci exécute alors de manière séquentielle (une par une) une suite d'instructions.

Dans le second cas, la compilation aboutit à la configuration des ressources logiques (de la structure matérielle) du FPGA.

Nous décrivons ci-dessous les principales instructions concurrentes. Notons que l'instanciation d'entités est également uns instruction concurrente – les différents modules d'un circuit doivent s'exécuter en parallèle – mais nous la traiterons à part.

2.3.1 Instructions d'assignation concurrente conditionnelle et concurrente sélective

On les utilise pour décrire des circuits (ou des portions de circuits) combinatoires, comme notre décodeur.

L'*assignation simple* assigne le résultat d'une *expression*, typiquement, une opération logique ou arithmétique, à la cible :

```
target <= expression;
```

Nous l'avons utilisée pour décrire la porte XOR. En général, toutefois, on préfère utiliser l'une des deux autres assignations, qui rendent le code plus explicite et ne nécessitent pas de calcul d'équation logique.

L'*assignation concurrente conditionnelle* a le format :

```
target <=
        value_1 when condition_1 else
        value_2 when condition_2 else
        default_value;
```

C'est l'assignation que nous avons utilisée ci-dessus pour la sortie `error`. La condition doit donner un résultat vrai ou faux : c'est typiquement un test d'égalité (opérateur `=`), de supériorité, etc. La cible prend la valeur correspondant à la première condition vraie rencontrée (la plupart du temps, on donne des conditions exclusives).

L'*assignation sélective* a le format :

```
with expression select
        target <=
                value_1 when choice_1,
                value_2 when choice_2,
                default_value when others;
```

Cette fois, c'est une expression pouvant prendre plusieurs valeurs qui permet de choisir la valeur de la cible. Cette expression doit être un signal (entrée par exemple). Cette assignation a été utilisée pour la sortie `digit`.

2.3.2 L'instruction process

L'utilisation de process est incontournable en VHDL, et il est fondamental de bien comprendre son fonctionnement. Un *process* est une partie de code *séquentiel* dans une architecture. Par séquentiel, on entend ici que les instructions utilisées dans un process sont obligatoirement des instructions séquentielles (que nous décrivons un peu plus loin).

Bien que le process contienne des instructions séquentielles, il est lui même une instruction concurrente, dans la mesure où il s'exécute en parallèle avec les autres process ou instructions concurrentes de l'architecture.

Un process peut être dans deux états : *suspendu* ou *en train de s'exécuter*. La condition qui permet à un process de passer du premier au second état peut être écrite de deux manières : soit à l'aide d'une *liste de sensibilité*, soit à l'aide d'une *instruction wait*.

Un process a le format suivant :

```
process (signal_1, signal_2, ...)
-- ici des déclarations (typiquement de variables)
begin
 -- ici des assignations simples ou instructions séquentielles
end process;
```

La liste de signaux placée entre parenthèses immédiatement après le mot process est la liste de sensibilité ; elle est optionnelle. Si elle est absente, le process doit impérativement posséder une instruction `wait`.

Si le process utilise une liste de sensibilité, il commence à s'exécuter *quand l'un des signaux de cette liste change* (par exemple passe de 0 à 1). Les instructions séquentielles situées dans le process s'exécutent, précisément, *de façon séquentielle* (les unes à la suite des autres). Quand la dernière instruction du process est atteinte, le process recommence au début.

L'exécution est suspendue :

- quand on rencontre une instruction `wait` ;
- dans le cas de l'utilisation d'une liste de sensibilité, après avoir exécuté la dernière instruction (un process utilisant une liste de sensibilité possède un `wait` implicite tout en bas).

Les process sont typiquement utilisés pour décrire des circuits logiques séquentiels. Supposons par exemple que l'on souhaite décrire un circuit dont la sortie devienne égale, au front montant d'horloge, à l'inverse de l'entrée (une bascule JK avec J et K à 1). On utilise un process avec le signal d'horloge dans la liste de sensibilité :

```
process (clock) begin
        if rising_edge(clock) then
                q <= not(q);
        end if;
end process;
```

A chaque passage de l'horloge de 0 à 1 (front montant) ou de 1 à 0 (front descendant) le process, commence à s'exécuter. L'instruction `if ...else ...end if;` est une instruction séquentielle que nous décrirons plus loin ; elle permet de tester une condition – ici le résultat de la fonction

`rising_edge` (définie dans `std_logic_1164`) pour sélectionner le front montant. Une fois l'assignation effectuée, le process arrive en bas, est suspendu, et ne sera de nouveau exécuté qu'au prochain front de `clock`.

Un process équivalent, utilisant l'instruction `wait` serait :

```
process begin
        wait until rising_edge(clock);
        q <= not(q);
end process;
```

Un process peut être utilisé également pour décrire un circuit combinatoire. Par exemple, pour la sortie `error` de notre décodeur, on peut remplacer l'affectation concurrente conditionnelle par un process :

```
process (digit) begin
        if digit > 9 then
                error <= '1';
        else
                error <= '0';
        end if;
end process;
```

Dès que `digit` change, le process est réactivé et la sortie `error` est recalculée.

2.3.3 L'instruction generate

Elle se décline en deux versions. L'instruction *if-generate* inclut ou exclut des parties de hardware en fonction du résultat d'un test. La condition *est évaluée à la compilation*, pour que seul le hardware nécessaire soit synthétisé ; elle porte typiquement sur une constante (une option de configuration par exemple).

```
if condition_1 generate
        --instructions concurrentes;
elsif condition_2 generate
        --instructions concurrentes;
else generate
        --instructions concurrentes;
end generate;
```

Attention à ne pas la confondre avec l'instruction séquentielle conditionnelle `if ...then ...else` qui ne peut figurer que dans les process (voir la suite), et dont la condition porte sur un signal qui peut varier pendant l'exécution (typiquement, une entrée d'horloge ou une entrée d'un circuit combinatoire).

L'instruction *for-generate* permet de recopier des parties du hardware (pour éviter d'avoir à taper les mêmes lignes un grand nombre de fois) :

```
for generate_parameter in generate_range generate
        instructions concurrentes;
end generate;
```

2.4 Opérateurs et opérandes

Pour le calcul d'expressions, VHDL utilise :

— des opérateurs logiques : `and`, `or`, `nand`, `nor`, `not`, etc. Celles-ci portent typiquement sur des opérandes de type `bit` ou `std_logic`, ou des arrays de `bits` ou `std_logic` (`std_logic_vector`, `unsigned`, `signed`)

— des opérateurs arithmétiques : addition (+), soustraction (-) multiplication (*), division (/), reste d'une division (`mod`). Il est important de réaliser qu'une addition ou multiplication sera synthétisée sous forme d'un circuit logique *qui peut être différent selon le type des opérandes* ;

— des opérateurs de décalage et de rotation de mots binaires : `sll`, `srl`, `rol`, `ror`.

Pour les évaluations de conditions, on utilise des opérateurs relationnels : d'égalité (=), d'inégalité (/=), de comparaison (>, >, <=, >=).

Les opérandes peuvent être des variables, des signaux ou des constantes (définis un peu plus loin), ou alors être donnés en notation littérale. Les notations littérales que nous utiliserons le plus souvent sont :

— numeric literal, pour le type `integer`. Exemple : 52
— character literal, pour le type `bit` ou `std_logic`. Exemple : '1'
— string literal, pour les arrays de `bits` ou `std_logic` (`std_logic_vector`, `signed`, `unsigned`). Exemple : "1100"

TP3 : le compteur

Il s'agit de créer un circuit qui compte de 0 à 9. L'évènement qui incrémente le compteur est le front montant d'horloge. Quand le compteur vaut 9 et que survient un nouveau front montant, il doit être remis à 0.

Créer un nouveau fichier vhdl, avec le nom `compteur` (ce nom devra être celui de l'entité), et le définir comme entité de plus haut niveau.

Puis, saisir la description ci-dessous :

```
library ieee;
use ieee.std_logic_1164.all;

entity compteur is
   port(
      clock : in std_logic;
      digit : out integer range 0 to 9
   );
end;

architecture behav of compteur is
   signal count : integer range 0 to 9;
begin
   process(clock)
   begin
      if rising_edge(clock) then
         if count < 9 then
            count <= count + 1;
         else
            count <= 0;
         end if;
      end if;
   end process;

   digit <= count;
end;
```

Est-il possible d'utiliser un autre type pour `digit` ?

A quoi sert le signal `count` ? (Pourquoi ne pas incrémenter et remettre à zéro directement `digit` ?)

(La réponse à la seconde question se trouve à la section 2.1).

Lancer l'étape d'analyse et synthèse. Faire toutes les étapes pour tester ce module, en utilisant un bouton comme horloge et des leds comme sorties.

2.5 Constantes, signaux et variables

Une *constante* est créée dans la partie déclarative d'une architecture ou d'un process de la façon suivante :

```
constant CONSTANT_NAME : type_or_subtype_name := value;
```

L'utilisation de constantes aux noms explicites plutôt que de valeurs littérales rend le code plus lisible. On utilise assez souvent le type `integer`, ou son sous-type `natural` (entier positif ou nul) en restreignant l'intervalle avec `range`.

Les *signaux* sont utilisés pour la communication entre process ou entre design entities. Les entrées/sorties d'une entité sont des signaux. On est parfois amené à déclarer d'autres signaux que les entrées/sorties, typiquement pour y stocker la valeur d'un noeud d'un circuit. Cela se fait dans la zone déclarative de l'architecture, de la façon suivante :

```
signal signal_name : type_or_subtype_name;
```

Si le type du signal est `std_logic_vector`, `signed` ou `unsigned`, il faut obligatoirement préciser sa dimension.

Les signaux peuvent être utilisés avec des instructions concurrentes (hors process) ou séquentielles (dans un process). On leur assigne une valeur par le symbole `<=`

Enfin, les signaux possèdent une caractéristique très particulière lorsqu'ils sont utilisés dans un process. Lorsqu'on leur assigne une nouvelle valeur, le changement ne se fait qu'à la suspension du process. La valeur d'un signal est donc toujours la valeur qu'il avait au moment où le process a commencé à s'exécuter. Considérons par exemple le process suivant, où `original` et `copy` sont des signaux :

```
process begin
        wait until rising_edge(clock);
        original <= original + 1;
        copy <= original;
end process;
```

Le signal `copy` aura toujours un cycle de retard sur `original`.

A la différence d'un signal, une *variable* ne peut être utilisée que dans un process. Elle ne permet donc pas la communication entre le process et le reste de l'entité, par exemple. Autre différence majeure, une variable, lorsqu'on lui assigne une valeur, change d'état *instantanément* (et pas à la suspension du process), et peut donc être utilisée de la même façon que les variables des langages de programmation classique. L'assignation se fait avec le symbole `:=`

On déclare une variable dans la partie déclarative d'un process, de la façon suivante :

```
variable variable_name : type_or_subtype_name;
```

Le tableau suivant résume les principales différences entre signaux et variables.

Signal	Variable
Peut être utilisé dans des instructions concurrentes ou séquentielles	Ne peut être utilisée qu'avec des instructions séquentielles (process ou corps de package)
Utilisé pour la communication entre plusieurs instructions concurrentes (y compris des process) ou pour transporter des valeurs entre deux modules (deux design entities)	Utilisé pour un usage local dans un process (ou un sous-programme dans un package)
Dans un process, la valeur lue est la valeur qu'avait le signal au moment de l'activation du process. Tout changement de la valeur d'un signal ne sera effectif qu'à la suspension du process	Tout changement de la valeur d'une variable est effectif immédiatement. La valeur lue est la valeur courante. Utilisable comme une variable dans un langage de programmation classique

Dans l'entité compteur, est-il possible d'utiliser une variable plutôt qu'un signal ?

2.6 Instructions séquentielles

Dans un process, on peut trouver des assignations simples, ou des instructions séquentielles. A la différence des instructions concurrentes, qui s'exécutent toutes en même temps, les instructions séquentielles sont exécutées à la suite les unes des autres.

2.6.1 Instruction séquentielle conditionnelle et sélective

L'instruction séquentielle conditionnelle a le format suivant :

```
if expression then
        instruction_1;
        instruction_2;
elsif condition_2 then
        instruction_3;
else
        instruction_4;
end if;
```

Les `instruction_1`, `instruction_2`, etc, sont des affectations simples ou d'autres instructions séquentielles imbriquées. Le `elsif` est optionnel et permet de limiter les imbrications.

Cette instruction est typiquement utilisée au début des process lorsque l'on synthétise un circuit séquentiel synchrone, pour discriminer le front montant du descendant.

L'instruction séquentielle sélective est proche de la précédente. Son format est :

```
case expression is
        when value_1 =>
                instruction_1;
                instruction_2;
        when value_2 =>
                instruction_3;
        when others =>
                instruction_4;
end case;
```

2.6.2 Instruction Loop

Cette instruction permet typiquement de générer des circuits constitués de bascules D opérant en parallèle ou en série. Il existe plusieurs syntaxes ; la plus courante est :

```
for i in discrete_range loop
        statement_1;
        statement_2;
end loop;
```

L'intervalle `discrete_range` des valeurs prises par l'index `i` peut être, par exemple : `1 to 10`.

2.6.3 Instruction Wait

L'instruction `wait` suspend l'exécution d'un process jusqu'à ce qu'une condition soit remplie. On rencontre souvent la forme suivante :

```
wait on liste_de_sensibilité;
```

Le process est suspendu quand il atteint cette instruction. Un changement de niveau de l'un des signaux de la liste de sensibilité (ce qu'on appelle un *évènement* en vhdl) entraîne la reprise de l'exécution. Exemple :

```
wait on reset, clock;
```

L'autre forme est :

```
wait until condition;
```

Dans ce cas, il y a une liste de sensibilité implicite, dans laquelle figurent tous les signaux (mais pas les variables) dont dépend la condition. Le process continue à s'exécuter si l'un des signaux de cette liste change, si au même instant la condition est vraie.

TP4 : le générateur d'impulsions périodiques

Le générateur de pulses, à partir d'une horloge de fréquence 50 MHz, génère toutes les secondes une impulsion de durée 20 ns.

Créer un nouveau fichier vhdl, avec le nom `gen_pulse` (ce nom devra être celui de l'entité), et le définir comme entité de plus haut niveau.

Puis, saisir la description ci-dessous.

```
library ieee;
use ieee.std_logic_1164.all;

entity gen_pulse is
port(
clock : in std_logic;
pulse : out std_logic
);
end;

architecture behave of gen_pulse is
begin
process
constant COUNT_MAX : natural := 50e6-1;
variable count : natural range 0 to COUNT_MAX;
begin
wait until rising_edge(clock);
if count < COUNT_MAX then
count := count+1;
pulse <= '0';
else
count := 0;
pulse <= '1';
end if;
end process;
end;
```

Pourquoi peut-on utiliser ici une variable pour `count` ? (C'est une bonne pratique d'utiliser si possible une `variable` plutôt qu'un `signal`)

Lancer l'étape d'analyse et synthèse (CTRLE+K). Continuer les étapes suivantes (jusqu' à l'implantation dans le FPGA) .

2.7 Instanciation d'entités

L'instanciation d'entité est une instruction concurrente, donc elle ne peut pas figurer dans un process.

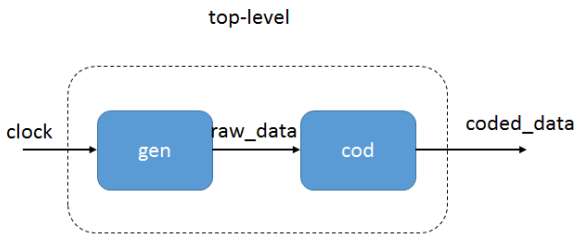
Elle peut se faire de plusieurs façons ; nous en traiterons une seule, dite *instanciation directe* :

```
label : entity work.entity_name generic map(...) port map(...);
```

Le label est obligatoire. La partie `generic map(...)` ne figure que si l'entité instanciée possède des paramètres. Entre parenthèses derrière `generic map` et `port map`, on donne la correspondance (ou connexion) entre un élément du module instancié (à gauche), et un élément du module instanciant (à droite) :

```
(instancié_1 => instanciant_1, instancié_2 => instanciant_2, ...)
```

Mettons par exemple que l'on ait un fichier top-level contenant 2 modules appelés `gen` (entrée : `clk`, sortie : `data`) et `cod` (entrée : `uncoded`, sortie : `coded`), selon le schéma ci-dessous :



On créé un fichier appelé (par exemple) `top_example.vhd` contenant une entité appelée `top_example`, déclarant une entrée `clock` et une sortie `coded_data`. Le code de l'architecture est donné ci-dessous ; noter qu'il faut déclarer un `signal` appelé ici `raw_data` pour connecter les deux modules.

```
architecture structural of top_example is
   signal raw_data : std_logic;
begin
module1 : entity work.gen port map(clk => clock, data => raw_data);
module2 : entity work.cod port map(uncoded => raw_data, coded => coded_data);
end;
```

TP5 : le compteur 1 digit complet

Créer un nouveau fichier et le définir comme top-level. Déclarer l'entrée d'horloge et la sortie sur 7 bits permettant de commander l'afficheur, et instancier les 3 modules. Effectuer toutes les étapes pour implanter le circuit dans un FPGA.

BONUS 1 :
Modifier le compteur afin qu'il compte de 0 à 99, en utilisant pour la sortie le code BCD (Binary Coded Decimal). Pour cela, déclarer deux sorties entières entre 0 et 9 (vous pouvez les appeler `dizaine` et `unite`) et modifier le code en conséquence.

BONUS 2 :
Ajouter au compteur une entrée de Reset, de type `std_logic`. Noter que sur notre carte, quand on appuie sur un bouton, la pin correspondante est à 0.
Modifier la liste de sensibilité du process pour inclure cette nouvelle entrée. Dans le process, modifier l'instruction conditionnelle pour discriminer le cas `Reset='0'` (à traiter dans l'alternative `if`) du front d'horloge (à traiter par un `elsif`). Le Reset est-il synchrone ou asynchrone ?