eBook Gratuit

APPRENEZ vhdl

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#vhdl

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: vhdl

It is an unofficial and free vhdl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official vhdl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec vhdl

Remarques

VHDL est un acronyme composé de VHSIC (circuit intégré à très haut débit) HDL (Hardware Description Language). En tant que langage de description du matériel, il est principalement utilisé pour décrire ou modéliser des circuits. Le langage VHDL est un langage idéal pour décrire des circuits car il offre des constructions de langage qui décrivent facilement les comportements simultanés et séquentiels, ainsi qu'un modèle d'exécution qui supprime les ambiguïtés introduites lors de la modélisation d'un comportement concurrent.

Le VHDL est généralement interprété dans deux contextes différents: pour la simulation et pour la synthèse. Lorsqu'il est interprété pour la synthèse, le code est converti (synthétisé) en éléments matériels équivalents qui sont modélisés. Seul un sous-ensemble du VHDL est généralement disponible pour être utilisé pendant la synthèse, et les constructions de langage prises en charge ne sont pas normalisées. c'est une fonction du moteur de synthèse utilisé et du matériel cible. Lorsque VHDL est interprété pour la simulation, toutes les constructions de langage sont disponibles pour modéliser le comportement du matériel.

Versions

Version	Date de sortie
IEEE 1076-1987	1988-03-31
IEEE 1076-1993	1994-06-06
IEEE 1076-2000	2000-01-30
IEEE 1076-2002	2002-05-17
IEEE 1076c-2007	2007-09-05
IEEE 1076-2008	2009-01-26

Examples

Installation ou configuration

Un programme VHDL peut être simulé ou synthétisé. La simulation est ce qui ressemble le plus à l'exécution dans d'autres langages de programmation. Synthesis traduit un programme VHDL en un réseau de portes logiques. De nombreux outils de simulation et de synthèse VHDL font partie des suites EDA (Electronic Design Automation) commerciales. Ils gèrent également fréquemment d'autres langages de description du matériel (HDL), tels que Verilog, SystemVerilog ou SystemC.

Certaines applications libres et open source existent.

Simulation VHDL

GHDL est probablement le simulateur VHDL libre et open source le plus mature. Il existe trois `llvm` différentes en fonction du backend utilisé: `gcc` , `llvm` ou `mcode` . Les exemples suivants montrent comment utiliser GHDL (version `mcode`) et Modelsim, le simulateur HDL commercial de Mentor Graphics, sous un système d'exploitation GNU / Linux. Les choses seraient très similaires avec d'autres outils et d'autres systèmes d'exploitation.

Bonjour le monde

Créez un fichier `hello_world.vhd` contenant:

```
-- File hello_world.vhd
entity hello_world is
end entity hello_world;

architecture arc of hello_world is
begin
  assert false report "Hello world!" severity note;
end architecture arc;
```

Une unité de compilation VHDL est un programme VHDL complet qui peut être compilé seul. *Les entités* sont des unités de compilation VHDL utilisées pour décrire l'interface externe d'un circuit numérique, c'est-à-dire ses ports d'entrée et de sortie. Dans notre exemple, l' `entity` est nommée `hello_world` et est vide. Le circuit que nous modélisons est une boîte noire, il n'a pas d'entrée ni de sortie. *Les architectures* sont un autre type d'unité de compilation. Ils sont toujours associés à une `entity` et servent à décrire le comportement du circuit numérique. Une *entité* peut avoir une ou plusieurs *architectures* pour décrire le comportement de l'entité. Dans notre exemple, l' *entité* est associée à une seule *architecture* nommée `arc` qui ne contient qu'une seule déclaration VHDL:

```
  assert false report "Hello world!" severity note;
```

La déclaration sera exécutée au début de la simulation et imprimera le `Hello world!` message sur la sortie standard. La simulation prendra alors fin car il n'y a plus rien à faire. Le fichier source VHDL que nous avons écrit contient deux unités de compilation. Nous aurions pu les séparer dans deux fichiers différents, mais nous n'aurions pas pu les séparer dans des fichiers différents: une unité de compilation doit être entièrement contenue dans un fichier source. Notez que cette architecture ne peut pas être synthétisée car elle ne décrit pas une fonction pouvant être directement traduite en portes logiques.

Analysez et exécutez le programme avec GHDL:

```
$ mkdir gh_work
$ ghdl -a --workdir=gh_work hello_world.vhd
$ ghdl -r --workdir=gh_work hello_world
```

```
hello_world.vhd:6:8:@0ms:(assertion note): Hello world!
```

Le répertoire `gh_work` est l'endroit où GHDL stocke les fichiers qu'il génère. C'est ce que dit l'option `--workdir=gh_work` . La phase d'analyse vérifie l'exactitude de la syntaxe et produit un fichier texte décrivant les unités de compilation trouvées dans le fichier source. La phase d'exécution compile, relie et exécute le programme. Notez que, dans la version `mcode` de GHDL, aucun fichier binaire n'est généré. Le programme est recompilé chaque fois que nous le simulons. Les versions `gcc` ou `llvm` se comportent différemment. Notez également que `ghdl -r` ne prend pas le nom d'un fichier source VHDL, comme `ghdl -a` does, mais le nom d'une unité de compilation. Dans notre cas, nous lui passons le nom de l' `entity` . Comme il n'y a qu'une seule `architecture` associée, il n'est pas nécessaire de spécifier celle à simuler.

Avec Modelsim:

```
$ vlib ms_work
$ vmap work ms_work
$ vcom hello_world.vhd
$ vsim -c hello_world -do 'run -all; quit'
...
# ** Note: Hello world!
#    Time: 0 ns  Iteration: 0  Instance: /hello_world
...
```

`vlib` , `vmap` , `vcom` et `vsim` sont quatre commandes fournies par Modelsim. `vlib` crée un répertoire (`ms_work`) où les fichiers générés seront stockés. `vmap` associe un répertoire créé par `vlib` à un nom logique (`work`). `vcom` compile un fichier source VHDL et, par défaut, stocke le résultat dans le répertoire associé au nom logique de `work` . Enfin, `vsim` simule le programme et produit le même type de sortie que GHDL. Notez à nouveau que ce `vsim` demande `vsim` n'est pas un fichier source mais le nom d'une unité de compilation déjà compilée. L'option `-c` indique au simulateur de s'exécuter en mode ligne de commande au lieu du mode interface utilisateur graphique (GUI) par défaut. L'option `-do` est utilisée pour passer un script TCL à exécuter après le chargement du design. TCL est un langage de script très fréquemment utilisé dans les outils EDA. La valeur de l'option `-do` peut être le nom d'un fichier ou, comme dans notre exemple, une chaîne de commandes TCL. `run -all; quit` ordonne au simulateur d'exécuter la simulation jusqu'à ce qu'il se termine naturellement - ou pour toujours s'il dure éternellement - puis de quitter.

Compteur synchrone

```
-- File counter.vhd
-- The entity is the interface part. It has a name and a set of input / output
-- ports. Ports have a name, a direction and a type. The bit type has only two
-- values: '0' and '1'. It is one of the standard types.
entity counter is
  port(
    clock: in  bit;    -- We are using the rising edge of CLOCK
    reset: in  bit;    -- Synchronous and active HIGH
    data:  out natural -- The current value of the counter
  );
end entity counter;

-- The architecture describes the internals. It is always associated
```

```
-- to an entity.
architecture sync of counter is
  -- The internal signals we use to count. Natural is another standard
  -- type. VHDL is not case sensitive.
  signal current_value: natural;
  signal NEXT_VALUE:    natural;
begin
  -- A process is a concurrent statement. It is an infinite loop.
  process
  begin
    -- The wait statement is a synchronization instruction. We wait
    -- until clock changes and its new value is '1' (a rising edge).
    wait until clock = '1';
    -- Our reset is synchronous: we consider it only at rising edges
    -- of our clock.
    if reset = '1' then
      -- <= is the signal assignment operator.
      current_value <= 0;
    else
      current_value <= next_value;
    end if;
  end process;

  -- Another process. The sensitivity list is another way to express
  -- synchronization constraints. It (approximately) means: wait until
  -- one of the signals in the list changes and then execute the process
  -- body. Sensitivity list and wait statements cannot be used together
  -- in the same process.
  process(current_value)
  begin
    next_value <= current_value + 1;
  end process;

  -- A concurrent signal assignment, which is just a shorthand for the
  -- (trivial) equivalent process.
  data <= current_value;
end architecture sync;
```

Bonjour le monde

Il existe de nombreuses manières d'imprimer le classique "Hello world!" message dans VHDL. Le plus simple est probablement quelque chose comme:

```
-- File hello_world.vhd
entity hello_world is
end entity hello_world;

architecture arc of hello_world is
begin
  assert false report "Hello world!" severity note;
end architecture arc;
```

Un environnement de simulation pour le compteur synchrone

Environnements de simulation

Un environnement de simulation pour une conception VHDL (Design Under Test ou DUT) est une autre conception VHDL qui, au minimum:

- Déclare des signaux correspondant aux ports d'entrée et de sortie du DUT.
- Instancie le DUT et connecte ses ports aux signaux déclarés.
- Instancie les processus qui pilotent les signaux connectés aux ports d'entrée du DUT.

En option, un environnement de simulation peut instancier d'autres conceptions que le DUT, comme par exemple les générateurs de trafic sur les interfaces, les moniteurs pour vérifier les protocoles de communication, les vérificateurs automatiques des sorties du DUT ...

L'environnement de simulation est analysé, élaboré et exécuté. La plupart des simulateurs offrent la possibilité de sélectionner un ensemble de signaux à observer, de tracer leurs formes d'onde graphiques, de placer des points d'arrêt dans le code source, de saisir le code source ...

Idéalement, un environnement de simulation devrait être utilisable en tant que test de non-régression robuste, c'est-à-dire qu'il devrait automatiquement détecter les violations des spécifications DUT, signaler les messages d'erreur utiles et garantir une couverture raisonnable des fonctionnalités DUT. Lorsque de tels environnements de simulation sont disponibles, ils peuvent être réexécutés à chaque changement du dispositif sous test pour vérifier qu'il est toujours fonctionnellement correct, sans qu'il soit nécessaire de procéder à des inspections visuelles et fastidieuses des traces de simulation.

Dans la pratique, la conception d'environnements de simulation idéaux, voire simples, est un défi. Il est souvent aussi difficile, voire plus, que de concevoir le DUT lui-même.

Dans cet exemple, nous présentons un environnement de simulation pour l'exemple de compteur synchrone . Nous montrons comment l'exécuter en utilisant GHDL et Modelsim et comment observer des formes d'onde graphiques en utilisant GTKWave avec GHDL et le visualiseur de forme d'onde intégré avec Modelsim. Nous discutons ensuite d'un aspect intéressant des simulations: comment les arrêter?

Un premier environnement de simulation pour le compteur synchrone

Le compteur synchrone possède deux ports d'entrée et un port de sortie. Un environnement de simulation très simple pourrait être:

```
-- File counter_sim.vhd
-- Entities of simulation environments are frequently black boxes without
-- ports.
entity counter_sim is
end entity counter_sim;

architecture sim of counter_sim is

  -- One signal per port of the DUT. Signals can have the same name as
  -- the corresponding port but they do not need to.
```

```
  signal clk:  bit;
  signal rst:  bit;
  signal data: natural;

begin

  -- Instantiation of the DUT
  u0: entity work.counter(sync)
  port map(
    clock => clk,
    reset => rst,
    data  => data
  );

  -- A clock generating process with a 2ns clock period. The process
  -- being an infinite loop, the clock will never stop toggling.
  process
  begin
    clk <= '0';
    wait for 1 ns;
    clk <= '1';
    wait for 1 ns;
  end process;

  -- The process that handles the reset: active from beginning of
  -- simulation until the 5th rising edge of the clock.
  process
  begin
    rst  <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst  <= '0';
    wait; -- Eternal wait. Stops the process forever.
  end process;

end architecture sim;
```

Simuler avec GHDL

Laissez-nous compiler et simuler cela avec GHDL:

```
$ mkdir gh_work
$ ghdl -a --workdir=gh_work counter_sim.vhd
counter_sim.vhd:27:24: unit "counter" not found in 'library "work"'
counter_sim.vhd:50:35: no declaration for "rising_edge"
```

Alors les messages d'erreur nous disent deux choses importantes:

- L'analyseur GHDL a découvert que notre conception instancie une entité nommée `counter` mais cette entité n'a pas été trouvée dans le `work` bibliothèque. C'est parce que nous n'avons pas compilé de `counter` avant `counter_sim` . Lors de la compilation de conceptions VHDL qui instancient des entités, les niveaux inférieurs doivent toujours être compilés avant les niveaux supérieurs (les conceptions hiérarchiques peuvent également être compilées de haut en bas, mais uniquement si elles instancient des `component` et non des entités).

- La fonction `rising_edge` utilisée par notre conception n'est pas définie. Cela est dû au fait que cette fonction a été introduite dans VHDL 2008 et nous n'avons pas dit à GHDL d'utiliser cette version du langage (par défaut, il utilise VHDL 1993 avec une tolérance de la syntaxe VHDL 1987).

Laissez-nous réparer les deux erreurs et lancez la simulation:

```
$ ghdl -a --workdir=gh_work --std=08 counter.vhd counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
^C
```

Notez que l'option `--std=08` est nécessaire pour l'analyse **et la** simulation. Notez également que nous avons lancé la simulation sur l'entité `counter_sim` , architecture `sim` , pas sur un fichier source.

Comme notre environnement de simulation a un processus sans fin (le processus qui génère l'horloge), la simulation ne s'arrête pas et nous devons l'interrompre manuellement. Au lieu de cela, nous pouvons spécifier une heure d'arrêt avec l'option `--stop-time` :

```
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim --stop-time=60ns
ghdl:info: simulation stopped by --stop-time
```

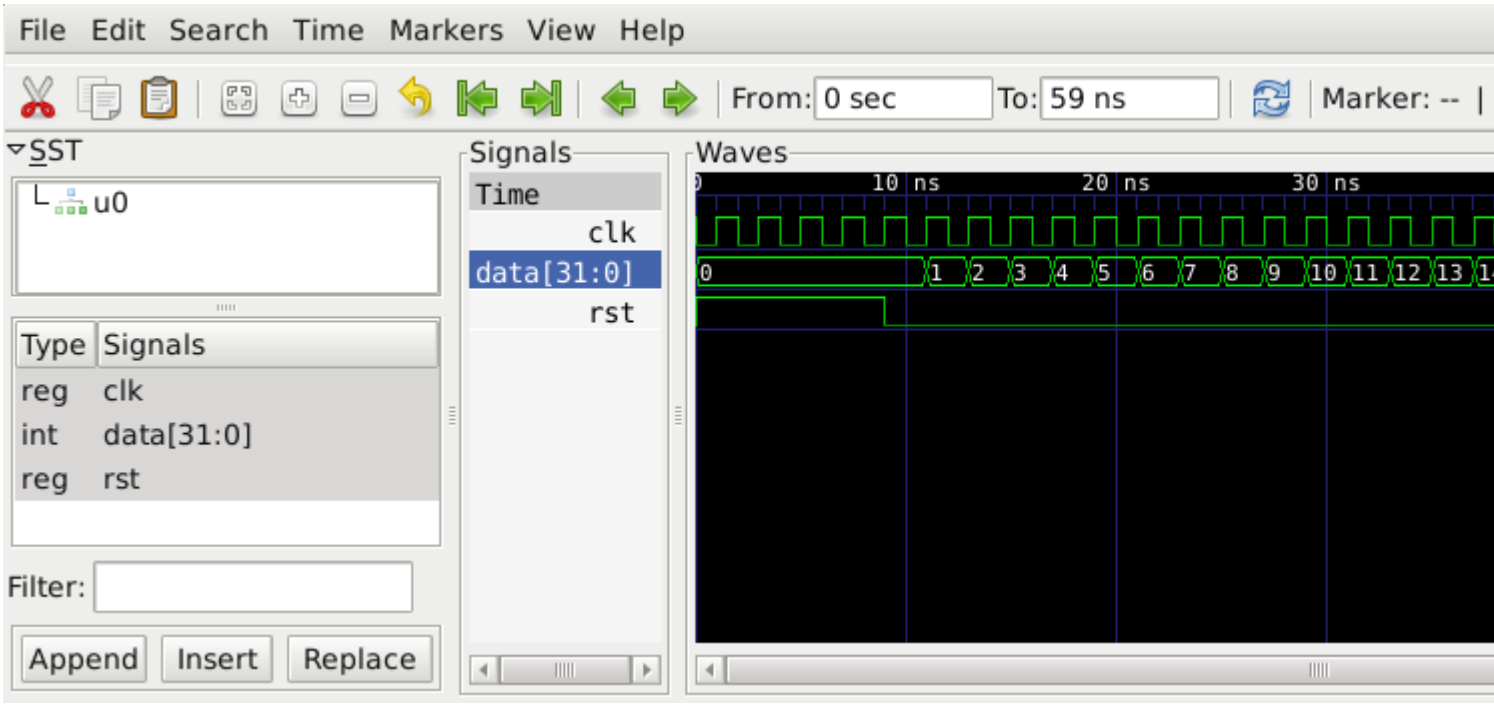
En l'état, la simulation ne nous en dit pas beaucoup sur le comportement de notre DUT. Vidons les changements de valeur des signaux dans un fichier:

```
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim --stop-time=60ns --vcd=counter_sim.vcd
Vcd.Avhpi_Error!
ghdl:info: simulation stopped by --stop-time
```

(ignorez le message d'erreur, c'est quelque chose qui doit être corrigé dans GHDL et cela n'a aucune conséquence). Un fichier `counter_sim.vcd` a été créé. Il contient au format VCD (ASCII) tous les changements de signal pendant la simulation. GTKWave peut nous montrer les formes d'onde graphiques correspondantes:

```
$ gtkwave counter_sim.vcd
```

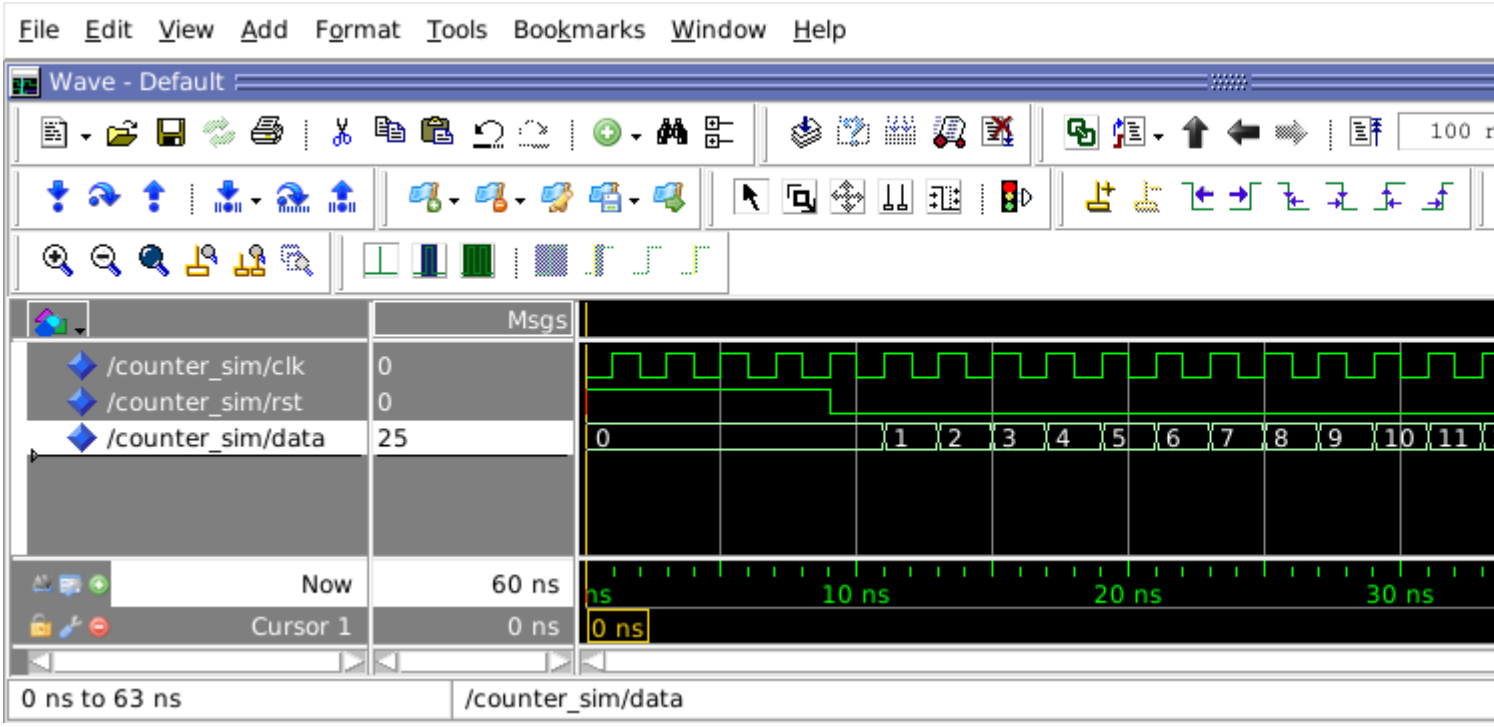
où on peut voir que le compteur fonctionne comme prévu.



Simuler avec Modelsim

Le principe est exactement le même avec Modelsim:

```
$ vlib ms_work
...
$ vmap work ms_work
...
$ vcom -nologo -quiet -2008 counter.vhd counter_sim.vhd
$ vsim -voptargs="+acc" 'counter_sim(sim)' -do 'add wave /*; run 60ns'
```



Notez l' `-voptargs="+acc"` transmise à `vsim` : elle empêche le simulateur d'optimiser le signal de `data` et nous permet de le voir sur les formes d'onde.

Finissant gracieusement les simulations

Avec les deux simulateurs, nous avons dû interrompre la simulation sans fin ou spécifier un temps d'arrêt avec une option dédiée. Ce n'est pas très pratique. Dans de nombreux cas, l'heure de fin d'une simulation est difficile à prévoir. Il serait préférable d'arrêter la simulation depuis l'intérieur du code VHDL de l'environnement de simulation, lorsqu'une condition particulière est atteinte, par exemple lorsque la valeur actuelle du compteur atteint 20%. Ceci peut être réalisé avec une assertion dans le code. processus qui gère la réinitialisation:

```
  process
  begin
    rst <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst <= '0';
    loop
      wait until rising_edge(clk);
      assert data /= 20 report "End of simulation" severity failure;
    end loop;
  end process;
```

Tant que les `data` diffèrent de 20, la simulation se poursuit. Lorsque les `data` atteignent 20, la simulation se bloque avec un message d'erreur:

```
$ ghdl -a --workdir=gh_work --std=08 counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
counter_sim.vhd:90:24:@51ns:(assertion failure): End of simulation
ghdl:error: assertion failed
  from: process work.counter_sim(sim2).P1 at counter_sim.vhd:90
ghdl:error: simulation failed
```

Notez que nous avons recompilé uniquement l'environnement de simulation: c'est le seul design qui a changé et c'est le niveau supérieur. Si nous n'avions modifié que `counter.vhd` , nous aurions dû recompiler les deux: `counter.vhd` car il a changé et `counter_sim.vhd` parce que cela dépend de `counter.vhd` .

Briser la simulation avec un message d'erreur n'est pas très élégant. L'analyse automatique des messages de simulation peut également poser un problème pour décider si un test de non-régression automatique est réussi ou non. Une solution meilleure et beaucoup plus élégante consiste à arrêter tous les processus lorsqu'une condition est atteinte. Cela peut être fait, par exemple, en ajoutant un signal `boolean` fin de simulation (`eof`). Par défaut, il est initialisé à `false` au début de la simulation. Un de nos processus le mettra à la valeur `true` au moment de mettre fin à la simulation. Tous les autres processus surveilleront ce signal et s'arrêteront avec une `wait` éternelle quand cela deviendra `true` :

```
  signal eos:  boolean;
...
  process
  begin
    clk <= '0';
    wait for 1 ns;
    clk <= '1';
    wait for 1 ns;
    if eos then
      report "End of simulation";
      wait;
    end if;
  end process;

  process
  begin
    rst <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst <= '0';
    for i in 1 to 20 loop
      wait until rising_edge(clk);
    end loop;
    eos <= true;
    wait;
  end process;
```

```
$ ghdl -a --workdir=gh_work --std=08 counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
counter_sim.vhd:120:24:@50ns:(report note): End of simulation
```

Last but not least, il existe une solution encore meilleure introduite dans VHDL 2008 avec le package standard `env` et les procédures d' `stop` et de `finish` qu'elle déclare:

```
use std.env.all;
...
  process
  begin
    rst    <= '1';
    for i in 1 to 5 loop
      wait until rising_edge(clk);
    end loop;
    rst    <= '0';
    for i in 1 to 20 loop
      wait until rising_edge(clk);
    end loop;
    finish;
  end process;
```

```
$ ghdl -a --workdir=gh_work --std=08 counter_sim.vhd
$ ghdl -r --workdir=gh_work --std=08 counter_sim sim
simulation finished @49ns
```

Signaux vs variables, bref aperçu de la sémantique de simulation de VHDL

Cet exemple traite de l'un des aspects les plus fondamentaux du langage VHDL: la sémantique de

la simulation. Il est destiné aux débutants VHDL et présente une vue simplifiée où de nombreux détails ont été omis (processus reportés, interface de procédure VHDL, variables partagées ...) Les lecteurs intéressés par la sémantique complète réelle doivent se référer au manuel de référence du langage (LRM).

Signaux et variables

La plupart des langages de programmation impératifs classiques utilisent des variables. Ce sont des conteneurs de valeur. Un opérateur d'affectation est utilisé pour stocker une valeur dans une variable:

```
a = 15;
```

et la valeur actuellement stockée dans une variable peut être lue et utilisée dans d'autres instructions:

```
if(a == 15) { print "Fifteen" }
```

VHDL utilise également des variables et ont exactement le même rôle que dans la plupart des langages impératifs. Mais VHDL offre également un autre type de conteneur de valeur: le signal. Les signaux stockent également des valeurs, peuvent également être affectés et lus. Le type de valeurs pouvant être stockées dans les signaux est (presque) identique à celui des variables.

Alors, pourquoi avoir deux types de conteneurs de valeur? La réponse à cette question est essentielle et au cœur de la langue. Comprendre la différence entre les variables et les signaux est la première chose à faire avant d'essayer de programmer quoi que ce soit dans VHDL.

Illustrons cette différence sur un exemple concret: l'échange.

> Remarque: tous les fragments de code suivants font partie de processus. Nous verrons plus tard quels sont les processus.

```
    tmp := a;
    a   := b;
    b   := tmp;
```

échange les variables `a` et `b` . Après avoir exécuté ces 3 instructions, le nouveau contenu de `a` est l'ancien contenu de `b` et inversement. Comme dans la plupart des langages de programmation, une troisième variable temporaire (`tmp`) est nécessaire. Si, au lieu de variables, nous voulions échanger des signaux, nous écririons:

```
    r <= s;
    s <= r;
```

ou:

```
    s <= r;
```

```
    r <= s;
```

avec le même résultat et sans besoin d'un troisième signal temporaire!

> Remarque: l'opérateur d'affectation de signaux VHDL `<=` est différent de l'opérateur d'attribution de variables `:=` .

Regardons un deuxième exemple dans lequel nous supposons que le sous-programme `print` imprime la représentation décimale de son paramètre. Si `a` est une variable entière et que sa valeur actuelle est 15, en cours d'exécution:

```
    a := 2 * a;
    a := a - 5;
    a := a / 5;
    print(a);
```

imprimera:

```
5
```

Si nous exécutons cette étape par étape dans un débogueur, nous pouvons voir la valeur d' `a` changement de 15 à 30, 25 et 5.

Mais si `s` est un signal entier et que sa valeur actuelle est 15, en cours d'exécution:

```
    s <= 2 * s;
    s <= s - 5;
    s <= s / 5;
    print(s);
    wait on s;
    print(s);
```

imprimera:

```
15
3
```

Si nous exécutons cette étape par étape dans un débogueur, nous ne verrons aucune modification de valeur de `s` avant l'instruction d' `wait` . De plus, la valeur finale de `s` ne sera pas 15, 30, 25 ou 5 mais 3!

Ce comportement apparemment étrange est dû à la nature fondamentalement parallèle du matériel numérique, comme nous le verrons dans les sections suivantes.

Parallélisme

VHDL étant un langage de description de matériel (HDL), il est parallèle par nature. Un programme VHDL est un ensemble de programmes séquentiels exécutés en parallèle. Ces

programmes séquentiels sont appelés processus:

```
P1: process
begin
  instruction1;
  instruction2;
  ...
  instructionN;
end process P1;

P2: process
begin
  ...
end process P2;
```

Les processus, tout comme le matériel qu'ils modélisent, ne s'arrêtent jamais: ce sont des boucles infinies. Après l'exécution de la dernière instruction, l'exécution continue avec la première.

Comme avec tout langage de programmation prenant en charge une forme ou une autre du parallélisme, un ordonnanceur est chargé de décider quel processus exécuter (et quand) lors d'une simulation VHDL. De plus, le langage offre des constructions spécifiques pour la communication et la synchronisation inter-processus.

La planification

Le planificateur gère une liste de tous les processus et, pour chacun d'entre eux, enregistre son état actuel qui peut être `running` , `run-able` ou `suspended` . Il y a au plus un processus en `running` : celui qui est actuellement exécuté. Tant que le processus en cours d'exécution n'exécute pas une instruction d' `wait` , il continue de s'exécuter et empêche l'exécution de tout autre processus. Le planificateur VHDL n'est pas préemptif: chaque processus est responsable de se suspendre et de laisser les autres processus s'exécuter. C'est l'un des problèmes que rencontrent fréquemment les débutants en VHDL: le processus libre.

```
  P3: process
    variable a: integer;
  begin
    a := s;
    a := 2 * a;
    r <= a;
  end process P3;
```

> Note: la variable `a` est déclarée localement alors que les signaux `s` et `r` sont déclarés ailleurs, à un niveau supérieur. Les variables VHDL sont locales au processus qui les déclare et ne peuvent pas être vues par d'autres processus. Un autre processus pourrait également déclarer une variable nommée `a` , ce ne serait pas la même variable que celle du processus `P3` .

Dès que le programmateur reprendra le processus `P3` , la simulation se bloquera, l'heure actuelle de la simulation ne progressera plus et le seul moyen de l'arrêter sera de tuer ou d'interrompre la simulation. La raison en est que `P3` n’a pas `wait` déclaration et restera donc en état d’éternité,

`running` revue ses 3 instructions. Aucun autre processus ne pourra jamais fonctionner, même s'il est `run-able` .

Même les processus contenant une instruction `wait` peuvent provoquer le même problème:

```
P4: process
  variable a: integer;
begin
  a := s;
  a := 2 * a;
  if a = 16 then
    wait on s;
  end if;
  r <= a;
end process P4;
```

> Remarque: l'opérateur d'égalité VHDL est `=` .

Si le processus `P4` est repris alors que la valeur du signal `s` est 3, il fonctionnera pour toujours car la condition a `a = 16` ne sera jamais vraie.

Supposons que notre programme VHDL ne contient pas de tels processus pathologiques. Lorsque le processus en cours exécute une instruction d' `wait` , celle-ci est immédiatement suspendue et le planificateur la met à l'état `suspended` . L'instruction d' `wait` comporte également la condition pour que le processus redevienne `run-able` . Exemple:

```
  wait on s;
```

signifie *me suspendre jusqu'à la valeur du signal `s` change* . Cette condition est enregistrée par le programmateur. Le planificateur sélectionne ensuite un autre processus parmi les `run-able` , le met en état de `running` et l'exécute. Et les mêmes répétitions jusqu'à ce que tous les `run-able` ont été exécutés et suspendu les processus.

> **Remarque importante:** lorsque plusieurs processus sont `run-able` , le standard VHDL ne spécifie pas comment le planificateur doit sélectionner celui à exécuter. Une conséquence est que, selon le simulateur, la version du simulateur, le système d'exploitation ou autre chose, deux simulations du même modèle VHDL pourraient, à un moment donné, faire des choix différents et sélectionner un processus différent à exécuter. Si ce choix avait un impact sur les résultats de la simulation, on pourrait dire que le VHDL est non déterministe. Comme le non-déterminisme est généralement indésirable, il serait de la responsabilité des programmeurs d'éviter les situations non déterministes. Heureusement, VHDL prend soin de cela et c'est là que les signaux entrent dans l'image.

Signaux et communication inter-processus

VHDL évite le non déterminisme en utilisant deux caractéristiques spécifiques:

1. **Les processus ne peuvent échanger des informations que par des signaux**

```
  signal r, s: integer;  -- Common to all processes
...
  P5: process
    variable a: integer; -- Different from variable a of process P6
  begin
    a := s + 1;
    r <= a;
    a := r + 1;
    wait on s;
  end process P5;

  P6: process
    variable a: integer; -- Different from variable a of process P5
  begin
    a := r + 1;
    s <= a;
    wait on r;
  end process P6;
```

Remarque: les commentaires VHDL s'étendent de `--` à la fin de la ligne.

2. **La valeur d'un signal VHDL ne change pas pendant l'exécution des processus**

Chaque fois qu'un signal est affecté, la valeur assignée est enregistrée par le programmateur mais la valeur actuelle du signal reste inchangée. Ceci est une autre différence majeure avec les variables qui prennent leur nouvelle valeur immédiatement après leur affectation.

Regardons une exécution du processus `P5` ci-dessus et supposons que `a=5` , `s=1` et `r=0` quand il est repris par le planificateur. Après avoir exécuté l'instruction `a := s + 1;` , la valeur de la variable `a` change et devient 2 (1 + 1). Lors de l'exécution de l'instruction suivante `r <= a;` il est la nouvelle valeur de `a` (2) qui est attribué à `r` . Mais `r` étant un signal, la valeur actuelle de `r` est toujours 0. Donc, lors de l'exécution d' `a := r + 1;` , la variable `a` prend (immédiatement) la valeur 1 (0 + 1), pas 3 (2 + 1) comme le dirait l'intuition.

Quand le signal `r` prendra-t-il vraiment sa nouvelle valeur? Lorsque le planificateur aura exécuté tous les processus exécutables, ils seront tous suspendus. Ceci est également appelé: *après un cycle* ***delta*** . Ce n'est qu'alors que le planificateur examinera toutes les valeurs affectées aux signaux et mettra à jour les valeurs des signaux. Une simulation VHDL est une alternance de phases d'exécution et de phases de mise à jour du signal. Pendant les phases d'exécution, la valeur des signaux est gelée. Symboliquement, on dit qu'entre une phase d'exécution et la phase de mise à jour du signal suivante, un *delta* de temps s'est écoulé. Ce n'est pas du temps réel. Un cycle *delta* n'a pas de durée physique.

Grâce à ce mécanisme de mise à jour du signal retardé, le VHDL est déterministe. Les processus ne peuvent communiquer qu'avec des signaux et les signaux ne changent pas pendant l'exécution des processus. Ainsi, l'ordre d'exécution des processus n'a pas d'importance: leur environnement externe (les signaux) ne change pas pendant l'exécution. Montrons ceci sur l'exemple précédent avec les processus `P5` et `P6` , où l'état initial est `P5.a=5` , `P6.a=10` , `s=17` , `r=0` et où le planificateur décide d'exécuter `P5` premier et `P6` suivant . Le tableau suivant montre la valeur des deux variables, les valeurs actuelles et suivantes des signaux après l'exécution de chaque instruction de chaque processus:

processus / instruction	P5.a	P6.a	s.current	s.next	r.current	r.next
Etat initial	5	dix	17		0	
`P5 / a := s + 1`	18	dix	17		0	
`P5 / r <= a`	18	dix	17		0	18
`P5 / a := r + 1`	1	dix	17		0	18
`P5 / wait on s`	1	dix	17		0	18
`P6 / a := r + 1`	1	1	17		0	18
`P6 / s <= a`	1	1	17	1	0	18
`P6 / wait on r`	1	1	17	1	0	18
Après la mise à jour du signal	1	1	1		18	

Avec les mêmes conditions initiales, si l'ordonnanceur décide d'exécuter `P6` premier et `P5` ensuite:

processus / instruction	P5.a	P6.a	s.current	s.next	r.current	r.next
Etat initial	5	dix	17		0	
`P6 / a := r + 1`	5	1	17		0	
`P6 / s <= a`	5	1	17	1	0	
`P6 / wait on r`	5	1	17	1	0	
`P5 / a := s + 1`	18	1	17	1	0	
`P5 / r <= a`	18	1	17	1	0	18
`P5 / a := r + 1`	1	1	17	1	0	18
`P5 / wait on s`	1	1	17	1	0	18
Après la mise à jour du signal	1	1	1		18	

Comme on peut le voir, après l'exécution de nos deux processus, le résultat est le même quel que soit l'ordre d'exécution.

Cette sémantique d'attribution de signaux contre-intuitive est la raison d'un deuxième type de problèmes que rencontrent fréquemment les débutants VHDL: l'affectation qui ne semble pas fonctionner car elle est retardée d'un cycle delta. Lors de l'exécution du processus `P5` étape par étape dans un débogueur, après que `r` ait été affecté à 18 et `a r + 1` a été attribué, on peut s'attendre à ce que la valeur de `a` soit 19 mais le débogueur dit obstinément que `r=0` et `a=1` ...

> Remarque: le même signal peut être attribué plusieurs fois au cours de la même phase d'exécution. Dans ce cas, c'est la dernière affectation qui décide de la valeur suivante du signal. Les autres affectations n'ont aucun effet, comme si elles n'avaient jamais été exécutées.

Il est temps de vérifier notre compréhension: revenez à notre tout premier exemple d'échange et essayez de comprendre pourquoi:

```
process
begin
  ---
  s <= r;
  r <= s;
  ---
end process;
```

échange en fait les signaux `r` et `s` sans avoir besoin d'un troisième signal temporaire et pourquoi:

```
process
begin
  ---
  r <= s;
  s <= r;
  ---
end process;
```

serait strictement équivalent. Essayez de comprendre pourquoi, si `s` est un signal entier et que sa valeur actuelle est 15, et que nous exécutons:

```
process
begin
  ---
  s <= 2 * s;
  s <= s - 5;
  s <= s / 5;
  print(s);
  wait on s;
  print(s);
  ---
end process;
```

les deux premières assignations de signal `s` n'ont aucun effet, pourquoi `s` est finalement assigné 3 et pourquoi les deux valeurs imprimées sont 15 et 3.

Temps physique

Afin de modéliser le matériel, il est très utile de pouvoir modéliser le temps physique nécessaire à certaines opérations. Voici un exemple de la façon dont cela peut être fait en VHDL. L'exemple modélise un compteur synchrone et il s'agit d'un code VHDL complet et autonome pouvant être compilé et simulé:

```
-- File counter.vhd
entity counter is
end entity counter;

architecture arc of counter is
  signal clk: bit; -- Type bit has two values: '0' and '1'
  signal c, nc: natural; -- Natural (non-negative) integers
begin
  P1: process
  begin
    clk <= '0';
    wait for 10 ns; -- Ten nano-seconds delay
    clk <= '1';
    wait for 10 ns; -- Ten nano-seconds delay
  end process P1;

  P2: process
  begin
    if clk = '1' and clk'event then
      c <= nc;
    end if;
    wait on clk;
  end process P2;

  P3: process
  begin
    nc <= c + 1 after 5 ns; -- Five nano-seconds delay
    wait on c;
  end process P3;
end architecture arc;
```

Dans le processus `P1` l'instruction `wait` n'est pas utilisée pour attendre que la valeur d'un signal change, comme nous l'avons vu jusqu'à présent, mais pour attendre une durée donnée. Ce processus modélise un générateur d'horloge. Le signal `clk` est l'horloge de notre système, il est périodique avec une période de 20 ns (50 MHz) et a un cycle de service.

Le processus `P2` modélise un registre qui, si un front montant de `clk` vient de se produire, assigne la valeur de son entrée `nc` à sa sortie `c` , puis attend le prochain changement de valeur de `clk` .

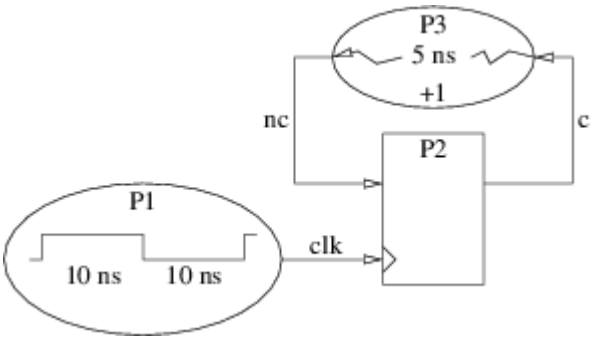
Le processus `P3` modélise un incrémenteur qui affecte la valeur de son entrée `c` , incrémentée de un, à sa sortie `nc` ... avec un délai physique de 5 ns. Il attend ensuite que la valeur de son entrée `c` change. C'est aussi nouveau. Jusqu'à présent, nous attribuions toujours des signaux avec:

```
  s <= value;
```

ce qui, pour les raisons expliquées dans les sections précédentes, peut implicitement se traduire par:

```
  s <= value; -- after delta
```

Ce petit système de matériel numérique pourrait être représenté par la figure suivante:



Avec l'introduction du temps physique, et sachant que nous avons aussi un temps symbolique mesuré en *delta* , nous avons maintenant un temps à deux dimensions que nous désignerons `T+D` où `T` est un temps physique mesuré en nano-secondes et `D` un nombre de deltas (sans durée physique).

L'image complète

Il y a un aspect important de la simulation VHDL que nous n'avons pas encore abordé: après une phase d'exécution, tous les processus sont `suspended` . Nous avons déclaré de manière informelle que le programmateur met à jour les valeurs des signaux qui ont été attribués. Mais, dans notre exemple de compteur synchrone, doit-il mettre à jour les signaux `clk` , `c` et `nc` en même temps? Qu'en est-il des délais physiques? Et ce qui arrive ensuite avec tous les processus en `suspended` état et aucun en `run-able` en `run-able` état?

L'algorithme de simulation complet (mais simplifié) est le suivant:

1. Initialisation
    - Régler l'heure actuelle `Tc` sur 0 + 0 (0 ns, 0 cycle delta)
    - Initialiser tous les signaux.
    - Exécutez chaque processus jusqu'à ce qu'il soit suspendu dans une déclaration d' `wait` .
        - Enregistrez les valeurs et les délais d'attribution des signaux.
        - Enregistrez les conditions pour que le processus reprenne (délai ou changement de signal).
    - Calculez la prochaine fois que `Tn` est la première des:
        - Le temps de reprise des processus est suspendu en `wait for <delay>` .
        - La prochaine fois qu'une valeur de signal doit changer.

2. Cycle de simulation
    - `Tc=Tn` .
    - Mettre à jour les signaux qui doivent être.
    - Mettre en état d' `run-able` tous les processus qui attendaient un changement de valeur de l'un des signaux mis à jour.
    - Mettez en état d' `run-able` tous les processus qui ont été suspendus par une instruction `wait for <delay>` et pour lesquels le temps de reprise est `Tc` .
    - Exécutez tous les processus exécutables jusqu'à ce qu'ils soient suspendus.
        - Enregistrez les valeurs et les délais d'attribution des signaux.

    - Enregistrez les conditions pour que le processus reprenne (délai ou changement de signal).
- Calculez la prochaine fois que `Tn` est la première des:
    - Le temps de reprise des processus est suspendu en `wait for <delay>` .
    - La prochaine fois qu'une valeur de signal doit changer.
- Si `Tn` est l'infini, arrêtez la simulation. Sinon, lancez un nouveau cycle de simulation.

Simulation manuelle

Pour conclure, nous allons maintenant exercer manuellement l'algorithme de simulation simplifié sur le compteur synchrone présenté ci-dessus. Nous décidons arbitrairement que, lorsque plusieurs processus sont exécutables, l'ordre sera `P3` > `P2` > `P1` . Les tableaux suivants représentent l'évolution de l'état du système lors de l'initialisation et des premiers cycles de simulation. Chaque signal a sa propre colonne dans laquelle la valeur actuelle est indiquée. Lorsqu'une affectation de signal est exécutée, la valeur programmée est ajoutée à la valeur actuelle, par exemple `a/b@T+D` si la valeur actuelle est `a` et la prochaine valeur sera `b` à l'instant `T+D` (temps physique plus cycles delta) . Les 3 dernières colonnes indiquent la condition pour reprendre les processus suspendus (nom des signaux qui doivent changer ou heure à laquelle le processus doit reprendre).

Phase d'initialisation:

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
Définir l'heure actuelle	0 + 0							
Initialiser tous les signaux	0 + 0		'0'	0	0			
`P3/nc<=c+1 after 5 ns`	0 + 0		'0'	0	0/1 @ 5 + 0			
`P3/wait on c`	0 + 0		'0'	0	0/1 @ 5 + 0			`c`
`P2/if clk='1'...`	0 + 0		'0'	0	0/1 @ 5 + 0			`c`
`P2/end if`	0 + 0		'0'	0	0/1 @ 5 + 0			`c`
`P2/wait on clk`	0 + 0		'0'	0	0/1 @ 5 + 0		`clk`	`c`
`P1/clk<='0'`	0 +		'0' / '0' @ 0 +	0	0/1 @ 5 +		`clk`	`c`

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
	0		1		0			
`P1/wait for 10 ns`	0 + 0		'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Calculer la prochaine fois	0 + 0	0 + 1	'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`

Cycle de simulation # 1

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
Définir l'heure actuelle	0 + 1		'0' / '0' @ 0 + 1	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Mise à jour des signaux	0 + 1		'0'	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Calculer la prochaine fois	0 + 1	5 + 0	'0'	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`

> Note: pendant le premier cycle de simulation, il n'ya pas de phase d'exécution car aucun de nos 3 processus n'a sa condition de reprise satisfaite. `P2` attend un changement de valeur de `clk` et il y a eu une *transaction* sur `clk` , mais comme les anciennes et les nouvelles valeurs sont identiques, ce n'est pas un *changement de* valeur.

Cycle de simulation # 2

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
Définir l'heure actuelle	5 + 0		'0'	0	0/1 @ 5 + 0	10 + 0	`clk`	`c`
Mise à jour des signaux	5 + 0		'0'	0	1	10 + 0	`clk`	`c`
Calculer la prochaine fois	5 + 0	10 + 0	'0'	0	1	10 + 0	`clk`	`c`

> Note: encore une fois, il n'y a pas de phase d'exécution. `nc` changé mais aucun processus n'attend `nc` .

Cycle de simulation # 3

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
Définir l'heure actuelle	10 + 0		'0'	0	1	10 + 0	clk	c
Mise à jour des signaux	10 + 0		'0'	0	1	10 + 0	clk	c
P1/clk<='1'	10 + 0		'0' / '1' @ 10 + 1	0	1		clk	c
P1/wait for 10 ns	10 + 0		'0' / '1' @ 10 + 1	0	1	20 + 0	clk	c
Calculer la prochaine fois	10 + 0	10 + 1	'0' / '1' @ 10 + 1	0	1	20 + 0	clk	c

Cycle de simulation # 4

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
Définir l'heure actuelle	10 + 1		'0' / '1' @ 10 + 1	0	1	20 + 0	clk	c
Mise à jour des signaux	10 + 1		'1'	0	1	20 + 0	clk	c
P2/if clk='1'...	10 + 1		'1'	0	1	20 + 0		c
P2/c<=nc	10 + 1		'1'	0/1 @ 10 + 2	1	20 + 0		c
P2/end if	10 + 1		'1'	0/1 @ 10 + 2	1	20 + 0		c
P2/wait on clk	10 + 1		'1'	0/1 @ 10 + 2	1	20 + 0	clk	c
Calculer la prochaine fois	10 + 1	10 + 2	'1'	0/1 @ 10 + 2	1	20 + 0	clk	c

Cycle de simulation # 5

Lire Démarrer avec vhdl en ligne: https://riptutorial.com/fr/vhdl/topic/3803/demarrer-avec-vhdl

Chapitre 2: Analyse de la synchronisation statique - qu'est-ce que cela signifie lorsqu'une conception échoue dans le

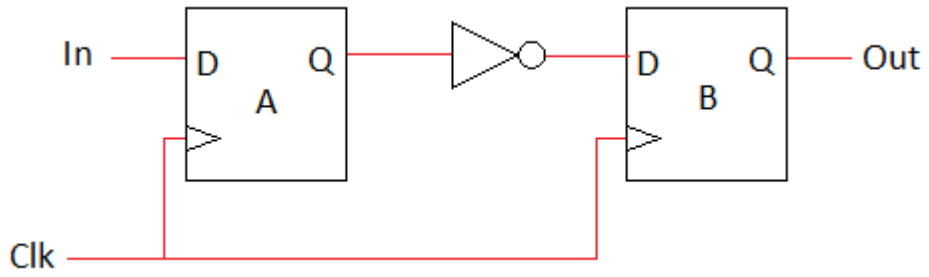
temps?

Examples

Qu'est-ce que le timing?

Le concept de timing est plus lié à la physique des tongs qu'à VHDL, mais est un concept important que tout concepteur utilisant VHDL pour créer du matériel devrait connaître.

Lors de la conception du matériel numérique, nous créons généralement une **logique synchrone** . Cela signifie que nos données voyagent du flip-flop au flip-flop, éventuellement avec une logique combinatoire entre elles. Le schéma de base de la logique synchrone intégrant une fonction combinatoire est illustré ci-dessous:



Un objectif de conception important est le **fonctionnement déterministe** . Dans ce cas, cela signifie que si la sortie Q du flop A présentait la logique `1` lorsque le front d'horloge se produisait, nous nous attendons à ce que la sortie Q du flop B commence à présenter la logique `0` chaque fois sans exception.

Avec **des** bascules **idéales** , comme décrit généralement avec VHDL (ex. `B <= not A when rising_edge(clk);`), l'opération déterministe est supposée. Les simulations VHDL comportementales supposent généralement des tongs idéales qui agissent toujours de manière déterministe. Avec des bascules réelles, ce n'est pas si simple et nous devons obéir à la **configuration** et **tenir** les exigences relatives au moment où l'entrée D d'un changement flop afin de garantir un fonctionnement fiable.

Le temps de **configuration** spécifie la durée pendant laquelle l'entrée D doit rester inchangée *avant* l'arrivée du bord de l'horloge. Le temps de **maintien** spécifie la durée pendant laquelle l'entrée D doit rester inchangée *après* l'arrivée du bord de l'horloge.

Les valeurs numériques sont basées sur la physique sous-jacente d'une bascule et varient considérablement avec le **processus** (imperfections dans le silicium à partir de la création du matériel), la **tension** (niveaux de logique «0» et «1») et la **température** . En règle générale, les valeurs utilisées pour les calculs sont les pires (exigence la plus longue), ce qui nous permet de garantir la fonctionnalité de toute puce et de tout environnement. Les puces sont fabriquées avec des plages autorisées pour l'alimentation en température afin de limiter le pire des cas à prendre en compte.

Violer les temps de configuration et de maintien peut entraîner une variété de comportements non déterministes, y compris la mauvaise valeur logique apparaissant à Q, une tension intermédiaire apparaissant à Q (peut être interprétée comme un `0` ou `1` par l'élément logique suivant), et avoir la La sortie Q oscille. Étant donné que tous les nombres utilisés sont les valeurs les plus défavorables, les violations modérées entraînent généralement un résultat déterministe normal sur un composant matériel spécifique, mais une implémentation qui présente un échec de synchronisation ne peut pas être distribuée sur plusieurs périphériques. les valeurs approchent, les valeurs les plus défavorables finiront par apparaître.

Les exigences typiques pour les bascules dans un FPGA moderne sont un temps de configuration de 60 pico-secondes, avec une exigence de maintien de 60 ps. Bien que les spécificités de l'implémentation soient données dans un contexte **FPGA** , presque tout ce matériel s'applique également à la conception **ASIC** .

Plusieurs autres délais et valeurs de temps doivent être pris en compte pour déterminer si le calendrier a été respecté. Ceux-ci inclus:

- **Délai d'acheminement** - le temps nécessaire aux signaux électriques pour parcourir les fils entre les éléments logiques
- **Délai logique** - le temps nécessaire à l'entrée de la logique combinatoire intermédiaire pour affecter la sortie. Aussi communément appelé délai de porte.
- **Clock-to-out Delay** - une autre propriété physique de la bascule, il s'agit du temps nécessaire pour que la sortie Q change après le passage de l'horloge.
- **Période de l'horloge** - le temps idéal entre deux bords de l'horloge. Une période type pour un FPGA moderne qui respecte le timing est facilement de 5 nanosecondes, mais la période réelle utilisée est choisie par le concepteur et peut être légèrement plus courte ou plus longue.
- **Clock Skew** - différence dans les retards d'acheminement de la source d'horloge vers le flop A et la source d'horloge vers le flop B
- **Gigue d'horloge / incertitude** - fonction du bruit électrique et des oscillateurs imparfaits. C'est la déviation maximale que la période d'horloge peut avoir par rapport à l'idéal, intégrant à la fois l'erreur de fréquence (par exemple, l'oscillateur tourne 1% trop rapidement, la période idéale de 5ns devenant 4,95ns avec une incertitude de 50ps) la période moyenne est de 5ns mais 1/1000 cycles a une période de 4.9ns avec 100ps de gigue)

Le fait de vérifier si la mise en œuvre d'un circuit respecte le calendrier est calculé en deux étapes avec deux ensembles de valeurs pour les retards, car les délais les plus défavorables pour l'exigence de maintien sont les meilleurs délais pour la configuration requise.

Le **contrôle de maintien** vérifie que la nouvelle valeur de la sortie Q de A sur le cycle d'horloge x n'arrive pas si tôt qu'elle perturbe la sortie Q de B au cycle d'horloge x et n'est donc pas fonction de la période d'horloge. bord de l'horloge aux deux flops. Lorsqu'un contrôle de mise en attente échoue, il est relativement facile de le réparer car la solution consiste à ajouter un délai. Les outils de mise en œuvre peuvent augmenter le délai simplement en ajoutant plus de longueur de fil dans la route.

Afin de satisfaire à l'exigence de maintien, les *délais* de sortie d'horloge, de logique et d'acheminement les *plus courts* possibles doivent être cumulativement plus longs que les besoins

de maintien lorsque l'exigence de maintien est modifiée par le décalage d'horloge.

La **vérification de la configuration** vérifie que la nouvelle valeur de la sortie Q de A au cycle d'horloge x arrive à temps pour que la sortie Q de B la considère au cycle d'horloge x + 1, et dépend donc de la période. Un échec de la vérification de la configuration nécessite la suppression du délai ou l'augmentation de l'exigence (période d'horloge). Les outils d'implémentation ne peuvent pas changer la période d'horloge (c'est le choix du concepteur), et il n'y a que peu de délais pouvant être supprimés sans modifier aucune fonctionnalité. Les outils ne sont donc pas toujours en mesure de modifier passer le contrôle d'installation.

Afin de satisfaire à l'exigence de configuration, les délais de sortie d'horloge, de logique et de routage les *plus longs* possibles doivent être cumulativement inférieurs à la période d'horloge (modifiée par le décalage d'horloge et la gigue / incertitude) moins la configuration requise.

Étant donné que la période de l'horloge (généralement fournie par des broches d'entrée d'horloge hors puce) doit être connue pour déterminer si la vérification de configuration a été effectuée, tous les outils d'implémentation nécessiteront au moins une **contrainte de synchronisation** l'horloge. La gigue / incertitude est supposée être 0 ou une petite valeur par défaut, et les autres valeurs sont toujours connues en interne par les outils du FPGA cible. Si une période d'horloge n'est pas fournie, la plupart des outils FPGA vérifieront le contrôle de mise en attente, puis trouveront l'horloge la plus rapide, même si le temps nécessaire pour optimiser les itinéraires lents est réduit. est inconnu.

Si la conception a les contraintes de période requises et si la logique non synchrone est correctement exclue de l'analyse temporelle (non abordée dans ce document), mais que la conception **échoue encore,** il existe plusieurs options:

- L'option la plus simple qui n'affecte pas du tout la fonctionnalité est d' **ajuster les directives** données à l'outil dans l'espoir d'essayer différentes stratégies d'optimisation pour produire un résultat conforme au timing. Ce n'est pas un succès fiable, mais peut souvent trouver une solution pour les cas limites.

- Le concepteur peut toujours **réduire la fréquence d'horloge** (augmenter la période) pour répondre aux vérifications de configuration, mais il a ses propres compromis fonctionnels, à savoir que votre système a réduit le débit de données proportionnel à la réduction de la vitesse d'horloge.

- Les conceptions peuvent parfois être **refaites** pour faire la même chose avec une logique plus simple, ou pour faire autre chose avec un résultat final tout aussi acceptable pour réduire les retards combinatoires, facilitant ainsi les vérifications de la configuration.

- Il est également courant de changer la conception décrite (dans le VHDL) pour les mêmes opérations logiques avec le même débit mais plus de latence en utilisant davantage de bascules et en divisant la logique combinatoire sur plusieurs cycles d'horloge. Ceci est connu sous le nom de **pipelining** et conduit à des retards combinatoires réduits (et supprime le délai de routage entre ce qui était auparavant plusieurs couches de logique combinatoire). Certaines conceptions se prêtent bien au traitement en pipeline, même si cela

peut ne pas être évident si un long trajet logique est une opération monolithique, alors que d'autres conceptions (telles que celles qui impliquent une grande quantité de **retour**) ne fonctionneront pas du tout avec la latence supplémentaire pipelining implique.

Lire Analyse de la synchronisation statique - qu'est-ce que cela signifie lorsqu'une conception échoue dans le temps? en ligne: https://riptutorial.com/fr/vhdl/topic/5936/analyse-de-la-synchronisation-statique---qu-est-ce-que-cela-signifie-lorsqu-une-conception-echoue-dans-le-temps-

Chapitre 3: Attendez

Syntaxe

- wait [on SIGNAL1 [, SIGNAL2 [...]]] [jusqu'à CONDITION] [pour TIMEOUT];
- attendez; - Eternal wait
- attendre s1, s2; - Attendez que les signaux s1 ou s2 (ou les deux) changent
- attendre que s1 = 15; - Attendez que le signal s1 change et sa nouvelle valeur est 15
- attendre que s1 = 15 pendant 10 ns; - Attendez que le signal s1 change et que sa nouvelle valeur soit 15 pendant au plus 10 ns

Examples

Éternel attendre

La forme la plus simple de déclaration d' `wait` est simplement:

```
wait;
```

Chaque fois qu'un processus l'exécute, il est suspendu pour toujours. Le planificateur de simulation ne le reprendra jamais. Exemple:

```
signal end_of_simulation: boolean := false;
...
process
begin
  clock <= '0';
  wait for 500 ps;
  clock <= '1';
  wait for 500 ps;
  if end_of_simulation then
    wait;
  end if;
end process;
```

Listes de sensibilité et déclarations d'attente

Un processus avec une liste de sensibilité ne peut pas également contenir des instructions d'attente. Cela équivaut au même processus, sans liste de sensibilité et avec une dernière déclaration:

```
wait on <sensitivity_list>;
```

Exemple:

```
  process(clock, reset)
  begin
```

```
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
end process;
```

est équivalent à:

```
process
begin
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
  wait on clock, reset;
end process;
```

VHDL2008 a introduit le mot-clé `all` dans les listes de sensibilité. C'est équivalent à *tous les signaux qui sont lus quelque part dans le processus* . Il est particulièrement utile d'éviter les listes de sensibilité incomplètes lors de la conception de processus combinatoires pour la synthèse. Exemple de liste de sensibilité incomplète:

```
process(a, b)
begin
  if ci = '0' then
    s  <= a xor b;
    co <= a and b;
  else
    s  <= a xnor b;
    co <= a or b;
  end if;
end process;
```

le signal `ci` ne fait pas partie de la liste de sensibilité et il s'agit très probablement d'une erreur de codage qui conduira à des erreurs de simulation avant et après la synthèse. Le code correct est:

```
process(a, b, ci)
begin
  if ci = '0' then
    s  <= a xor b;
    co <= a and b;
  else
    s  <= a xnor b;
    co <= a or b;
  end if;
end process;
```

Dans VHDL2008, le mot clé `all` simplifie cela et réduit le risque:

```
process(all)
begin
  if ci = '0' then
    s  <= a xor b;
```

```
      co <= a and b;
    else
      s  <= a xnor b;
      co <= a or b;
    end if;
  end process;
```

Attendez que la condition

Il est possible d'omettre les clauses `on <sensitivity_list>` et `for <timeout>` , comme dans:

```
    wait until CONDITION;
```

ce qui équivaut à:

```
    wait on LIST until CONDITION;
```

où `LIST` est la liste de tous les **signaux** qui apparaissent dans `CONDITION` . C'est aussi équivalent à:

```
    loop
      wait on LIST;
      exit when CONDITION;
    end loop;
```

Une conséquence importante est que si la `CONDITION` ne contient pas de signaux, puis:

```
    wait until CONDITION;
```

est équivalent à:

```
    wait;
```

Un exemple classique de ceci est le célèbre:

```
    wait until now = 1 sec;
```

cela ne fait pas ce que l'on pourrait penser: comme c'est `now` une fonction, pas un signal, l'exécution de cette instruction suspend le processus pour toujours.

Attendez une durée spécifique

En utilisant uniquement la clause `for <timeout>` , il est possible d'obtenir une attente inconditionnelle d'une durée spécifique. Ce n'est pas synthétisable (aucun matériel réel ne peut effectuer ce comportement aussi simplement), mais il est fréquemment utilisé pour planifier des événements et générer des horloges dans un banc de test.

Cet exemple génère une horloge cyclique à 100 MHz et 50% dans le banc de test de simulation pour piloter l'unité testée:

```
constant period : time := 10 ns;
...
process
begin
   loop
      clk <= '0';
      wait for period/2;
      clk <= '1';
      wait for period/2;
   end loop;
end process;
```

Cet exemple montre comment utiliser une durée littérale pour séquencer le processus de test / analyse du banc d'essai:

```
process
begin
   rst <= '1';
   wait for 50 ns;
   wait until rising_edge(clk); --deassert reset synchronously
   rst <= '0';
   uut_input <= test_constant;
   wait for 100 us; --allow time for the uut to process the input
   if uut_output /= expected_output_constant then
      assert false report "failed test" severity error;
   else
      assert false report "passed first stage" severity note;
      uut_process_stage_2 <= '1';
   end if;
   ...
   wait;
end process;
```

Lire Attendez en ligne: https://riptutorial.com/fr/vhdl/topic/6449/attendez

Chapitre 4: commentaires

Introduction

Tout langage de programmation décent prend en charge les commentaires. Dans VHDL, ils sont particulièrement importants car la compréhension d'un code VHDL, même modérément sophistiqué, est souvent difficile.

Examples

Commentaires sur une seule ligne

Un commentaire sur une seule ligne commence par deux tirets (`--`) et s'étend jusqu'à la fin de la ligne. Exemple :

```
-- This process models the state register
process(clock, aresetn)
begin
  if aresetn = '0' then         -- Active low, asynchronous reset
    state <= IDLE;
  elsif rising_edge(clock) then -- Synchronized on the rising edge of the clock
    state <= next_state;
  end if;
end process;
```

Commentaires délimités

À partir de VHDL 2008, un commentaire peut également s'étendre sur plusieurs lignes. Les commentaires sur plusieurs lignes commencent par `/*` et se terminent par `*/` . Exemple :

```
/* This process models the state register.
   It has an active low, asynchronous reset
   and is synchronized on the rising edge
   of the clock. */
process(clock, aresetn)
begin
  if aresetn = '0' then
    state <= IDLE;
  elsif rising_edge(clock) then
    state <= next_state;
  end if;
end process;
```

Les commentaires délimités peuvent également être utilisés sur moins d'une ligne:

```
-- Finally, we decided to skip the reset...
process(clock/*, aresetn*/)
begin
  /*if aresetn = '0' then
    state <= IDLE;
```

```
  els*/if rising_edge(clock) then
    state <= next_state;
  end if;
end process;
```

Commentaires imbriqués

Lancer un nouveau commentaire (simple ligne ou délimité) dans un commentaire (simple ligne ou délimité) n'a aucun effet et est ignoré. Exemples:

```
-- This is a single-line comment. This second -- has no special meaning.

-- This is a single-line comment. This /* has no special meaning.

/* This is not a
single-line comment.
And this -- has no
special meaning. */

/* This is not a
single-line comment.
And this second /* has no
special meaning. */
```

Lire commentaires en ligne: https://riptutorial.com/fr/vhdl/topic/9292/commentaires

Chapitre 5: Conception matérielle numérique en utilisant VHDL en un mot

Introduction

Dans cette rubrique, nous proposons une méthode simple pour concevoir correctement des circuits numériques simples avec VHDL. La méthode est basée sur des diagrammes graphiques et un principe facile à retenir:

Pensez d'abord au matériel, code VHDL ensuite

Il est destiné aux débutants en conception de matériel numérique utilisant VHDL, avec une compréhension limitée de la sémantique de synthèse du langage.

Remarques

La conception de matériel numérique utilisant VHDL est simple, même pour les débutants, mais il y a quelques points importants à connaître et un petit ensemble de règles à respecter. L'outil utilisé pour transformer une description VHDL en matériel numérique est un synthétiseur logique. La sémantique du langage VHDL utilisé par les synthétiseurs logiques est assez différente de la sémantique de simulation décrite dans le Manuel de référence du langage (LRM). Pire encore: il n'est pas standardisé et varie selon les outils de synthèse.

La méthode proposée introduit plusieurs limitations importantes par souci de simplicité:

- Aucune bascule déclenchée par le niveau.
- Les circuits sont synchrones sur le front montant d'une seule horloge.
- Aucune réinitialisation ou définition asynchrone.
- Pas de lecteur multiple sur les signaux résolus.

L'exemple de diagramme de blocs , le premier d'une série de trois, présente brièvement les bases du matériel numérique et propose une courte liste de règles pour concevoir un schéma de principe d'un circuit numérique. Les règles aident à garantir une traduction directe du code VHDL qui simule et synthétise comme prévu.

L'exemple de codage explique la traduction d'un diagramme en code VHDL et l'illustre sur un circuit numérique simple.

Enfin, l'exemple du concours de design de John Cooley montre comment appliquer la méthode proposée à un exemple plus complexe de circuit numérique. Il précise également les limitations introduites et en assouplit certains.

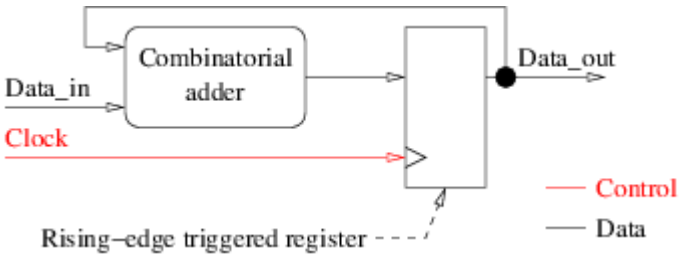
Examples

Diagramme

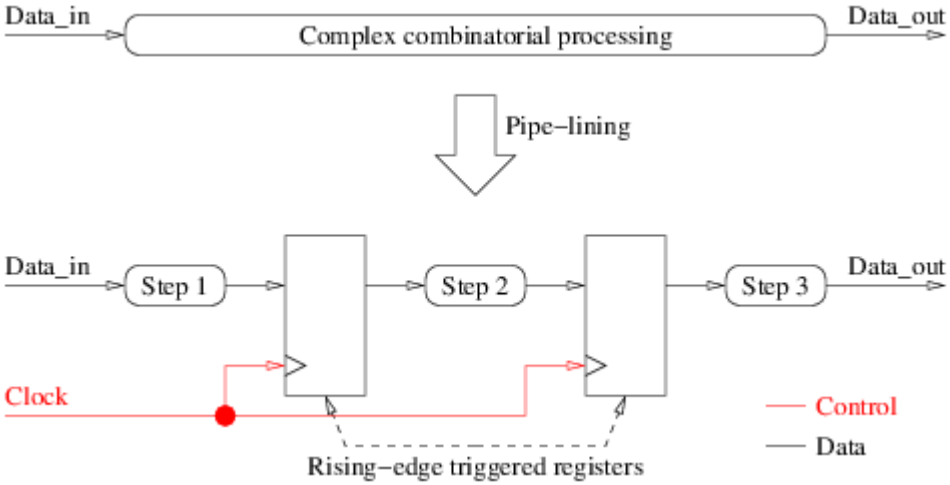
Le matériel numérique est construit à partir de deux types de primitives matérielles:

- Portes combinatoires (inverseurs et / ou xor, adders complets 1 bit, multiplexeurs 1 bit ...) Ces portes logiques effectuent un calcul booléen simple sur leurs entrées et produisent une sortie. Chaque fois qu'une de leurs entrées change, elles commencent à propager des signaux électriques et, après un court délai, la sortie se stabilise à la valeur résultante. Le délai de propagation est important car il est fortement lié à la vitesse à laquelle le circuit numérique peut fonctionner, c'est-à-dire sa fréquence d'horloge maximale.
- Eléments de mémoire (loquets, D-Flip-Flops, RAMs ...). Contrairement aux portes logiques combinatoires, les éléments de mémoire ne réagissent pas immédiatement au changement de leurs entrées. Ils ont des entrées de données, des entrées de contrôle et des sorties de données. Ils réagissent sur une combinaison particulière d'entrées de contrôle, et non sur un changement de leurs entrées de données. La bascule D déclenchée par un front montant, par exemple, possède une entrée d'horloge et une entrée de données. Sur chaque front montant de l'horloge, les données sont échantillonnées et copiées sur la sortie de données qui reste stable jusqu'au prochain front montant de l'horloge, même si l'entrée de données change entre les deux.

Un circuit matériel numérique est une combinaison de logique combinatoire et d'éléments de mémoire. Les éléments de mémoire ont plusieurs rôles. L'une d'elles est de permettre de réutiliser la même logique combinatoire pour plusieurs opérations consécutives sur des données différentes. Les circuits utilisant ceci sont fréquemment appelés *circuits séquentiels* . La figure ci-dessous montre un exemple de circuit séquentiel qui accumule des valeurs entières en utilisant le même additionneur combinatoire, grâce à un registre déclenché par front montant. C'est aussi notre premier exemple de diagramme.



La tuyauterie est une autre utilisation courante des éléments de mémoire et la base de nombreuses architectures de microprocesseurs. Il vise à augmenter la fréquence d'horloge d'un circuit en divisant un traitement complexe en une succession d'opérations plus simples et en parallélisant l'exécution de plusieurs traitements consécutifs:



Le diagramme est une représentation graphique du circuit numérique. Cela aide à prendre les bonnes décisions et à bien comprendre la structure globale avant de procéder au codage. C'est l'équivalent des phases d'analyse préliminaire recommandées dans de nombreuses méthodes de conception de logiciels. Les concepteurs expérimentés sautent fréquemment cette phase de conception, du moins pour les circuits simples. Si vous êtes un débutant dans la conception de matériel numérique, et si vous voulez coder un circuit numérique dans VHDL, adopter les 10 règles simples ci-dessous pour dessiner votre diagramme devrait vous aider à bien faire les choses:

1. Entourez votre dessin d'un grand rectangle. C'est la limite de votre circuit. Tout ce qui traverse cette limite est un port d'entrée ou de sortie. L'entité VHDL décrira cette limite.
2. Séparez clairement les registres déclenchés par la tranche (par exemple les blocs carrés) de la logique combinatoire (par exemple, les blocs ronds). En VHDL, ils seront traduits en processus mais de deux types très différents: synchrones et combinatoires.
3. N'utilisez pas de loquets déclenchés par le niveau, utilisez uniquement des registres déclenchés sur le front montant. Cette contrainte ne provient pas de VHDL, parfaitement utilisable pour modéliser les verrous. C'est juste un conseil raisonnable pour les débutants. Les verrous sont moins souvent nécessaires et leur utilisation pose de nombreux problèmes que nous devrions probablement éviter, du moins pour nos premiers modèles.
4. Utilisez la même horloge pour tous vos registres déclenchés sur le front montant. Là encore, cette contrainte est là pour simplifier. Il ne provient pas de VHDL, parfaitement utilisable pour modéliser des systèmes multi-horloges. Nommez l' `clock` . Il vient de l'extérieur et est une entrée de tous les blocs carrés et seulement d'eux. Si vous le souhaitez, ne représentez même pas l'horloge, il en va de même pour tous les blocs carrés et vous pouvez le laisser implicite dans votre diagramme.
5. Représente les communications entre les blocs avec des flèches nommées et orientées. Pour le bloc dont provient une flèche, la flèche est une sortie. Pour le bloc auquel une flèche va, la flèche est une entrée. Toutes ces flèches deviendront des ports de l'entité VHDL, si elles traversent le grand rectangle ou les signaux de l'architecture VHDL.
6. Les flèches ont une seule origine mais elles peuvent avoir plusieurs destinations. En effet, si une flèche avait plusieurs origines, nous créerions un signal VHDL avec plusieurs pilotes. Ceci n'est pas complètement impossible mais nécessite un soin particulier pour éviter les courts-circuits. Nous allons donc éviter cela pour l'instant. Si une flèche a plusieurs destinations, fourchez la flèche autant de fois que nécessaire. Utilisez des points pour

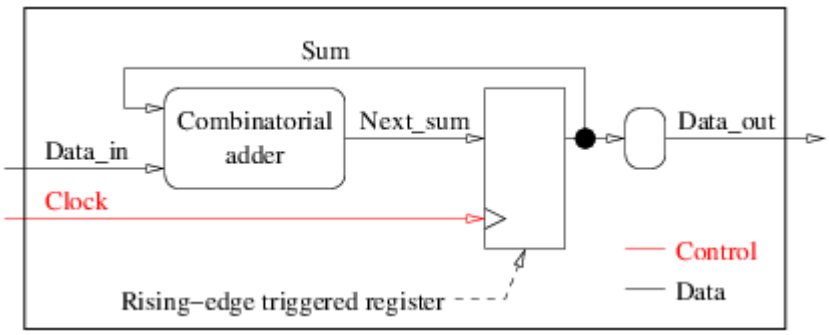
distinguer les traversées connectées et non connectées.

7. Certaines flèches proviennent de l'extérieur du grand rectangle. Ce sont les ports d'entrée de l'entité. Une flèche d'entrée ne peut également être la sortie d'aucun de vos blocs. Ceci est imposé par le langage VHDL: les ports d'entrée d'une entité peuvent être lus mais pas écrits. C'est encore pour éviter les courts-circuits.
8. Des flèches sortent. Ce sont les ports de sortie. Dans les versions VHDL antérieures à 2008, les ports de sortie d'une entité peuvent être écrits mais pas lus. Une flèche de sortie doit donc avoir une seule origine et une seule destination: l'extérieur. Pas de fourche sur les flèches de sortie, une flèche de sortie ne peut pas être aussi l'entrée d'un de vos blocs. Si vous souhaitez utiliser une flèche de sortie comme entrée pour certains de vos blocs, insérez un nouveau bloc rond pour le diviser en deux parties: la partie interne, avec autant de fourchettes que vous le souhaitez, et la flèche de sortie provenant du nouveau bloquer et va dehors. Le nouveau bloc deviendra une simple affectation continue dans VHDL. Une sorte de renommage transparent. Depuis VHDL 2008, les ports peuvent également être lus.
9. Toutes les flèches qui ne vont pas ou ne vont pas de / vers l'extérieur sont des signaux internes. Vous les déclarerez tous dans l'architecture VHDL.
10. Chaque cycle du diagramme doit comporter au moins un bloc carré. Ce n'est pas dû à VHDL. Il provient des principes de base de la conception de matériel numérique. Les boucles combinatoires doivent absolument être évitées. Sauf dans de très rares cas, ils ne produisent aucun résultat utile. Et un cycle du diagramme qui ne comprendrait que des blocs ronds serait une boucle combinatoire.

N'oubliez pas de vérifier soigneusement la dernière règle, elle est aussi essentielle que les autres mais peut être un peu plus difficile à vérifier.

À moins que vous ayez absolument besoin de fonctionnalités que nous avons exclues pour l'instant, comme les verrous, les horloges multiples ou les signaux comportant plusieurs pilotes, vous devriez facilement dessiner un schéma fonctionnel de votre circuit conforme aux 10 règles. Si ce n'est pas le cas, le problème vient probablement du circuit que vous souhaitez, pas du VHDL ou du synthétiseur logique. Et cela signifie probablement que le circuit que vous voulez n'est **pas** du matériel numérique.

L'application des 10 règles à notre exemple de circuit séquentiel conduirait à un schéma fonctionnel tel que:



1. Le grand rectangle autour du diagramme est traversé par 3 flèches, représentant les ports d'entrée et de sortie de l'entité VHDL.
2. Le diagramme a deux blocs (combinatoires) ronds - l'additionneur et le bloc de renommage de sortie - et un bloc (synchrone) carré - le registre.

3. Il utilise uniquement des registres à déclenchement par front.
4. Il n'y a qu'une seule horloge, nommée `clock` et nous utilisons uniquement son front montant.
5. Le diagramme a cinq flèches, une avec une fourchette. Ils correspondent à deux signaux internes, deux ports d'entrée et un port de sortie.
6. Toutes les flèches ont une origine et une destination sauf la flèche nommée `Sum` qui a deux destinations.
7. Les flèches `Data_in` et `Clock` sont nos deux ports d'entrée. Ils ne sont pas produits par nos propres blocs.
8. La flèche `Data_out` est notre port de sortie. Afin d'être compatible avec les versions de VHDL antérieures à 2008, nous avons ajouté un bloc de renommage supplémentaire (Round) entre `Sum` et `Data_out` . Ainsi, `Data_out` a exactement une source et une destination.
9. `Sum` et `Next_sum` sont nos deux signaux internes.
10. Il y a exactement un cycle dans le graphique et il comprend un bloc carré.

Notre diagramme est conforme aux 10 règles. L'exemple de codage détaillera comment traduire ce type de diagramme en VHDL.

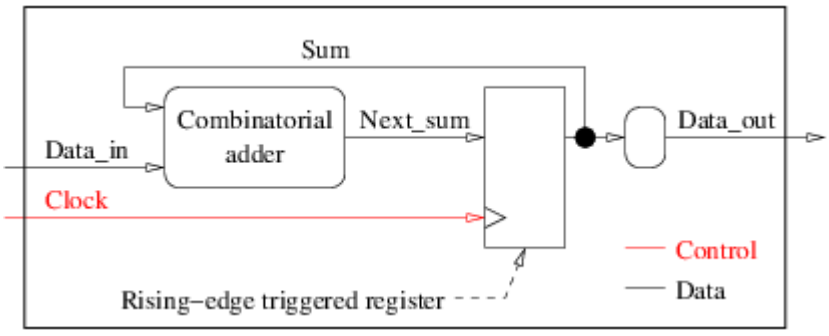
Codage

Cet exemple est le deuxième d'une série de 3. Si vous ne l'avez pas encore fait, lisez d'abord l'exemple du diagramme .

Avec un diagramme conforme aux 10 règles (voir l'exemple du diagramme), le codage VHDL devient simple:

- le grand rectangle environnant devient l'entité VHDL,
- les flèches internes deviennent des signaux VHDL et sont déclarées dans l'architecture,
- chaque bloc carré devient un processus synchrone dans le corps de l'architecture,
- chaque bloc rond devient un processus combinatoire dans le corps de l'architecture.

Illustrons ceci sur le schéma synoptique d'un circuit séquentiel:



Le modèle VHDL d'un circuit comprend deux unités de compilation:

- L'entité qui décrit le nom du circuit et son interface (noms des ports, directions et types). C'est une traduction directe du grand rectangle environnant du diagramme. En supposant que les données sont des nombres entiers et que l' `clock` utilise le `bit` type VHDL (deux valeurs uniquement: `'0'` et `'1'`), l'entité de notre circuit séquentiel pourrait être:

```
entity sequential_circuit is
  port(
    Data_in:  in  integer;
    Clock:    in  bit;
    Data_out: out integer
  );
end entity sequential_circuit;
```

- L'architecture qui décrit les composants internes du circuit (ce qu'il fait). C'est là que les signaux internes sont déclarés et que tous les processus sont instanciés. Le squelette de l'architecture de notre circuit séquentiel pourrait être:

```
architecture ten_rules of sequential_circuit is
  signal Sum, Next_sum: integer;
begin
  <...processes...>
end architecture ten_rules;
```

Nous avons trois processus à ajouter au corps de l'architecture, un synchrone (bloc carré) et deux combinatoires (blocs ronds).

Un processus synchrone ressemble à ceci:

```
process(clock)
begin
  if rising_edge(clock) then
    o1 <= i1;
    ...
    ox <= ix;
  end if;
end process;
```

où `i1, i2,..., ix` sont **toutes les** flèches qui entrent dans le bloc carré correspondant du diagramme et `o1, ..., ox` sont **toutes les** flèches qui génèrent le bloc carré correspondant du diagramme. Absolument rien ne sera changé, sauf les noms des signaux, bien sûr. Rien. Pas même un seul personnage.

Le processus synchrone de notre exemple est donc:

```
  process(clock)
  begin
    if rising_edge(clock) then
      Sum <= Next_sum;
    end if;
  end process;
```

Ce qui peut être traduit de manière informelle en: si l' `clock` change, et alors seulement, si le changement est un front montant (`'0'` à `'1'`), attribuez la valeur du signal `Next_sum` au signal `Sum` .

Un processus combinatoire ressemble à ceci:

```
process(i1, i2,... , ix)
  variable v1: <type_of_v1>;
```

```
  ...
  variable vy: <type_of_vy>;
begin
  v1 := <default_value_for_v1>;
  ...
  vy := <default_value_for_vy>;
  o1 <= <default_value_for_o1>;
  ...
  oz <= <default_value_for_oz>;
  <statements>
end process;
```

où `i1, i2,..., in` sont **toutes les** flèches qui entrent dans le bloc rond correspondant du diagramme. **tout** et pas plus. Nous n'oublierons aucune flèche et nous n'ajouterons rien à la liste.

`v1, ..., vy` sont des variables dont nous pouvons avoir besoin pour simplifier le code du processus. Ils ont exactement le même rôle que dans tout autre langage de programmation impératif: contenir des valeurs temporaires. Ils doivent absolument tous être assignés avant d'être lus. Si nous ne le garantissons pas, le processus ne sera plus combinatoire puisqu'il modélisera le type d'éléments de mémoire pour conserver la valeur de certaines variables d'une exécution de processus à l'autre. C'est la raison des instructions `vi := <default_value_for_vi>` au début du processus. Notez que `<default_value_for_vi>` doit être une constante. Sinon, si ce sont des expressions, nous pourrions utiliser accidentellement des variables dans les expressions et lire une variable avant de l'affecter.

`o1, ..., om` sont **toutes les** flèches qui génèrent le bloc rond correspondant de votre diagramme. **tout** et pas plus. Ils doivent absolument être tous affectés au moins une fois au cours de l'exécution du processus. Comme les structures de contrôle VHDL (`if` , `case` ...) peuvent très facilement empêcher l'attribution d'un signal de sortie, il est fortement conseillé d'attribuer chacune d'entre elles, sans condition, à une valeur constante `<default_value_for_oi>` au début du processus. De cette façon, même si une instruction `if` masque une affectation de signal, elle aura de toute façon reçu une valeur.

Absolument rien ne sera changé pour ce squelette VHDL, sauf les noms des variables, le cas échéant, les noms des entrées, les noms des sorties, les valeurs des constantes `<default_value_for_..>` et des `<statements>` . N'oubliez **pas** une seule attribution de valeur par défaut, si vous faites la synthèse, vous en déduirez les éléments de mémoire indésirables (les verrous les plus probables) et le résultat ne sera pas ce que vous vouliez initialement.

Dans notre exemple de circuit séquentiel, le processus additionneur combinatoire est:

```
  process(Sum, Data_in)
  begin
    Next_sum <= 0;
    Next_sum <= Sum + Data_in;
  end process;
```

Lesquels peuvent être traduits de manière informelle en: si `Sum` ou `Data_in` (ou les deux) changent, affectez la valeur 0 pour signaler `Next_sum` , puis attribuez-lui à nouveau la valeur `Sum + Data_in` .

Comme la première affectation (avec la valeur par défaut constante `0`) est immédiatement suivie

d'une autre affectation qui la remplace, nous pouvons simplifier:

```
  process(Sum, Data_in)
  begin
    Next_sum <= Sum + Data_in;
  end process;
```

Le deuxième processus combinatoire correspond au bloc que nous avons ajouté sur une flèche de sortie avec plusieurs destinations afin de se conformer aux versions VHDL antérieures à 2008. Son code est simplement:

```
  process(Sum)
  begin
    Data_out <= 0;
    Data_out <= Sum;
  end process;
```

Pour la même raison que pour l'autre processus combinatoire, nous pouvons le simplifier comme suit:

```
  process(Sum)
  begin
    Data_out <= Sum;
  end process;
```

Le code complet du circuit séquentiel est:

```
-- File sequential_circuit.vhd
entity sequential_circuit is
  port(
    Data_in:  in  integer;
    Clock:    in  bit;
    Data_out: out integer
  );
end entity sequential_circuit;

architecture ten_rules of sequential_circuit is
  signal Sum, Next_sum: integer;
begin
  process(clock)
  begin
    if rising_edge(clock) then
      Sum <= Next_sum;
    end if;
  end process;

  process(Sum, Data_in)
  begin
    Next_sum <= Sum + Data_in;
  end process;

  process(Sum)
  begin
    Data_out <= Sum;
  end process;
end architecture ten_rules;
```

Note: nous pourrions écrire les trois processus dans n'importe quel ordre, cela ne changerait rien au résultat final en simulation ou en synthèse. En effet, les trois processus sont des instructions concurrentes et VHDL les traite comme s'ils étaient réellement parallèles.

Concours de design de John Cooley

Cet exemple est directement dérivé du concours de design de John Cooley à SNUG'95 (réunion du groupe d'utilisateurs de Synopsys). Le concours était destiné à opposer les concepteurs VHDL et Verilog sur le même problème de conception. Ce que John avait en tête était probablement de déterminer quelle langue était la plus efficace. Les résultats ont été que 8 des 9 concepteurs de Verilog ont réussi à terminer le concours de design, mais aucun des 5 concepteurs de VHDL n’a pu le faire. Espérons que, en utilisant la méthode proposée, nous ferons un meilleur travail.

Caractéristiques

Notre objectif est de concevoir en VHDL synthétisable (entité et architecture) un compteur modulaire synchrone up-3, down-by-5, loadable, avec sortie carry, sortie emprunt et parité. Le compteur est un compteur non signé de 9 bits, il se situe donc entre 0 et 511. La spécification d'interface du compteur est donnée dans le tableau suivant:

... passer à l'anglais pour continuer à lire

Les opérations	Tc	Tn	clk	c	nc	P1	P2	P3
Définir l'heure actuelle	10 + 2		'1'	0/1 @ 10 + 2	1	20 + 0	clk	c
Mise à jour des signaux	10 + 2		'1'	1	1	20 + 0	clk	c
P3/nc<=c+1 after 5 ns	10 + 2		'1'	1	1/2 @ 15 + 0	20 + 0	clk	
P3/wait on c	10 + 2		'1'	1	1/2 @ 15 + 0	20 + 0	clk	c

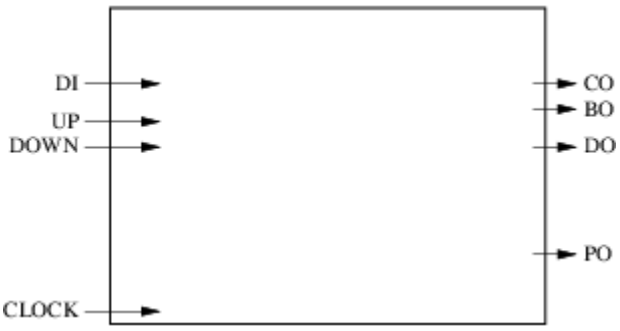
prénom	Largeur de bit	Direction	La description
L'HORLOGE	1	Contribution	Horloge principale; le compteur est synchronisé sur le front montant de l'HORLOGE
DI	9	Contribution	Bus d'entrée de données; le compteur est chargé avec DI lorsque UP et DOWN sont tous deux faibles
UP	1	Contribution	Commande de compte par 3; lorsque UP est haut et DOWN est bas, le compteur s'incrémente de 3, entourant sa valeur maximale (511)
VERS LE BAS	1	Contribution	Commande décompte par 5; lorsque DOWN est haut et que UP est bas, le compteur décroît de 5, entourant sa valeur minimale (0)
CO	1	Sortie	Exécuter le signal; élevé uniquement en comptant au-delà de la valeur maximale (511) et en faisant ainsi le tour
BO	1	Sortie	Emprunter le signal; élevé uniquement lorsque le compte à rebours est inférieur à la valeur minimale (0)
FAIRE	9	Sortie	Bus de sortie; la valeur actuelle du compteur; lorsque UP et DOWN sont tous deux élevés, le compteur conserve sa valeur
PO	1	Sortie	Signal de sortie de parité; high lorsque la valeur actuelle du compteur contient un nombre pair de 1

Lorsque vous comptez au-delà de sa valeur maximale ou lorsque vous comptez en dessous de sa valeur minimale, le compteur se déroule comme suit:

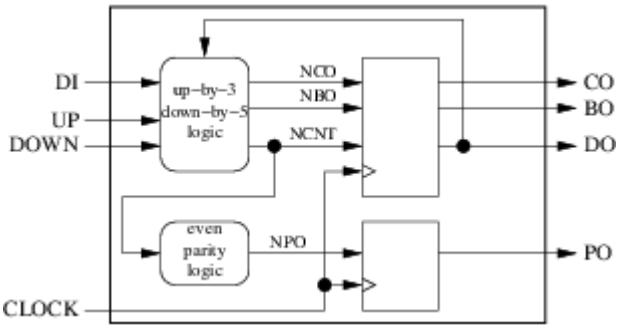
Valeur actuelle du compteur	UP DOWN	Contrer la valeur suivante	CO suivant	Suivant BO	Prochaine PO
X	00	DI	0	0	parité (DI)
X	11	X	0	0	parité (x)
0 ≤ x ≤ 508	dix	x + 3	0	0	parité (x + 3)
509	dix	0	1	0	1
510	dix	1	1	0	0
511	dix	2	1	0	0
5 ≤ x ≤ 511	01	x-5	0	0	parité (x – 5)
4	01	511	0	1	0
3	01	510	0	1	1
2	01	509	0	1	1
1	01	508	0	1	0
0	01	507	0	1	1

Diagramme

Sur la base de ces spécifications, nous pouvons commencer à concevoir un diagramme. Représentons d'abord l'interface:

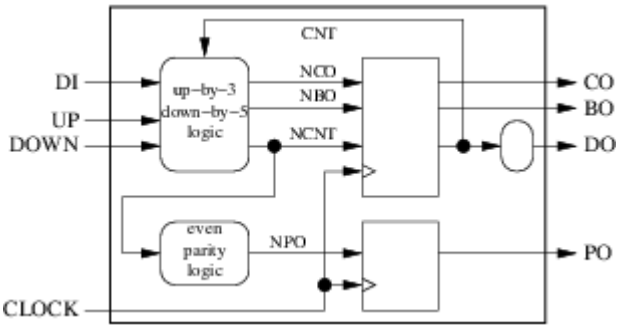


Notre circuit comporte 4 entrées (dont l'horloge) et 4 sorties. L'étape suivante consiste à décider combien de registres et de blocs combinatoires nous utiliserons et quels seront leurs rôles. Pour cet exemple simple, nous allons dédier un bloc combinatoire au calcul de la prochaine valeur du compteur, du résultat et de l'emprunt. Un autre bloc combinatoire sera utilisé pour calculer la valeur suivante de la parité. Les valeurs actuelles du compteur, de l'exécution et de l'emprunt seront stockées dans un registre tandis que la valeur actuelle de la parité sera stockée dans un registre distinct. Le résultat est indiqué sur la figure ci-dessous:



La vérification de la conformité du diagramme avec nos 10 règles de conception est rapide:

1. Notre interface externe est correctement représentée par le grand rectangle environnant.
2. Nos 2 blocs combinatoires (ronds) et nos 2 registres (carrés) sont clairement séparés.
3. Nous utilisons uniquement des registres déclenchés par un front montant.
4. Nous utilisons une seule horloge.
5. Nous avons 4 flèches internes (signaux), 4 flèches d'entrée (ports d'entrée) et 4 flèches de sortie (ports de sortie).
6. Aucune de nos flèches n'a plusieurs origines. Trois ont plusieurs destinations (`clock` , `ncnt` et `do`).
7. Aucune de nos 4 flèches d'entrée n'est une sortie de nos blocs internes.
8. Trois de nos flèches de sortie ont exactement une origine et une destination. Mais `do` a 2 destinations: l'extérieur et l' un de nos blocs combinatoires. Cela viole la règle numéro 8 et doit être corrigé en insérant un nouveau bloc combinatoire si nous voulons nous conformer aux versions VHDL antérieures à 2008:



9. Nous avons maintenant exactement 5 signaux internes (`cnt` , `nco` , `nbo` , `ncnt` et `npo`).
10. Il n'y a qu'un cycle dans le diagramme, formé par `cnt` et `ncnt` . Il y a un bloc carré dans le cycle.

Codage dans les versions VHDL antérieures à 2008

La traduction de notre diagramme dans VHDL est simple. La valeur actuelle du compteur va de 0 à 511, nous allons donc utiliser un signal `bit_vector` 9 bits pour le représenter. La seule subtilité vient de la nécessité d'effectuer des opérations binaires (comme le calcul de la parité) et des opérations arithmétiques sur les mêmes données. Le package standard `numeric_bit` de la bibliothèque `ieee` résout ce problème: il déclare un type `unsigned` avec exactement la même

déclaration que `bit_vector` et surcharge les opérateurs arithmétiques de sorte qu'ils prennent tout mélange de `unsigned` et d'entiers. Afin de calculer le résultat et l'emprunt, nous utiliserons une valeur temporaire `unsigned` 10 bits.

Les déclarations de la bibliothèque et l'entité:

```
library ieee;
use ieee.numeric_bit.all;

entity cooley is
  port(
        clock: in  bit;
        up:    in  bit;
        down:  in  bit;
        di:    in  bit_vector(8 downto 0);
        co:    out bit;
        bo:    out bit;
        po:    out bit;
        do:    out bit_vector(8 downto 0)
      );
end entity cooley;
```

Le squelette de l'architecture est:

```
architecture arc1 of cooley is
  signal cnt:  unsigned(8 downto 0);
  signal ncnt: unsigned(8 downto 0);
  signal nco:  bit;
  signal nbo:  bit;
  signal npo:  bit;
begin
    <...processes...>
end architecture arc1;
```

Chacun de nos 5 blocs est modélisé comme un processus. Les processus synchrones correspondant à nos deux registres sont très faciles à coder. Nous utilisons simplement le modèle proposé dans l'exemple de codage . Le registre qui stocke l'indicateur de parité, par exemple, est codé:

```
  poreg: process(clock)
  begin
    if rising_edge(clock) then
      po <= npo;
    end if;
  end process poreg;
```

et l'autre registre qui stocke `co` , `bo` et `cnt` :

```
  cobocntreg: process(clock)
  begin
    if rising_edge(clock) then
      co  <= nco;
      bo  <= nbo;
      cnt <= ncnt;
    end if;
```

```
  end process cobocntreg;
```

Le processus combinatoire de renommage est également très simple:

```
  rename: process(cnt)
  begin
    do <= (others => '0');
    do <= bit_vector(cnt);
  end process rename;
```

Le calcul de parité peut utiliser une variable et une boucle simple:

```
  parity: process(ncnt)
    variable tmp: bit;
  begin
    tmp := '0';
    npo <= '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    npo <= not tmp;
  end process parity;
```

Le dernier processus combinatoire est le plus complexe de tous, mais l'application stricte de la méthode de traduction proposée le rend également facile:

```
  u3d5: process(up, down, di, cnt)
    variable tmp: unsigned(9 downto 0);
  begin
    tmp  := (others => '0');
    nco  <= '0';
    nbo  <= '0';
    ncnt <= (others => '0');
    if up = '0' and down = '0' then
      ncnt <= unsigned(di);
    elsif up = '1' and down = '1' then
      ncnt <= cnt;
    elsif up = '1' and down = '0' then
      tmp   := ('0' & cnt) + 3;
      ncnt  <= tmp(8 downto 0);
      nco   <= tmp(9);
    elsif up = '0' and down = '1' then
      tmp   := ('0' & cnt) - 5;
      ncnt  <= tmp(8 downto 0);
      nbo   <= tmp(9);
    end if;
  end process u3d5;
```

Notez que les deux processus synchrones peuvent également être fusionnés et que l'un de nos processus combinatoires peut être simplifié dans une simple affectation simultanée de signaux. Le code complet, avec les déclarations de bibliothèque et de paquet, et avec les simplifications proposées est le suivant:

```
library ieee;
use ieee.numeric_bit.all;
```

```
entity cooley is
  port(
        clock: in  bit;
        up:    in  bit;
        down:  in  bit;
        di:    in  bit_vector(8 downto 0);
        co:    out bit;
        bo:    out bit;
        po:    out bit;
        do:    out bit_vector(8 downto 0)
      );
end entity cooley;

architecture arc2 of cooley is
  signal cnt:  unsigned(8 downto 0);
  signal ncnt: unsigned(8 downto 0);
  signal nco:  bit;
  signal nbo:  bit;
  signal npo:  bit;
begin
  reg: process(clock)
  begin
    if rising_edge(clock) then
      co  <= nco;
      bo  <= nbo;
      po  <= npo;
      cnt <= ncnt;
    end if;
  end process reg;

  do <= bit_vector(cnt);

  parity: process(ncnt)
    variable tmp: bit;
  begin
    tmp := '0';
    npo <= '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    npo <= not tmp;
  end process parity;

  u3d5: process(up, down, di, cnt)
    variable tmp: unsigned(9 downto 0);
  begin
    tmp  := (others => '0');
    nco  <= '0';
    nbo  <= '0';
    ncnt <= (others => '0');
    if up = '0' and down = '0' then
      ncnt <= unsigned(di);
    elsif up = '1' and down = '1' then
      ncnt <= cnt;
    elsif up = '1' and down = '0' then
      tmp   := ('0' & cnt) + 3;
      ncnt  <= tmp(8 downto 0);
      nco   <= tmp(9);
    elsif up = '0' and down = '1' then
      tmp   := ('0' & cnt) - 5;
```

```
      ncnt  <= tmp(8 downto 0);
      nbo   <= tmp(9);
    end if;
  end process u3d5;
end architecture arc2;
```

Aller un peu plus loin

La méthode proposée est simple et sûre mais repose sur plusieurs contraintes qui peuvent être assouplies.

Ignorer le dessin du diagramme

Les concepteurs expérimentés peuvent ignorer le dessin d'un diagramme pour des conceptions simples. Mais ils pensent toujours au matériel en premier. Ils dessinent dans leur tête plutôt que sur une feuille de papier mais continuent à dessiner.

Utiliser des réinitialisations asynchrones

Dans certaines circonstances, les réinitialisations (ou ensembles) asynchrones peuvent améliorer la qualité d'une conception. La méthode proposée ne prend en charge que les réinitialisations synchrones (c'est-à-dire les réinitialisations prises en compte sur les fronts montants de l'horloge):

```
  process(clock)
  begin
    if rising_edge(clock) then
      if reset = '1' then
        o <= reset_value_for_o;
      else
        o <= i;
      end if;
    end if;
  end process;
```

La version avec réinitialisation asynchrone modifie notre modèle en ajoutant le signal de réinitialisation dans la liste de sensibilité et en lui donnant la priorité la plus élevée:

```
  process(clock, reset)
  begin
    if reset = '1' then
      o <= reset_value_for_o;
    elsif rising_edge(clock) then
      o <= i;
    end if;
  end process;
```

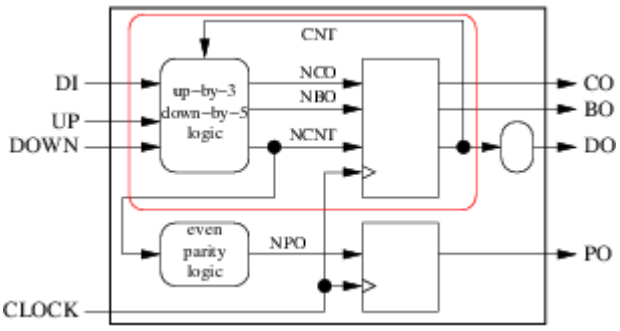
Fusionner plusieurs processus simples

Nous l'avons déjà utilisé dans la version finale de notre exemple. La fusion de plusieurs processus synchrones, s'ils ont tous la même horloge, est triviale. La fusion de plusieurs processus combinatoires en un seul est également triviale et n'est qu'une simple réorganisation du diagramme.

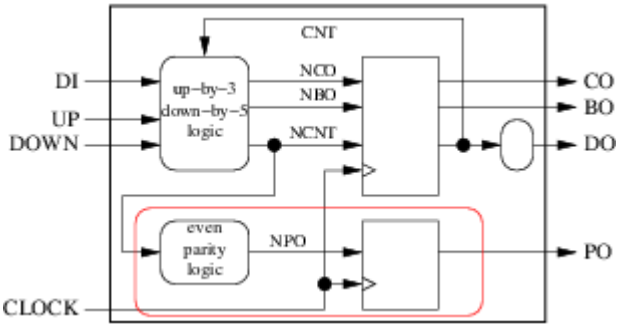
Nous pouvons également fusionner certains processus combinatoires avec des processus synchrones. Mais pour ce faire, nous devons revenir à notre diagramme et ajouter une onzième règle:

11. Groupez plusieurs blocs ronds et au moins un bloc carré en dessinant un boîtier autour d'eux. Joignez également les flèches qui peuvent être. Ne laissez pas une flèche traverser les limites de l'enceinte si elle ne vient pas ou ne va pas de / à l'extérieur de l'enceinte. Une fois cela fait, regardez toutes les flèches de sortie du boîtier. Si l'un d'entre eux provient d'un bloc rond du boîtier ou est également une entrée du boîtier, nous ne pouvons pas fusionner ces processus dans un processus synchrone. Sinon nous pouvons.

Dans notre exemple, par exemple, nous ne pouvions pas regrouper les deux processus dans l'enceinte rouge de la figure suivante:



parce que `ncnt` est une sortie du boîtier et son origine est un bloc (combinatoire) rond. Mais on pourrait grouper:



Le signal interne `npo` deviendrait inutile et le processus résultant serait:

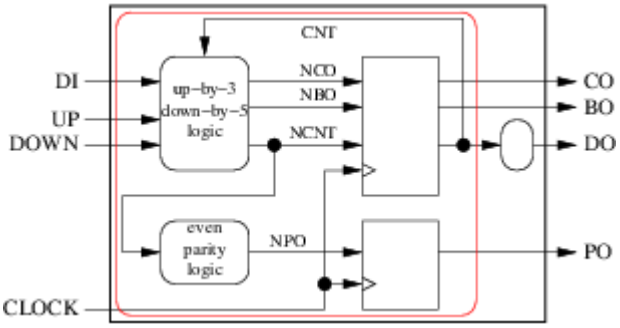
```
poreg: process(clock)
  variable tmp: bit;
begin
  if rising_edge(clock) then
    tmp := '0';
    for i in 0 to 8 loop
      tmp := tmp xor ncnt(i);
    end loop;
    po <= not tmp;
```

```
    end if;
  end process poreg;
```

qui pourrait également être fusionné avec l'autre processus synchrone:

```
  reg: process(clock)
    variable tmp: bit;
  begin
    if rising_edge(clock) then
      co  <= nco;
      bo  <= nbo;
      cnt <= ncnt;
      tmp := '0';
      for i in 0 to 8 loop
        tmp := tmp xor ncnt(i);
      end loop;
      po <= not tmp;
    end if;
  end process reg;
```

Le regroupement pourrait même être:



Menant à l'architecture beaucoup plus simple:

```
architecture arc5 of cooley is
  signal cnt: unsigned(8 downto 0);
begin
  process(clock)
    variable ncnt: unsigned(9 downto 0);
    variable tmp:  bit;
  begin
    if rising_edge(clock) then
      ncnt := '0' & cnt;
      co   <= '0';
      bo   <= '0';
      if up = '0' and down = '0' then
        ncnt := unsigned('0' & di);
      elsif up = '1' and down = '0' then
        ncnt := ncnt + 3;
        co   <= ncnt(9);
      elsif up = '0' and down = '1' then
        ncnt := ncnt - 5;
        bo   <= ncnt(9);
      end if;
      tmp := '0';
      for i in 0 to 8 loop
        tmp := tmp xor ncnt(i);
```

```
      end loop;
      po  <= not tmp;
      cnt <= ncnt(8 downto 0);
    end if;
  end process;

  do <= bit_vector(cnt);
end architecture arc5;
```

avec deux processus (l'attribution du signal simultané de `do` est un raccourci pour le processus équivalent). La solution avec un seul processus est laissée comme exercice. Attention, cela soulève des questions intéressantes et subtiles.

Aller encore plus loin

Les verrous déclenchés par le niveau, les fronts d'horloge qui tombent, les horloges multiples (et les resynchroniseurs entre les domaines d'horloge), les pilotes multiples pour le même signal, etc. ne sont pas mauvais. Ils sont parfois utiles. Mais apprendre à les utiliser et à éviter les pièges associés va bien au-delà de cette courte introduction à la conception de matériel numérique avec VHDL.

Codage en VHDL 2008

VHDL 2008 a introduit plusieurs modifications que nous pouvons utiliser pour simplifier davantage notre code. Dans cet exemple, nous pouvons bénéficier de 2 modifications:

- les ports de sortie peuvent être lus, nous n'avons plus besoin du signal `cnt` ,
- l'opérateur `xor` unaire peut être utilisé pour calculer la parité.

Le code VHDL 2008 pourrait être:

```
library ieee;
use ieee.numeric_bit.all;

entity cooley is
  port(
        clock: in  bit;
        up:    in  bit;
        down:  in  bit;
        di:    in  bit_vector(8 downto 0);
        co:    out bit;
        bo:    out bit;
        po:    out bit;
        do:    out bit_vector(8 downto 0)
      );
end entity cooley;

architecture arc6 of cooley is
begin
  process(clock)
    variable ncnt: unsigned(9 downto 0);
```

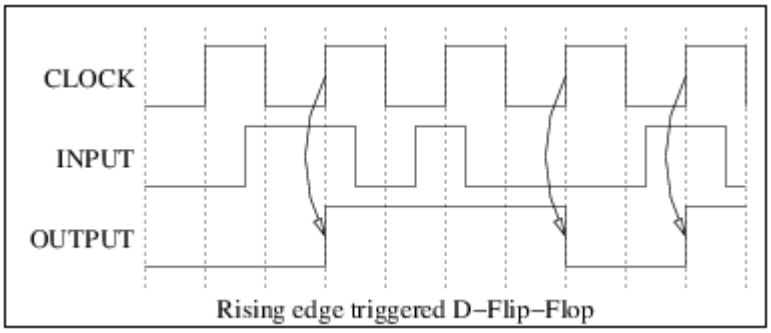
```
  begin
    if rising_edge(clock) then
      ncnt := unsigned('0' & do);
      co   <= '0';
      bo   <= '0';
      if up = '0' and down = '0' then
        ncnt := unsigned('0' & di);
      elsif up = '1' and down = '0' then
        ncnt := ncnt + 3;
        co   <= ncnt(9);
      elsif up = '0' and down = '1' then
        ncnt := ncnt - 5;
        bo   <= ncnt(9);
      end if;
      po <= not (xor ncnt(8 downto 0));
      do <= bit_vector(ncnt(8 downto 0));
    end if;
  end process;
end architecture arc6;
```

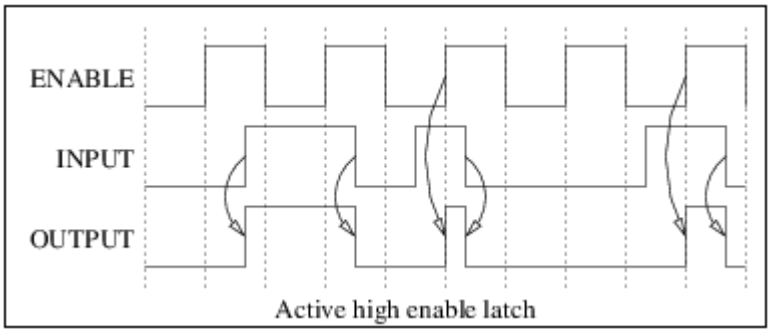
Lire Conception matérielle numérique en utilisant VHDL en un mot en ligne: https://riptutorial.com/fr/vhdl/topic/5525/conception-materielle-numerique-en-utilisant-vhdl-en-un-mot

Chapitre 6: D-Flip-Flops (DFF) et loquets

Remarques

D-Flip-Flops (DFF) et les loquets sont des éléments de mémoire. Un DFF échantillonne son entrée sur l'un ou l'autre bord de son horloge (pas les deux), tandis qu'un verrou est transparent à un niveau de son activation et mémorisé sur l'autre. La figure suivante illustre la différence:





Modéliser des DFF ou des verrous dans VHDL est facile mais il y a quelques aspects importants à prendre en compte:

- Les différences entre les modèles VHDL de DFF et les verrous.
- Comment décrire les bords d'un signal.
- Comment décrire un ensemble ou des réinitialisations synchrones ou asynchrones.

Examples

D-Flip-Flops (DFF)

Dans tous les exemples:

- `clk` est l'horloge,
- `d` est l'entrée,
- `q` est la sortie,
- `srst` est une réinitialisation active haute synchrone,
- `srstn` est une réinitialisation active synchrone basse,

- `arst` est une réinitialisation asynchrone active,
- `arstn` est une réinitialisation asynchrone active,
- `sset` est un ensemble haute synchrone actif,
- `ssetn` est un ensemble synchrone actif,
- `aset` est un ensemble haut asynchrone actif,
- `asetn` est un ensemble bas asynchrone actif

Tous les signaux sont de type `ieee.std_logic_1164.std_ulogic` . La syntaxe utilisée est celle qui permet de corriger les résultats de synthèse avec tous les synthétiseurs logiques. Veuillez vous reporter à l'exemple de *détection du bord de* l' *horloge* pour une discussion sur la syntaxe alternative.

Horloge de bord montante

```
process(clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Horloge de bord tombant

```
process(clk)
begin
  if falling_edge(clk) then
    q <= d;
  end if;
end process;
```

Horloge à front montant, réinitialisation active synchrone

```
process(clk)
begin
  if rising_edge(clk) then
    if srst = '1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

Horloge de bord montante, réinitialisation active asynchrone

```
process(clk, arst)
begin
  if arst = '1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Horloge à front descendant, asynchrone active à faible réinitialisation, synchrone active synchrone

```
process(clk, arstn)
begin
  if arstn = '0' then
    q <= '0';
  elsif falling_edge(clk) then
    if sset = '1' then
      q <= '1';
    else
      q <= d;
    end if;
  end if;
end process;
```

Horloge de bord montante, asynchrone active haute réinitialisation, bas active asynchrone

Remarque: set a une priorité plus élevée que reset

```
process(clk, arst, asetn)
begin
  if asetn = '0' then
    q <= '1';
  elsif arst = '1' then
    q <= '0';
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Loquets

Dans tous les exemples:

- `en` est le signal d'activation,
- `d` est l'entrée,
- `q` est la sortie,
- `srst` est une réinitialisation active haute synchrone,
- `srstn` est une réinitialisation active synchrone basse,
- `arst` est une réinitialisation asynchrone active,
- `arstn` est une réinitialisation asynchrone active,
- `sset` est un ensemble haute synchrone actif,
- `ssetn` est un ensemble synchrone actif,
- `aset` est un ensemble haut asynchrone actif,
- `asetn` est un ensemble bas asynchrone actif

Tous les signaux sont de type `ieee.std_logic_1164.std_ulogic` . La syntaxe utilisée est celle qui permet de corriger les résultats de synthèse avec tous les synthétiseurs logiques. Veuillez vous reporter à l'exemple de *détection du bord de* l' *horloge* pour une discussion sur la syntaxe alternative.

Active haute active

```
process(en, d)
begin
  if en = '1' then
    q <= d;
  end if;
end process;
```

Actif bas actif

```
process(en, d)
begin
  if en = '0' then
    q <= d;
  end if;
end process;
```

Activation haute active, réinitialisation active synchrone

```
process(en, d)
begin
```

```
  if en = '1' then
    if srst = '1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
end process;
```

Activation haute active, réinitialisation active asynchrone

```
process(en, d, arst)
begin
  if arst = '1' then
    q <= '0';
  elsif en = '1' then
    q <= d;
  end if;
end process;
```

Actif bas actif, asynchrone actif à faible réinitialisation, ensemble haut actif synchrone

```
process(en, d, arstn)
begin
  if arstn = '0' then
    q <= '0';
  elsif en = '0' then
    if sset = '1' then
      q <= '1';
    else
      q <= d;
    end if;
  end if;
end process;
```

Activation haute active, asynchrone active réinitialisation élevée, bas active asynchrone

Remarque: set a une priorité plus élevée que reset

```
process(en, d, arst, asetn)
```

```
begin
  if asetn = '0' then
    q <= '1';
  elsif arst = '1' then
    q <= '0';
  elsif en = '1' then
    q <= d;
  end if;
end process;
```

Détection de bord d'horloge

La petite histoire

Avec VHDL 2008 et si le type de l'horloge est `bit` , `boolean` , `ieee.std_logic_1164.std_ulogic` ou `ieee.std_logic_1164.std_logic` , une détection de bord d'horloge peut être codée pour le front montant

- `if rising_edge(clock) then`
- `if clock'event and clock = '1' then -- type bit, std_ulogic or std_logic`
- `if clock'event and clock then -- type boolean`

et pour front descendant

- `if falling_edge(clock) then`
- `if clock'event and clock = '0' then -- type bit, std_ulogic or std_logic`
- `if clock'event and not clock then -- type boolean`

Cela se comportera comme prévu, à la fois pour la simulation et la synthèse.

> Note: la définition d'un front montant sur un signal de type `std_ulogic` est un peu plus complexe que le simple `if clock'event and clock = '1' then` . `rising_edge` exemple, la fonction standard d' `rising_edge` contient une définition différente. Même si cela ne fera probablement pas de différence pour la synthèse, cela pourrait en faire une pour la simulation.

L'utilisation des fonctions standard `rising_edge` et `falling_edge` est fortement encouragée. Avec les versions précédentes de VHDL, l'utilisation de ces fonctions peut nécessiter de déclarer explicitement l'utilisation de packages standard (par exemple `ieee.numeric_bit` pour le type `bit`) ou même de les définir dans un package personnalisé.

> Remarque: n'utilisez pas les fonctions standard `rising_edge` et `falling_edge` pour détecter les contours des signaux hors horloge. Certains synthétiseurs pourraient conclure que le signal est une horloge. Astuce: la détection d'un front sur un signal hors horloge peut souvent être effectuée en échantillonnant le signal dans un registre à décalage et en comparant les valeurs échantillonnées à différents stades du registre à décalage.

La longue histoire

Décrire correctement la détection des bords d'un signal d'horloge est essentiel lors de la modélisation de D-Flip-Flops (DFF). Une arête est, par définition, une transition d'une valeur particulière à une autre. Par exemple, on peut définir le front montant d'un signal de type `bit` (le type énuméré VHDL standard qui prend deux valeurs: `'0'` et `'1'`) comme transition de `'0'` à `'1'` . Pour le type `boolean` nous pouvons le définir comme une transition de `false` à `true` .

Souvent, des types plus complexes sont utilisés. Le type `ieee.std_logic_1164.std_ulogic` , par exemple, est également un type énuméré, tout comme `bit` ou `boolean` , mais il a 9 valeurs au lieu de 2:

Valeur	Sens
`'U'`	Non initialisé
`'X'`	Forçage inconnu
`'0'`	Forcer le bas niveau
`'1'`	Forcer haut niveau
`'Z'`	Haute impédance
`'W'`	Faible inconnu
`'L'`	Faible niveau bas
`'H'`	Faible niveau élevé
`'-'`	Ne t'en fais pas

Définir un front montant sur un tel type est un peu plus complexe que pour `bit` ou `boolean` . Nous pouvons, par exemple, décider que c'est une transition de `'0'` à `'1'` . Mais on peut aussi décider que c'est une transition de `'0'` ou `'L'` à `'1'` ou `'H'` .

> Note: c'est cette seconde définition que le standard utilise pour la fonction `rising_edge(signal s: std_ulogic)` définie dans `ieee.std_logic_1164` .

Lorsque vous discutez des différentes manières de détecter les contours, il est donc important de prendre en compte le type de signal. Il est également important de prendre en compte l'objectif de modélisation: simulation uniquement ou synthèse logique? Illustrons ceci sur quelques exemples:

Bord montant DFF avec bit de type

```
signal clock, d, q: bit;
...
```

```
P1: process(clock)
begin
  if clock = '1' then
    q <= d;
  end if;
end process P1;
```

Techniquement, du point de vue de la sémantique de simulation, le processus `P1` modélise un DFF déclenché par un front montant. En effet, l'attribution `q <= d` est exécutée si et seulement si:

- `clock` changé (c'est ce que la liste de sensibilité exprime) **et**
- la valeur actuelle de l' `clock` est `'1'` .

Comme l' `clock` est de type bit et que le type bit n'a que des valeurs `'0'` et `'1'` , c'est exactement ce que nous avons défini comme un front montant d'un signal de type bit. Tout simulateur traitera ce modèle comme prévu.

> Note: Pour les synthétiseurs logiques, les choses sont un peu plus complexes, comme nous le verrons plus tard.

Front montant DFF avec réinitialisation active asynchrone et bit de type

Pour ajouter une réinitialisation active asynchrone à notre DFF, on pourrait essayer quelque chose comme:

```
signal clock, reset, d, q: bit;
...
P2_BOGUS: process(clock, reset)
begin
  if reset = '1' then
    q <= '0';
  elsif clock = '1' then
    q <= d;
  end if;
end process P2_BOGUS;
```

Mais **cela ne fonctionne pas** . La condition pour que l'attribution `q <= d` à exécuter soit: *un front montant de l'* `clock` *tandis que* `reset = '0'` . Mais ce que nous avons modélisé est:

- `clock` ou `reset` ou à la fois changé **et**
- `reset = '0'` **et**
- `clock = '1'`

Ce qui n'est pas la même chose: si la `reset` passe de `'1'` à `'0'` alors que `clock = '1'` l'assignation sera exécutée alors qu'il **ne** s'agit **pas d'** un front montant de l' `clock` .

En fait, il n'y a aucun moyen de modéliser cela dans VHDL sans l'aide d'un attribut de signal:

```
P2_OK: process(clock, reset)
begin
```

```
  if reset = '1' then
    q <= '0';
  elsif clock = '1' and clock'event then
    q <= d;
  end if;
end process P2_OK;
```

Le `clock'event` est l'attribut de signal `event` appliqué pour signaler l' `clock` . Il est évalué comme un `boolean` et il est `true` si et seulement si l' `clock` signal a changé pendant la phase de mise à jour du signal qui a juste précédé la phase d'exécution en cours. Grâce à cela, le processus `P2_OK` modélise parfaitement ce que nous voulons en simulation (et en synthèse).

Sémantique de synthèse

De nombreux synthétiseurs logiques identifient les détections de fronts de signal basées sur des schémas syntaxiques et non sur la sémantique du modèle VHDL. En d'autres termes, ils considèrent ce à quoi ressemble le code VHDL, et non son comportement. L'un des modèles qu'ils reconnaissent tous est:

```
if clock = '1' and clock'event then
```

Donc, même dans l'exemple du processus `P1` nous devrions l'utiliser si nous voulons que notre modèle soit synthétisable par tous les synthétiseurs logiques:

```
signal clock, d, q: bit;
...
P1_OK: process(clock)
begin
  if clock = '1' and clock'event then
    q <= d;
  end if;
end process P1_OK;
```

La partie `and clock'event` partie `and clock'event` de la condition sont complètement redondantes avec la liste de sensibilité mais comme certains synthétiseurs en ont besoin ...

DFF de front montant avec réinitialisation active asynchrone et type `std_ulogic`

Dans ce cas, l'expression du front montant de l'horloge et la condition de réinitialisation peuvent devenir compliquées. Si nous retenons la définition d'un front que nous avons proposée ci-dessus et si nous considérons que la réinitialisation est active si elle est `'1'` ou `'H'` , le modèle devient:

```
library ieee;
use ieee.std_logic_1164.all;
...
signal clock, reset, d, q: std_ulogic;
...
P4: process(clock, reset)
```

```
begin
  if reset = '1' or reset = 'H' then
    q <= '0';
  elsif clock'event and
        (clock'last_value = '0' or clock'last_value = 'L') and
        (clock = '1' or clock = 'H') then
    q <= d;
  end if;
end process P4;
```

> Remarque: `'last_value` est un autre attribut de signal qui renvoie la valeur que le signal avait avant la dernière modification de valeur.

Fonctions d'aide

La norme VHDL 2008 offre plusieurs fonctions d'assistance pour simplifier la détection des `std_ulogic` de signal, en particulier avec les types énumérés à plusieurs valeurs tels que `std_ulogic` . Le package `std.standard` définit les fonctions `rising_edge` et `falling_edge` sur les types `bit` et `boolean` et le package `ieee.std_logic_1164` les définit sur les types `std_ulogic` et `std_logic` .

> Remarque: avec les versions précédentes de VHDL, l'utilisation de ces fonctions peut nécessiter de déclarer explicitement l'utilisation de packages standard (par exemple, ieee.numeric_bit pour le type bit) ou même de les définir dans un package utilisateur.

Revenons aux exemples précédents et utilisons les fonctions d'aide:

```
signal clock, d, q: bit;
...
P1_OK_NEW: process(clock)
begin
  if rising_edge(clock) then
    q <= d;
  end if;
end process P1_OK_NEW;
```

```
signal clock, d, q: bit;
...
P2_OK_NEW: process(clock, reset)
begin
  if reset = '1' then
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
end process P2_OK_NEW;
```

```
library ieee;
use ieee.std_logic_1164.all;
...
signal clock, reset, d, q: std_ulogic;
...
P4_NEW: process(clock, reset)
begin
  if reset = '1' then
```

```
    q <= '0';
  elsif rising_edge(clock) then
    q <= d;
  end if;
end process P4_NEW;
```

> Remarque: dans ce dernier exemple, nous avons également simplifié le test lors de la réinitialisation. Flottant, haute impédance, les réinitialisations sont assez rares et, dans la plupart des cas, cette version simplifiée fonctionne pour la simulation et la synthèse.

Lire D-Flip-Flops (DFF) et loquets en ligne: https://riptutorial.com/fr/vhdl/topic/5983/d-flip-flops--dff--et-loquets

Chapitre 7: Fonctions de résolution, types non résolus et résolus

Introduction

Les types VHDL peuvent être *non résolus* ou *résolus* . Le type de `bit` déclaré par le package `std.standard` , par exemple, n'est pas résolu tant que le type `std_logic` déclaré par le package `ieee.std_logic_1164` est résolu.

Un signal dont le type n'est pas résolu ne peut pas être piloté (assigné) par plus d'un processus VHDL alors qu'un signal dont le type est résolu peut le faire.

Remarques

L'utilisation de types résolus devrait être réservée aux situations où l'intention est réellement de modéliser un fil matériel (ou un ensemble de fils) piloté par plus d'un circuit matériel. Le bus de données bidirectionnel d'une mémoire en est un exemple typique: lorsque la mémoire est écrite, c'est le périphérique d'écriture qui pilote le bus, tandis que lorsque la mémoire est lue, c'est la mémoire qui pilote le bus.

L'utilisation de types résolus dans d'autres situations, bien qu'une pratique fréquemment rencontrée, est une mauvaise idée, car elle supprime les erreurs de compilation très utiles lorsque des situations de lecteurs multiples indésirables sont créées accidentellement.

Le `ieee.numeric_std` package déclare la `signed` et `unsigned` types de vecteurs et les opérateurs arithmétiques surchargent sur eux. Ces types sont fréquemment utilisés lorsque des opérations arithmétiques et bit par bit sont nécessaires sur les mêmes données. Les types `signed` et `unsigned` sont résolus. Avant VHDL2008, l'utilisation de `ieee.numeric_std` et de ses types impliquait que des situations de lecteurs multiples accidentelles ne `ieee.numeric_std` pas d'erreurs de compilation. VHDL2008 ajoute de nouvelles déclarations de type à `ieee.numeric_std` : `unresolved_signed` et `unresolved_unsigned` (alias `u_signed` et `u_unsigned`). Ces nouveaux types doivent être préférés dans tous les cas où plusieurs situations de lecteur ne sont pas souhaitées.

Examples

Deux processus pilotant le même signal de type `bit`

Le modèle VHDL suivant pilote le signal `s` de deux processus différents. Comme le type de `s` est `bit` , un type non résolu, cela n'est pas autorisé.

```
-- File md.vhd
entity md is
end entity md;
```

```
architecture arc of md is

  signal s: bit;

begin

  p1: process
  begin
    s <= '0';
    wait;
  end process p1;

  p2: process
  begin
    s <= '0';
    wait;
  end process p2;

end architecture arc;
```

Compiler, élaborer et simuler, par exemple avec GHDL, déclencher une erreur:

```
ghdl -a md.vhd
ghdl -e md
./md
for signal: .md(arc).s
./md:error: several sources for unresolved signal
./md:error: error during elaboration
```

Notez que l'erreur est augmentée même si, comme dans notre exemple, tous les pilotes sont d'accord sur la valeur de conduite.

Fonctions de résolution

Un signal dont le type est résolu a une *fonction de résolution* associée. Il peut être piloté par plusieurs processus VHDL. La fonction de résolution est appelée pour calculer la valeur résultante lorsqu'un pilote attribue une nouvelle valeur.

Une fonction de résolution est une fonction pure qui prend un paramètre et retourne une valeur du type à résoudre. Le paramètre est un tableau unidimensionnel non contraint d'éléments du type à résoudre. Pour le `bit` type, par exemple, le paramètre peut être de type `bit_vector` . Au cours de la simulation, la fonction de résolution est appelée au besoin pour calculer la valeur résultante à appliquer à un signal à commandes multiples. Un tableau de toutes les valeurs est transmis à toutes les sources et renvoie la valeur résultante.

Le code suivant montre la déclaration d'une fonction de résolution pour le type `bit` qui se comporte comme un câble `and` . Il montre également comment déclarer un sous-type résolu de type `bit` et comment il peut être utilisé.

```
-- File md.vhd
entity md is
end entity md;
```

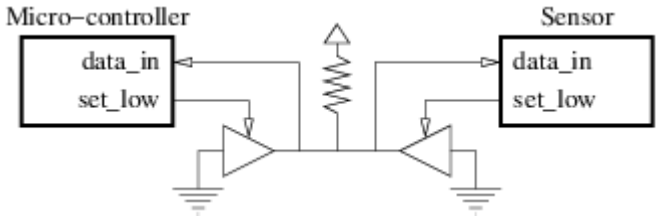
```
architecture arc of md is

  function and_resolve_bit(d: bit_vector) return bit is
    variable r: bit := '1';
  begin
    for i in d'range loop
      if d(i) = '0' then
        r := '0';
      end if;
    end loop;
    return r;
  end function and_resolve_bit;

  subtype res_bit is and_resolve_bit bit;

  signal s: res_bit;

begin

  p1: process
  begin
    s <= '0', '1' after 1 ns, '0' after 2 ns, '1' after 3 ns;
    wait;
  end process p1;

  p2: process
  begin
    s <= '0', '1' after 2 ns;
    wait;
  end process p2;

  p3: process(s)
  begin
    report bit'image(s); -- show value changes
  end process p3;

end architecture arc;
```

Compiler, élaborer et simuler, par exemple avec GHDL, ne génère pas d'erreur:

```
ghdl -a md.vhd
ghdl -e md
./md
md.vhd:39:5:@0ms:(report note): '0'
md.vhd:39:5:@3ns:(report note): '1'
```

Un protocole de communication un bit

Certains dispositifs matériels très simples et peu coûteux, tels que des capteurs, utilisent un protocole de communication à un bit. Une seule ligne de données bidirectionnelle connecte l'appareil à une sorte de microcontrôleur. Il est fréquemment tiré par une résistance de traction. Les appareils en communication pilotent la ligne basse pendant une durée prédéfinie pour envoyer une information à l'autre. La figure ci-dessous illustre ceci:



Cet exemple montre comment modéliser ceci en utilisant le type résolu
`ieee.std_logic_1164.std_logic` .

```
-- File md.vhd
library ieee;
use ieee.std_logic_1164.all;

entity one_bit_protocol is
end entity one_bit_protocol;

architecture arc of one_bit_protocol is

  component uc is
    port(
      data_in: in  std_ulogic;
      set_low: out std_ulogic
    );
  end component uc;

  component sensor is
    port(
      data_in: in  std_ulogic;
      set_low: out std_ulogic
    );
  end component sensor;

  signal data:           std_logic;  -- The bi-directional data line
  signal set_low_uc:     std_ulogic;
  signal set_low_sensor: std_ulogic;

begin

  -- Micro-controller
  uc0: uc port map(
    data_in => data,
    set_low => set_low_uc
  );

  -- Sensor
  sensor0: sensor port map(
    data_in => data,
    set_low => set_low_sensor
  );

  data <= 'H'; -- Pull-up resistor

  -- Micro-controller 3-states buffer
  data <= '0' when set_low_uc = '1' else 'Z';

  -- Sensor 3-states buffer
  data <= '0' when set_low_sensor = '1' else 'Z';

end architecture arc;
```

Lire Fonctions de résolution, types non résolus et résolus en ligne:
https://riptutorial.com/fr/vhdl/topic/9534/fonctions-de-resolution--types-non-resolus-et-resolus

Chapitre 8: Identifiants

Examples

Identifiants de base

Les identifiants de base sont composés de lettres, de traits de soulignement et de chiffres et doivent commencer par une lettre. Ils ne sont pas sensibles à la casse. Les mots réservés de la langue ne peuvent pas être des identificateurs de base. Exemples d'identificateurs de base VHDL valides:

```
A_myId90
a_MYID90
abcDEf100_1
ABCdef100_1
```

Les deux premiers sont équivalents et les deux derniers sont également équivalents (insensibilité à la casse).

Exemples d'identificateurs de base non valides:

```
_not_reset   -- start with underscore
85MHz_clock  -- start with digit
LooP         -- reserved word of the language
```

Identifiants étendus

Les identificateurs étendus VHDL sont délimités par des barres obliques inversées (`\`) et peuvent contenir des lettres, des caractères de soulignement, des chiffres, des espaces et d'autres caractères spéciaux (voir le Manuel de référence linguistique pour une définition complète des caractères spéciaux). La séquence de caractères entre les barres obliques inverses peut être réservée aux mots du langage VHDL. Les barres obliques inverses peuvent être incluses dans les identificateurs étendus en les doublant (`\\`). Les identificateurs étendus sont sensibles à la casse. Exemples d'identificateurs étendus (tous différents):

```
\if\
\If\
\My Identifier\
\An \\ Identifier \\ With \\ Backslashes\
\&#@[]:.*\
\$£$%{}\
```

Lire Identifiants en ligne: https://riptutorial.com/fr/vhdl/topic/9540/identifiants

Chapitre 9: Littéraux

Introduction

Cela a pour but de spécifier des constantes, appelées littérales dans VHDL

Examples

Littéraux numériques

```
16#A8# -- hex
2#100# -- binary
2#1000_1001_1111_0000 -- long number, adding (optional) _ (one or more) for readability
1234 -- decimal
```

Littéral énuméré

```
type state_t is (START, READING, WRITING); -- user-defined enumerated type
```

Lire Littéraux en ligne: https://riptutorial.com/fr/vhdl/topic/9344/litteraux

Chapitre 10: Récursivité

Introduction

La récursivité est une méthode de programmation où les sous-programmes s'appellent eux-mêmes. Il est très pratique de résoudre certains types de problèmes de manière élégante et générique. VHDL prend en charge la récursivité. La plupart des synthétiseurs logiques le prennent également en charge. Dans certains cas, le matériel inféré est encore meilleur (plus rapide, même taille) que la description équivalente en boucle.

Examples

Calcul du poids de Hamming d'un vecteur

```
-- loop-based version
function hw_loop(v: std_logic_vector) return natural is
  variable h: natural;
begin
  h := 0;
  for i in v'range loop
    if v(i) = '1' then
      h := h + 1;
    end if;
  end loop;
  return h;
end function hw_loop;

-- recursive version
function hw_tree(v: std_logic_vector) return natural is
  constant size: natural := v'length;
  constant vv: std_logic_vector(size - 1 downto 0) := v;
  variable h: natural;
begin
  h := 0;
  if size = 1 and vv(0) = '1' then
    h := 1;
  elsif size > 1 then
    h := hw_tree(vv(size - 1 downto size / 2)) + hw_tree(vv(size / 2 - 1 downto 0));
  end if;
  return h;
end function hw_tree;
```

Lire Récursivité en ligne: https://riptutorial.com/fr/vhdl/topic/10775/recursivite

Chapitre 11: Souvenirs

Introduction

Cela couvre les mémoires à port unique et à double port.

Syntaxe

- Type de mémoire pour largeur et profondeur constantes.

```
type MEMORY_TYPE is array (0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0);
```

Type de mémoire pour profondeur variable et largeur constante.

```
type MEMORY_TYPE is array (natural range <>) of std_logic_vector(WIDTH-1 downto 0);
```

Examples

Registre à décalage

Un registre à décalage de longueur générique. Avec entrée série et sortie série.

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

entity SHIFT_REG is
    generic(
        LENGTH: natural := 8
    );
    port(
        SHIFT_EN : in  std_logic;
        SO       : out std_logic;
        SI       : in  std_logic;
        clk      : in  std_logic;
        rst      : in  std_logic
    );
end entity SHIFT_REG;

architecture Behavioral of SHIFT_REG is
    signal reg : std_logic_vector(LENGTH-1 downto 0) := (others => '0');
begin
    main_process : process(clk) is
    begin
        if rising_edge(clk) then
            if rst = '1' then
                reg <= (others => '0');
            else
                if SHIFT_EN = '1' then
                    --Shift
```

```
                    reg <= reg(LENGTH-2 downto 0) & SI;
                else
                    reg <= reg;
                end if;
            end if;
        end if;
    end process main_process;

    SO <= reg(LENGTH-1);
end architecture Behavioral;
```

Pour Parallel out,

```
--In port
DOUT: out std_logic_vector(LENGTH-1 downto 0);
----------------------------------------------
--In architecture
DOUT <= REG;
```

Registre à décalage avec contrôle de direction, charge parallèle, sortie parallèle. (Utilisation de la variable au lieu du signal)

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

entity SHIFT_REG_UNIVERSAL is
    generic(
        LENGTH : integer := 8
    );
    port(
        DIN  : in  std_logic_vector(LENGTH - 1 downto 0);
        DOUT : out std_logic_vector(LENGTH - 1 downto 0);
        MODE : in  std_logic_vector(1 downto 0);
        SI   : in  std_logic;
        clk  : in  std_logic;
        rst  : in  std_logic
    );
end entity SHIFT_REG_UNIVERSAL;

architecture RTL of SHIFT_REG_UNIVERSAL is
begin
    main : process(clk, rst) is
        variable reg : std_logic_vector(LENGTH - 1 downto 0) := (others => '0');
    begin
        if rst = '1' then
            reg := (others => '0');
        elsif rising_edge(clk) then
            case MODE is
                when "00" =>
                    -- Hold Value
                    reg := reg;
                when "01" =>
                    -- Shift Right
                    reg := SI & reg(LENGTH - 1 downto 1);
                when "10" =>
                    -- Shift Left
                    reg := reg(LENGTH - 2 downto 0) & SI;
```

```
                when "11" =>
                    -- Parallel Load
                    reg := DIN;
                when others =>
                    null;
            end case;
        end if;
        DOUT <= reg;
    end process main;

end architecture RTL;
```

ROM

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

entity ROM is
    port(
        address : in  std_logic_vector(3 downto 0);
        dout    : out std_logic_vector(3 downto 0)
    );
end entity ROM;

architecture RTL of ROM is
    type MEMORY_16_4 is array (0 to 15) of std_logic_vector(3 downto 0);
    constant ROM_16_4 : MEMORY_16_4 := (
        x"0",
        x"1",
        x"2",
        x"3",
        x"4",
        x"5",
        x"6",
        x"7",
        x"8",
        x"9",
        x"a",
        x"b",
        x"c",
        x"d",
        x"e",
        x"f"
    );
begin
    main : process(address)
    begin
        dout <= ROM_16_4(to_integer(unsigned(address)));
    end process main;

end architecture RTL;
```

LIFO

Mémoire Last In First Out (Stack)

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

entity LIFO is
    generic(
        WIDTH : natural := 8;
        DEPTH : natural := 128
    );
    port(
        I_DATA  : in  std_logic_vector(WIDTH - 1 downto 0); --Input Data Line
        O_DATA  : out std_logic_vector(WIDTH - 1 downto 0); --Output Data Line
        I_RD_WR : in  std_logic; --Input RD/~WR signal. 1 for READ, 0 for Write
        O_FULL  : out std_logic; --Output Full signal. 1 when memory is full.
        O_EMPTY : out std_logic; --Output Empty signal. 1 when memory is empty.
        clk     : in  std_logic;
        rst     : in  std_logic
    );
end entity LIFO;

architecture RTL of LIFO is
    -- Helper Function to convert Boolean to Std_logic
    function to_std_logic(B : boolean) return std_logic is
    begin
        if B = false then
            return '0';
        else
            return '1';
        end if;
    end function to_std_logic;

    type memory_type is array (0 to DEPTH - 1) of std_logic_vector(WIDTH - 1 downto 0);
    signal memory : memory_type;
begin
    main : process(clk, rst) is
        variable stack_pointer : integer range 0 to DEPTH := 0;
        variable EMPTY, FULL   : boolean                  := false;
    begin
        --Async Reset
        if rst = '1' then
            memory   <= (others => (others => '0'));
            EMPTY := true;
            FULL  := false;

            stack_pointer := 0;
        elsif rising_edge(clk) then
            if I_RD_WR = '1' then
                -- READ
                if not EMPTY then
                    O_DATA        <= memory(stack_pointer);
                    stack_pointer := stack_pointer - 1;
                end if;
            else
                if stack_pointer < 16 then
                    stack_pointer          := stack_pointer + 1;
                    memory(stack_pointer - 1) <= I_DATA;
                end if;
            end if;

            -- Check for Empty
            if stack_pointer = 0 then
```

```
                EMPTY := true;
            else
                EMPTY := false;
            end if;

            -- Check for Full
            if stack_pointer = DEPTH then
                FULL := true;
            else
                FULL := false;
            end if;
        end if;
        O_FULL  <= to_std_logic(FULL);
        O_EMPTY <= to_std_logic(EMPTY);
    end process main;

end architecture RTL;
```

Lire Souvenirs en ligne: https://riptutorial.com/fr/vhdl/topic/9521/souvenirs

Chapitre 12: Types protégés

Remarques

Avant VHDL 1993, deux processus concurrents pouvaient communiquer uniquement avec des signaux. Grâce à la sémantique de simulation du langage qui met à jour les signaux uniquement entre les étapes de simulation, le résultat d'une simulation était déterministe: il ne dépendait pas de l'ordre choisi par le planificateur de simulation pour exécuter les processus.

[En fait, ce n'est pas vrai à 100%. Les processus peuvent également communiquer en utilisant une entrée / sortie de fichier. Mais si un concepteur compromettait le déterminisme en utilisant des fichiers, cela ne pouvait pas vraiment être une erreur.]

La langue était donc sûre. Sauf à dessein, il était presque impossible de concevoir des modèles VHDL non déterministes.

VHDL 1993 a introduit des variables partagées et la conception de modèles VHDL non déterministes est devenue très facile.

VHDL 2000 a introduit des types protégés et la contrainte que les variables partagées doivent être de type protégé.

Dans VHDL, les types protégés ressemblent à la plupart des concepts d'objets en langage orienté objet (OO). Ils implémentent l'encapsulation des structures de données et de leurs méthodes. Ils garantissent également l'accès exclusif et atomique à leurs membres de données. Cela n'empêche pas complètement le non-déterminisme mais, au moins, ajoute une exclusivité et une atomicité aux variables partagées.

Les types protégés sont très utiles lors de la conception de modèles VHDL de haut niveau destinés à la simulation uniquement. Ils ont plusieurs très bonnes propriétés des langages OO. Leur utilisation rend le code plus lisible, facile à entretenir et réutilisable.

Remarques:

- Certaines chaînes d'outils de simulation, par défaut, émettent uniquement des avertissements lorsqu'une variable partagée n'est pas d'un type protégé.
- Certains outils de synthèse ne prennent pas en charge les types protégés.
- Certains outils de synthèse ont un support limité des variables partagées.
- On pourrait penser que les variables partagées ne sont pas utilisables pour modéliser le matériel et doivent être réservées à l'instrumentation de code sans effets secondaires. Mais les modèles VHDL conseillés par plusieurs fournisseurs d'EDA pour modéliser le plan de mémoire des mémoires RAM (RAM) multi-ports utilisent des variables partagées. Donc, oui, les variables partagées peuvent être synthétisées dans certaines circonstances.

Examples

Un générateur pseudo-aléatoire

Les générateurs peu-aléatoires sont souvent utiles lors de la conception d'environnements de simulation. Le package VHDL suivant montre comment utiliser les types protégés pour concevoir un générateur pseudo-aléatoire de `boolean` , `bit` et `bit_vector` . Il peut facilement être étendu pour générer également `std_ulogic_vector` aléatoire, `signed` , `unsigned` . L'extendre pour générer des entiers aléatoires avec des limites arbitraires et une distribution uniforme est un peu plus compliqué mais réalisable.

La déclaration de package

Un type protégé a une déclaration où tous les accesseurs de sous-programmes publics sont déclarés. Pour notre générateur aléatoire, nous rendrons publique une procédure d'initialisation de graine et trois fonctions impures renvoyant un `boolean` aléatoire, un `bit` ou un `bit_vector` . Notez que les fonctions ne peuvent pas être pures car différents appels de l'un d'entre eux, avec les mêmes paramètres, peuvent renvoyer des valeurs différentes.

```
-- file rnd_pkg.vhd
package rnd_pkg is
    type rnd_generator is protected
        procedure init(seed: bit_vector);
        impure function get_boolean return boolean;
        impure function get_bit return bit;
        impure function get_bit_vector(size: positive) return bit_vector;
    end protected rnd_generator;
end package rnd_pkg;
```

Le corps du colis

Le corps de type protégé définit les structures de données internes (membres) et les corps de sous-programme. Notre générateur aléatoire est basé sur un registre à décalage à retour linéaire (LFSR) de 128 bits à quatre prises. La variable d' `state` stocke l'état actuel du LFSR. Un privé `throw` procédure déplace le LFSR chaque fois que le générateur est utilisé.

```
-- file rnd_pkg.vhd
package body rnd_pkg is
    type rnd_generator is protected body
        constant len: positive := 128;
        constant default_seed: bit_vector(1 to len) := X"8bf052e898d987c7c31fc71c1fc063bc";
        type tap_array is array(natural range <>) of positive range 1 to len;
        constant taps: tap_array(0 to 3) := (128, 126, 101, 99);

        variable state: bit_vector(1 to len) := default_seed;

        procedure throw(n: positive := 1) is
            variable tmp: bit;
        begin
            for i in 1 to n loop
                tmp := '1';
```

```
                for j in taps'range loop
                    tmp := tmp xnor state(taps(j));
                end loop;
                state := tmp & state(1 to len - 1);
            end loop;
        end procedure throw;

        procedure init(seed: bit_vector) is
            constant n:   natural            := seed'length;
            constant tmp: bit_vector(1 to n) := seed;
            constant m:   natural            := minimum(n, len);
        begin
            state         := (others => '0');
            state(1 to m) := tmp(1 to m);
        end procedure init;

        impure function get_boolean return boolean is
            constant res: boolean := state(len) = '1';
        begin
            throw;
            return res;
        end function get_boolean;

        impure function get_bit return bit is
            constant res: bit := state(len);
        begin
            throw;
            return res;
        end function get_bit;

        impure function get_bit_vector(size: positive) return bit_vector is
            variable res: bit_vector(1 to size);
        begin
            if size <= len then
                res := state(len + 1 - size to len);
                throw(size);
            else
                res(1 to len) := state;
                throw(len);
                res(len + 1 to size) := get_bit_vector(size - len);
            end if;
            return res;
        end function get_bit_vector;
    end protected body rnd_generator;
end package body rnd_pkg;
```

Le générateur aléatoire peut alors être utilisé dans un style OO comme dans:

```
-- file rnd_sim.vhd
use std.env.all;
use std.textio.all;
use work.rnd_pkg.all;

entity rnd_sim is
end entity rnd_sim;

architecture sim of rnd_sim is
    shared variable rnd: rnd_generator;
begin
    process
```

```
        variable l: line;
    begin
        rnd.init(X"fe39_3d9f_24bb_5bdc_a7d0_2572_cbff_0117");
        for i in 1 to 10 loop
            write(l, rnd.get_boolean);
            write(l, HT);
            write(l, rnd.get_bit);
            write(l, HT);
            write(l, rnd.get_bit_vector(10));
            writeline(output, l);
        end loop;
        finish;
    end process;
end architecture sim;
```

```
$ mkdir gh_work
$ ghdl -a --std=08 --workdir=gh_work rnd_pkg.vhd rnd_sim.vhd
$ ghdl -r --std=08 --workdir=gh_work rnd_sim
TRUE    1    0001000101
FALSE   0    1111111100
TRUE    1    0010110010
TRUE    1    0010010101
FALSE   0    0111110100
FALSE   1    1101110010
TRUE    1    1011010110
TRUE    1    0010010010
TRUE    1    1101100111
TRUE    1    0011100100
simulation finished @0ms
```

Lire Types protégés en ligne: https://riptutorial.com/fr/vhdl/topic/6362/types-proteges

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec vhdl	Community, DavideM, Florian, Jon Klapel, Josh, Renaud Pacalet
2	Analyse de la synchronisation statique - qu'est-ce que cela signifie lorsqu'une conception échoue dans le temps?	QuantumRipple
3	Attendez	Brian Carlton, QuantumRipple, Renaud Pacalet
4	commentaires	Renaud Pacalet
5	Conception matérielle numérique en utilisant VHDL en un mot	Renaud Pacalet
6	D-Flip-Flops (DFF) et loquets	Brian Carlton, Renaud Pacalet
7	Fonctions de résolution, types non résolus et résolus	Renaud Pacalet
8	Identifiants	Renaud Pacalet
9	Littéraux	Brian Carlton, QuantumRipple
10	Récursivité	Renaud Pacalet
11	Souvenirs	Brian Carlton, Parth Parikh, QuantumRipple, Renaud Pacalet
12	Types protégés	Renaud Pacalet