Introduction au langage VHDL

Jean-Christophe Le Lann

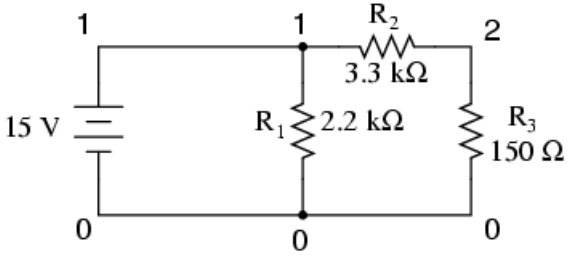
ENSTA Bretagne

4 juillet 2019

HDL : Hardware description language

- Les HDL permettent de **décrire** des systèmes.
- Ils sont donc très différents des langages de programmation classiques.

A titre d'exemple, nous donnons ici un code d'un HDL très important en Electronique *analogique* : Spice. Spice permet de décrire comment les composants usuels (résistance, capacité, inductance) sont interconnectés. Le simulateur Spice permet de rendre compte du comportement du circuit. **Nous allons procéder de même avec VHDL**, mais à plus haut niveau




```
Example netlist
v1 1 0 dc 15
r1 1 0 2.2k
r2 1 2 3.3k
r3 2 0 150
.end
```

Historique de VHDL

- VHDL est l'acronyme de : VHSIC Hardware Description Language.
- VHDL a été commandé par le Département à la Défense Américaine (DoD).
- Sa syntaxe s'inspire fortement d'ADA, très utilisé dans les années 80.
- VHDL est fortement et explicitement **typé**.
- Il permet notamment aux acteurs de la Silicon Valley d'échanger des informations/designs de manière cohérente.
- Son concurrent direct est le langage Verilog.
- VHDL parle de signaux, structures qui opèrent en parallèle, et temps : c'est un langage difficile.

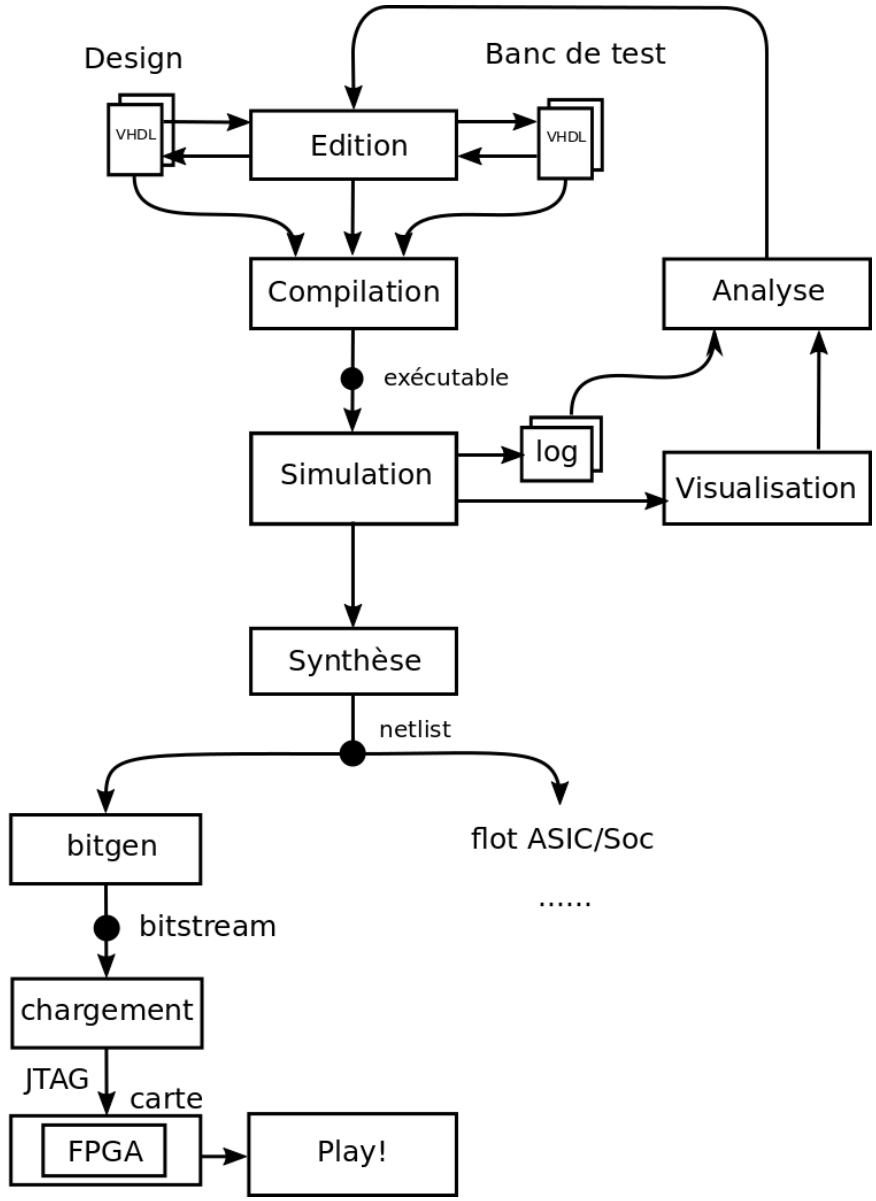
Ces langages sont standardisés par l'IEEE. Plusieurs versions pour VHDL : 87,**93**,08 notamment.

VHDL : Simulation & Synthèse

VHDL est utilisé pour deux activités différentes :

- La **simulation** : vérification fonctionnelle sur PC, du bon fonctionnement du système décrit.
- La **synthèse** : la génération automatique du système à partir des descriptions.
  - Ces descriptions sont de plus en plus abstraites.
  - On se limitera pour l'essentiel ici à décrire le circuit en terme d'équations logiques.
  - Le niveau directement plus abstrait est le niveau RTL.

Flot général



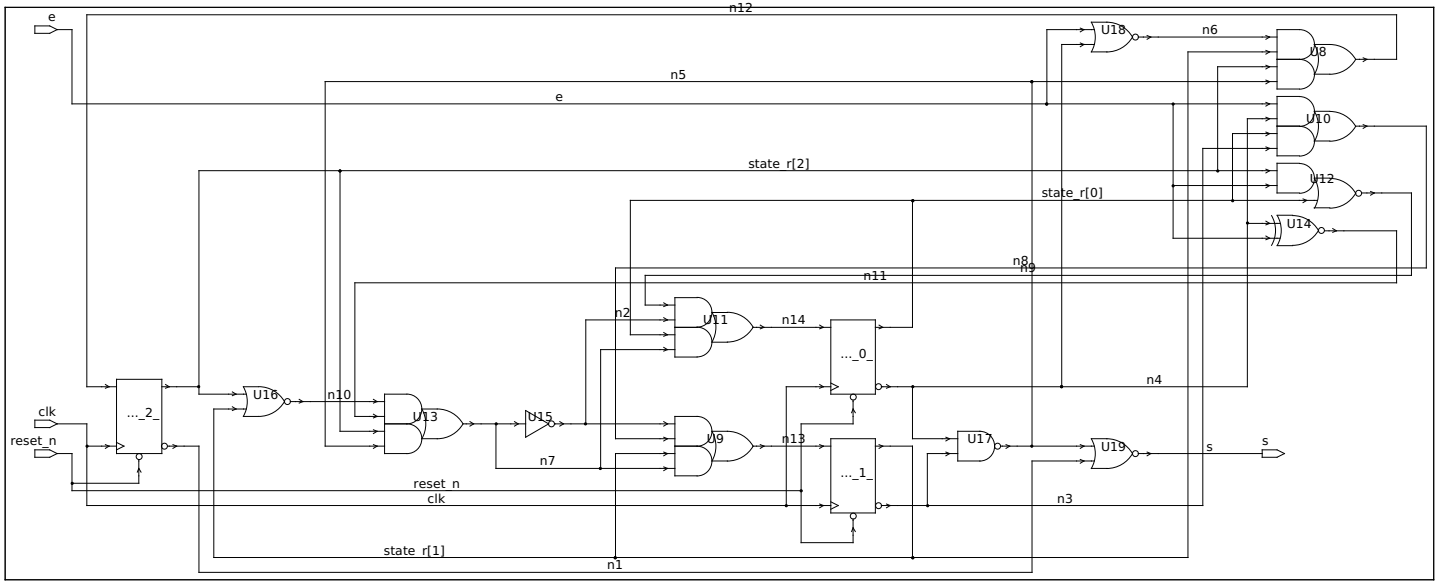
Survol de la Simulation VHDL

La simulation, réalisée sur PC :

- Repose sur un **mécanisme d'échéancier** : to-do list.
- Le temps est incrémenté petit-à-petit :
  - Temps physique
  - Temps causal : *delta delay*. Pas infinitésimaux de propagation.
- Le simulateur préserve la *causalité* entre signaux.
- C'est une tâche non-triviale pour un programme (simulateur) qui s'exécute sur un ordinateur séquentiel.

Survol de la Synthèse VHDL

- La synthèse, réalisée sur PC, permet d'obtenir un circuit (optimisé) à partir d'une description.
- Seul un sous-ensemble du langage est "synthétisable", mais il est très large.
- Le résultat est fourni sous un format de graphe précisant toutes les interconnexions : "netlist"
- Dans le cas de la synthèse sur FPGA, cette netlist est ensuite transformée en **bitstream**.



Structure globale d'une description VHDL

En première approche, on peut proposer la structure de fichier suivante :

- Déclaration des bibliothèques et packages utilisés : **library**...,**use**...
- Déclaration des entrées-sorties du circuit : **entity**
- Déclaration de l'intérieur du circuit : **architecture**

Structure globale d'une description VHDL (UE 1.2 !)

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity circuit_fsm is
  port (
    reset_n : in  std_logic;
    clk     : in  std_logic;
    e1      : in  std_logic;
    e2      : in  std_logic;
    o1      : out std_logic
    );
end circuit_fsm;

architecture solution of circuit_fsm is
  signal D, Q : std_logic_vector(1 downto 0);  --2 bits d'états
begin
  ----------------------------------------
  -- equations logiques d'etat suivant :
  ----------------------------------------
  D(1) <=   not Q(1) and Q(0) and e1 and e2;
  D(0) <=  (not Q(1) and  not Q(0) and e1 ) or (not Q(1) and Q(0) and not e2);
  ----------------------------------------
  -- equations logiques des sorties :
  -------------------------------       ---------
  o1   <= (not Q(1) and not Q(0) and e1) or (not Q(1) and Q(0) and e2) or (Q(1) and not Q(0));
  ----------------------------------------
  -- registres d'état (bascules D)
  ----------------------------------------
  regs_etat : process(reset_n, clk)
  begin
    if reset_n = '0' then
      Q <= "00";                                --etat S0
    elsif rising_edge(clk) then
      Q <= D;
    end if;
  end process;

end solution;
```

Netlist : après synthèse

```
library IEEE;
use IEEE.std_logic_1164.all;

entity circuit_fsm is
  port(reset_n, clk : in  std_logic;
       e1, e2       : in  std_logic;
       o1           : out std_logic);
end circuit_fsm;

architecture SYN_solution of circuit_fsm is

  component NAND2X1
    port(IN1, IN2 : in std_logic; QN : out std_logic);
  end component;

  -- code supprimé...

  component DFFARX1
    port(D, CLK, RSTB : in std_logic; Q, QN : out std_logic);
  end component;

  signal Q, D : std_logic_vector(1 downto 0);

  signal n1, n2, n3, n4, n5, n6 : std_logic;

begin

  Q_reg_0_inst : DFFARX1 port map(D => D(0), CLK => clk, RSTB => reset_n, Q
                                  => Q(0), QN => n2);
  Q_reg_1_inst : DFFARX1 port map(D => D(1), CLK => clk, RSTB => reset_n, Q
                                  => Q(1), QN => n1);
  U7  : AO21X1  port map(IN1  => n3, IN2 => n1, IN3 => Q(0), Q => n5);
  U8  : NAND3X0 port map(IN1 => Q(0), IN2 => n1, IN3 => e2, QN => n4);
  U9  : OA22X1  port map(IN1  => Q(0), IN2 => n3, IN3 => e2, IN4 => n2, Q => n6);
  U10 : NOR2X0  port map(IN1  => n4, IN2 => n3, QN => D(1));
  U11 : NOR2X0  port map(IN1  => Q(1), IN2 => n6, QN => D(0));
  U12 : INVX0   port map(INP   => e1, ZN => n3);
  U13 : NAND2X1 port map(IN1 => n4, IN2 => n5, QN => o1);

end SYN_solution;
```

Notion d'entité

L'**entity** est la **vue extérieure** d'un composant.

- Nom du composant
- Liste de ports :
  - Entrées : *nom* : **in** *type*;
  - Sorties : *nom* : **out** *type*;
  - Il est possible de déclarer plusieurs ports comme ceci : *n*1, *n*2, *n*3 : **out** *type*;

Type en VHDL

On dispose de type de base, inclus dans le langage (bit par exemple), mais on repose plutôt sur des librairies complémentaires :

- std_logic_1164 : type std_logic, std_logic_vector
- std_logic : '1','0', mais aussi 'Z','X','U','L','H'
  - '0','1' : logique booléenne traditionnelle
  - 'U' : signal 'Undefined' (non connecté)
  - 'X' : signal en conflit. Plusieurs connexions cherchent à affecter le signal.
  - 'Z' : haute impédance.
  - 'L','H' : signal à peu près à '0' et '1' (peu usités)

Notion d'architecture

- L'architecture : la vue **interne** d'un composant.
- Partie déclaration suivie d'un corps de l'architecture.
- Le corps contient des **éléments qui fonctionnent en parallèle** !
  - L'ordre dans lesquels ils sont écrits est indifférent !
  - On parle plutôt de **concurrence** (en simulation ces éléments partagent le processeur qui simule...)

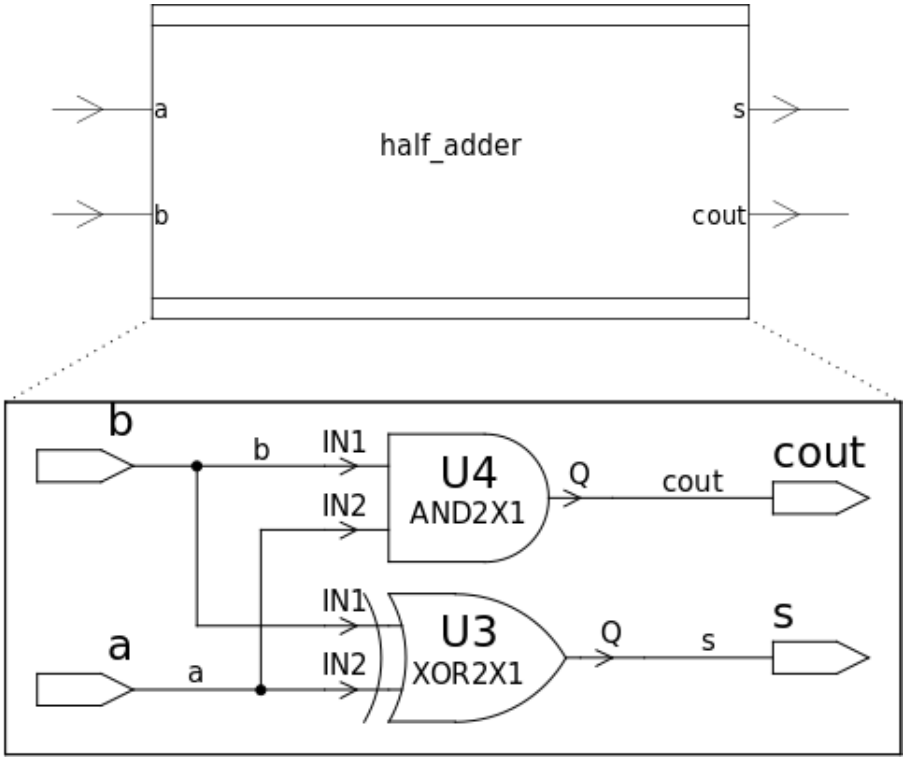
Notion d'architecture : éléments concurrents

- Assignations de signaux : comme des équations.
  - les assignations peuvent être conditionnelles ou non.
- Instanciations de composants : assemblage hiérarchique.
- Processus : codes séquentiels isolés les uns des autres.

Nous allons les passer en revue ici.

Assignations concurrentes

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

entity half_adder is
  port(
    a, b    : in  std_logic;
    s, cout : out std_logic;
    );
end entity;

architecture logic of half_adder is
begin

  s    <= a xor b;
  cout <= a and b;

end logic;
```

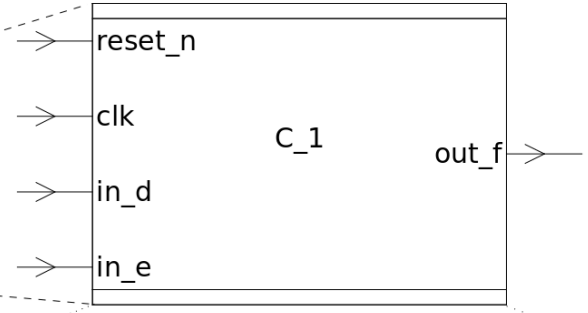


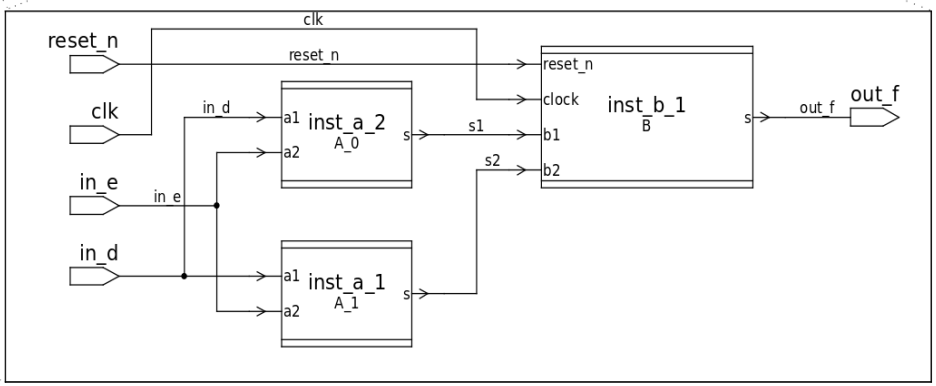
Instanciation de composants

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

entity A is
  port(
    a1 : in  std_logic;
    a2 : in  std_logic;
    s  : out std_logic
    );
end entity;

architecture simple of A is
begin
  s <= a1 and a2;
end simple;
```

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

entity B is
  port(
    reset_n : in  std_logic;
    clock   : in  std_logic;
    b1, b2  : in  std_logic;
    s       : out std_logic
    );
end entity;

architecture simple of B is
begin
  process(reset_n, clock)
  begin
    if reset_n = '0' then
      s <= '0';
    elsif rising_edge(clock) then
      s <= b1 xor b2;
    end if;
  end process;
end simple;
```

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

entity C is
  port(
    reset_n : in  std_logic;
    clk     : in  std_logic;
    in_d    : in  std_logic;
    in_e    : in  std_logic;
    out_f   : out std_logic
    );
end entity;

architecture rtl of C is
  signal s1, s2 : std_logic;
begin

  inst_a_1 : entity work.A(rtl)
    port map(
      a1 => in_d,
      a2 => in_e,
      s  => s2
      );

  inst_a_2 : entity work.A(rtl)
    port map(
      a1 => in_d,
      a2 => in_e,
      s  => s1
      );

  inst_b_1 : entity work.B(rtl)
    port map(
      reset_n => reset_n,
      clock   => clk,
      b1      => s1,
      b2      => s2,
      s       => out_f
      );

end rtl;
```



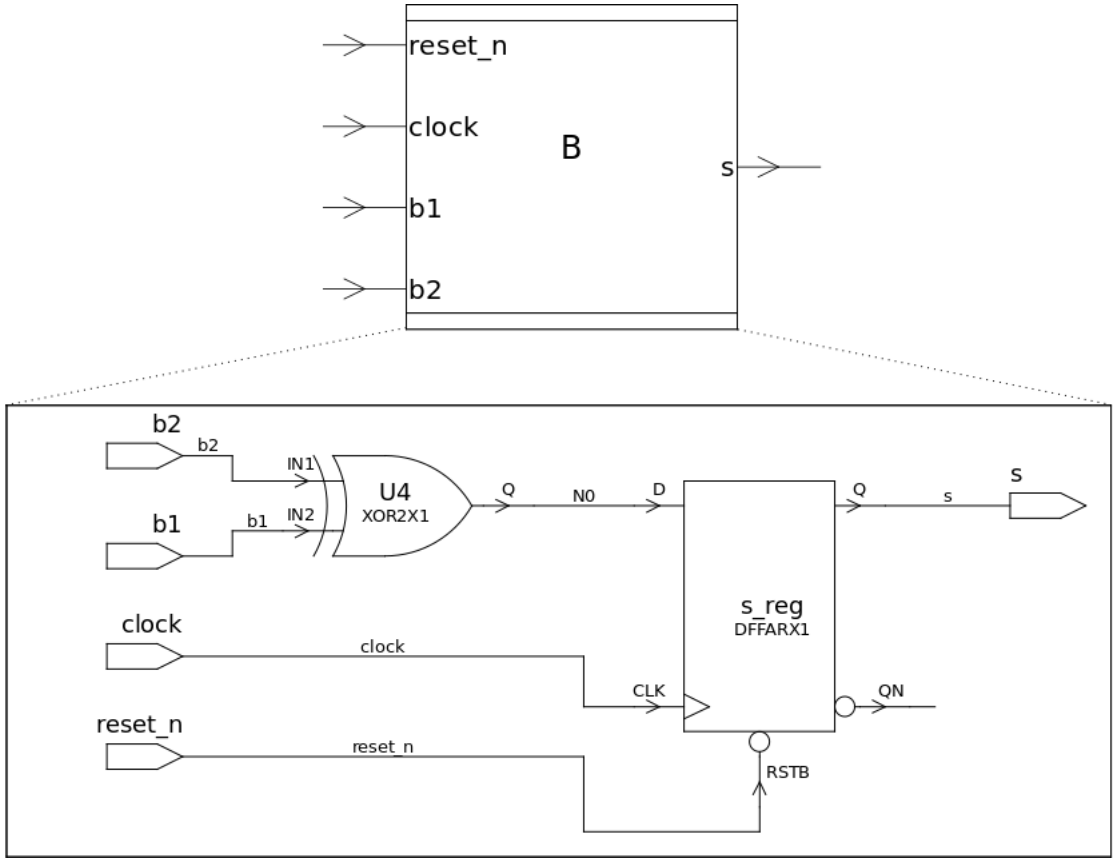


Processus concurrents

- Un processus est un petit programme séquentiel.
- Les processus fonctionnent en parallèle (concurrence).
- Ils communiquent par signaux.
- Une **liste de sensibilité** explicite permet au simulateur de savoir quand (i.e sur quel changement de valeur) il doit ré-évaluer le processus.
- Du point de vue du simulateur, les processus s'exécutent jusqu'à rencontrer un éventuel **wait**. Sinon, ils rendent la main, avant de se ré-executer.
- Du point de vue de la synthèse, les processus nous permettront notamment de décrire les bascules D.

Processus concurrents : exemple

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

entity B is
  port(
    reset_n : in  std_logic;
    clock   : in  std_logic;
    b1, b2  : in  std_logic;
    s       : out std_logic
    );
end entity;

architecture simple of B is

begin

  process(reset_n, clock)
  begin
    if reset_n = '0' then
      s <= '0';
    elsif rising_edge(clock) then
      s <= b1 xor b2;
    end if;
  end process;

end simple;
```

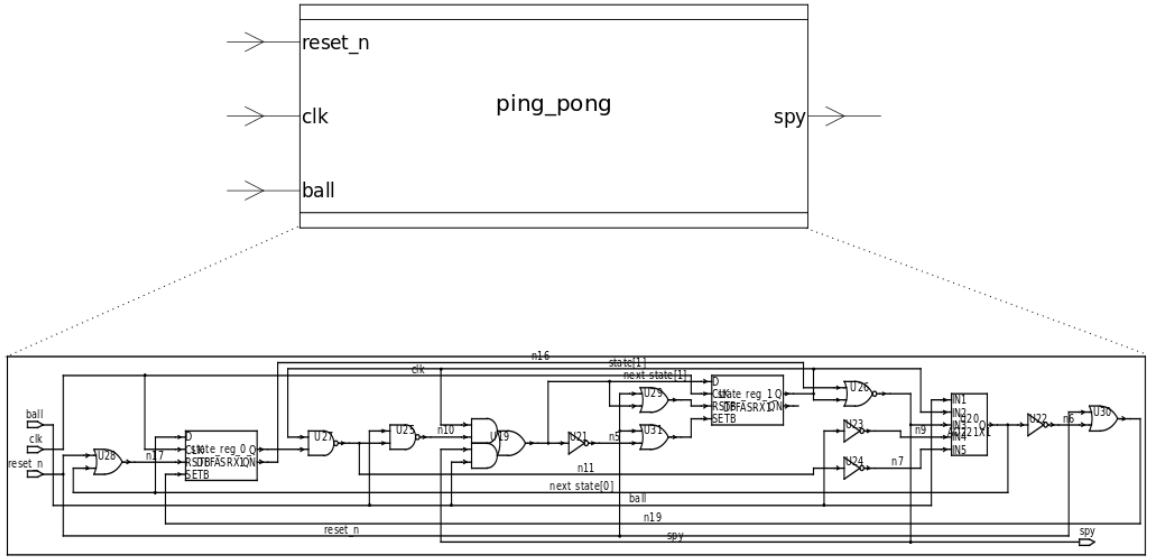


Décrire des automates : en équations

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity circuit_fsm is
  port (
    reset_n : in  std_logic;
    clk     : in  std_logic;
    e1      : in  std_logic;
    e2      : in  std_logic;
    o1      : out std_logic
    );
end circuit_fsm;

architecture solution of circuit_fsm is
  signal D, Q : std_logic_vector(1 downto 0);  --2 bits d'états
begin
  ----------------------------------------
  -- equations logiques d'etat suivant :
  ----------------------------------------
  D(1) <=   not Q(1) and Q(0) and e1 and e2;
  D(0) <=  (not Q(1) and  not Q(0) and e1 ) or (not Q(1) and Q(0) and not e2);
  ----------------------------------------
  -- equations logiques des sorties :
  -------------------------------       ---------
  o1   <= (not Q(1) and not Q(0) and e1) or (not Q(1) and Q(0) and e2) or (Q(1) and not Q(0));
  ----------------------------------------
  -- registres d'état (bascules D)
  ----------------------------------------
  regs_etat : process(reset_n, clk)
  begin
    if reset_n = '0' then
      Q <= "00";                                --etat S0
    elsif rising_edge(clk) then
      Q <= D;
    end if;
  end process;

end solution;
```

Décrire des automates : en RTL

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

entity ping_pong is
  port(
    reset_n : in  std_logic;
    clk     : in  std_logic;
    ball    : in  std_logic;
    spy     : out std_logic
    );
end entity;

architecture rtl of ping_pong is

  -- creation d'un type enumeré pour les états symboliques.
  type state_t is (IDLE, PING, PONG);

  signal state, next_state : state_t;

begin

  bascules_d : process(reset_n, clk)
  begin
    if reset_n = '0' then
      state <= next_state;
    elsif rising_edge(clk) then
      state <= next_state;
    end if;
  end process;


  next_state_func : process(ball, state)
  begin
    --pas defaut, reste dans le meme etat
    next_state <= state;

    case state is

      when PING =>
        if ball = '1' then
          next_state <= PONG;
        end if;

      when PONG =>
        if ball = '1' then
          next_state <= PING;
        end if;

      when others =>
        null;
    end case;

  end process;

  spy <= '1' when state = PING else '0';


end rtl;
```



Décrire des automates : en RTL

- Les états sont représentés par un **type énumeré**.
- Deux signaux **state,next_state** codent l'état courant et l'état futur.
- Un processus combinatoire permet d'expliciter le signal *next_state*, à l'aide de différents *cas* présents dans le **case...when** du langage.
- Un processus séquentiel réalise l'échantillonnage et l'affectation de *next_state* dans *state*

Décrire des automates : en RTL

- A ce stade, le synthétiseur a suffisamment d'informations, pour **inférer** les équations logiques.
- Il a au préalable choisi un encodage de l'état. Cet encodage peut être forcé dans VHDL par des attributs ou guidés par des scripts.
- Le synthétiseur est capable d'explorer plusieurs solutions, notamment en fonction de l'encodage.
- Les sorties peuvent être encodées dans un troisème processus, mais généralement on les retrouvent dans le même processus que le calcul de l'état suivant.

Bancs de tests

Les *testbenchs* représentent un **laboratoire virtuel**, où l'on retrouve :

- le circuit à tester (DUT)
- des générateurs de signaux : horloge, reset, flux de données complexes.
- des instruments d'observations : loggers, comparateurs, etc
- la simulation elle-même fournit un oscilloscope, capable d'observer :
  - la totalité des signaux
  - sur une durée limitée par la seule mémoire de l'ordinateur

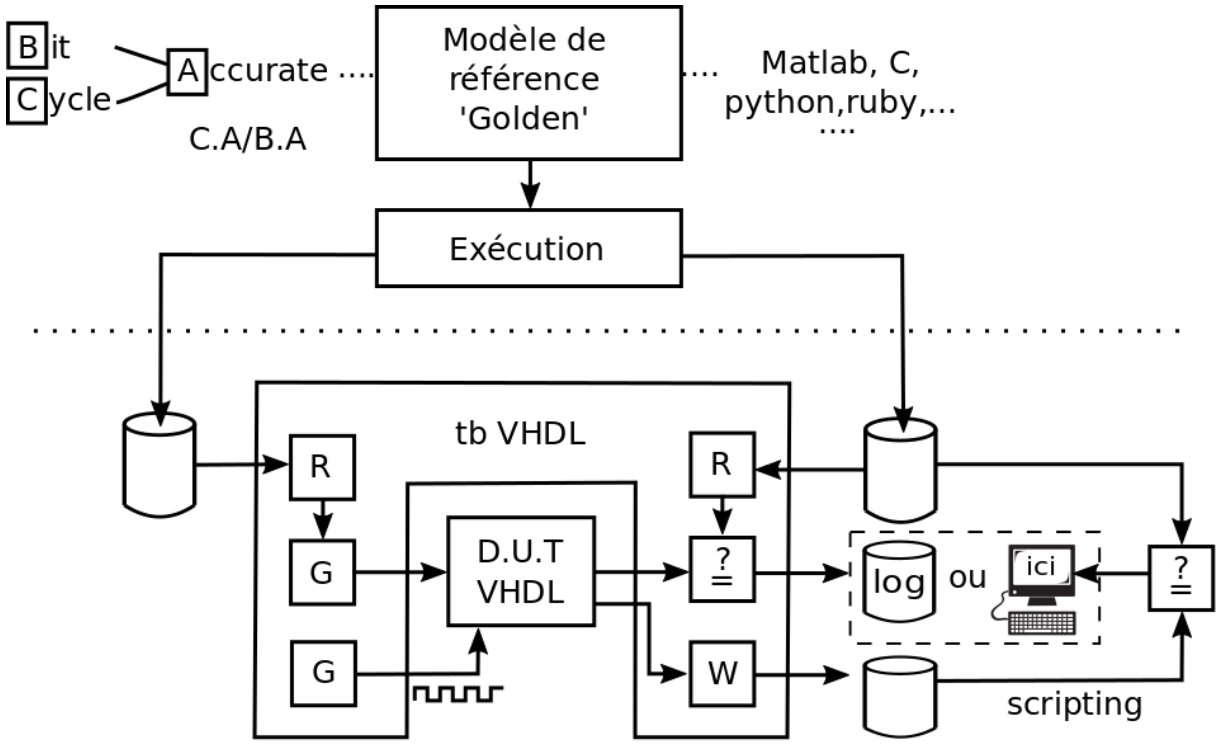
A noter :

- seul un testbench est simulable.
- les bancs de test sont également décrits par un couple entité-architecture.
- il est logique que cette entité ne présente aucun port : laboratoire portes closes !

Bancs de tests

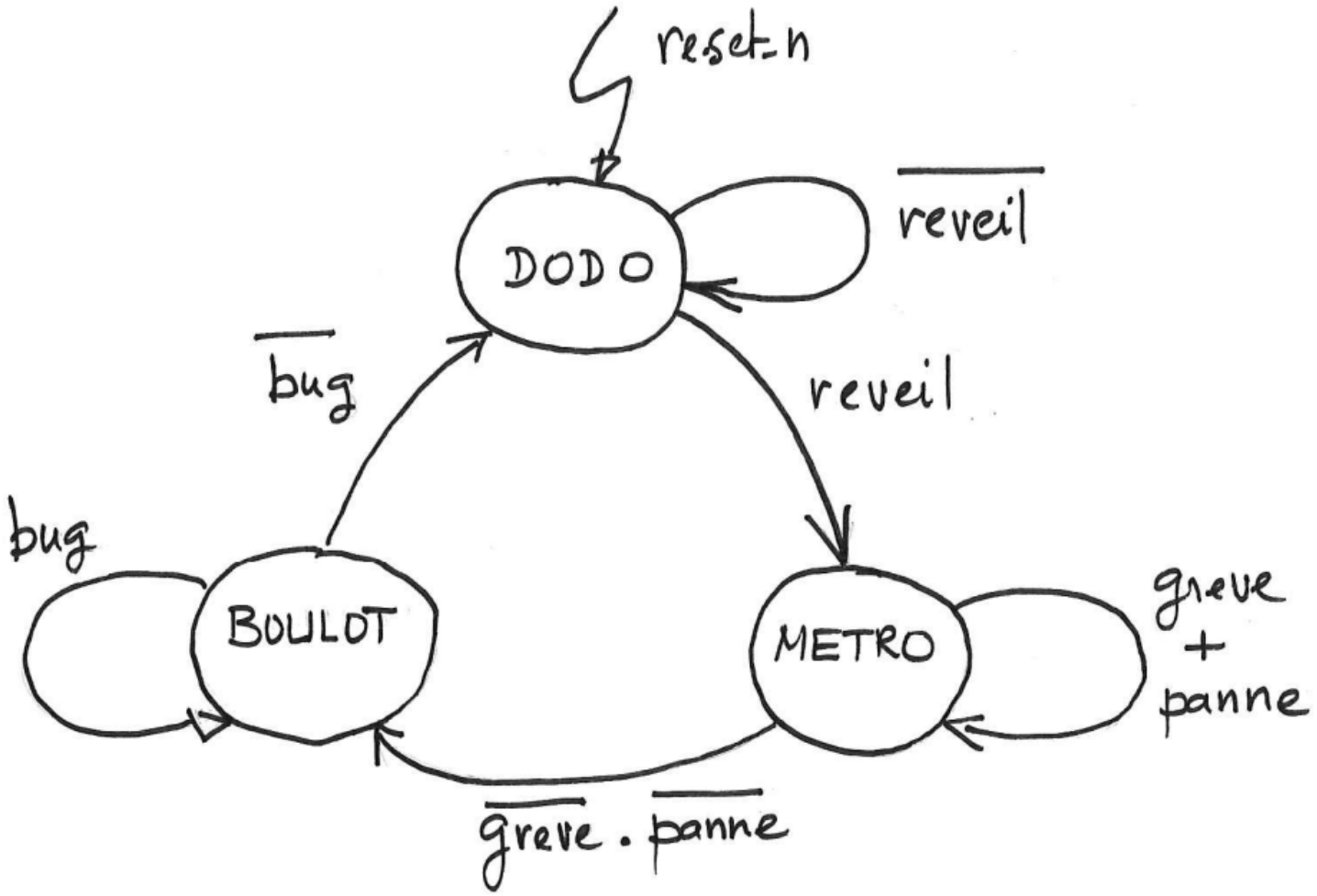
- Les *testbenchs* VHDL sont le seul moyen de tester si un circuit fonctionne correctement : un circuit seul ne fonctionne pas.
- Les *testbenchs* peuvent être de différente nature et représentent l'essentiel du travail de conception.
  - Testbenchs qui stimulent seulement les entrées, à partir de stimuli internes.
  - Testbenchs qui stimulent seulement les entrées, à partir de stimuli externes (matlab, python, ruby, c, c++).
  - Testbenchs qui vérifient les sorties, en calculant en interne les sorties attendues.
  - Testbenchs qui vérifient les sorties, en les comparant à des valeurs calculées par un modèle externe : le **golden model**.
  - Testbenchs qui "randomisent" la vérification, afin d'assurer une **couverture de code**.

Banc de test et Golden model



R	lecture de fichier
W	écriture de fichier
G	génération de signaux
? =	comparaison

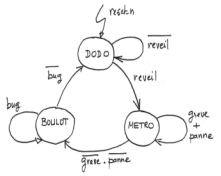
Banc de test VHDL : "I love Paris" (TD)



Banc de test VHDL : exemple

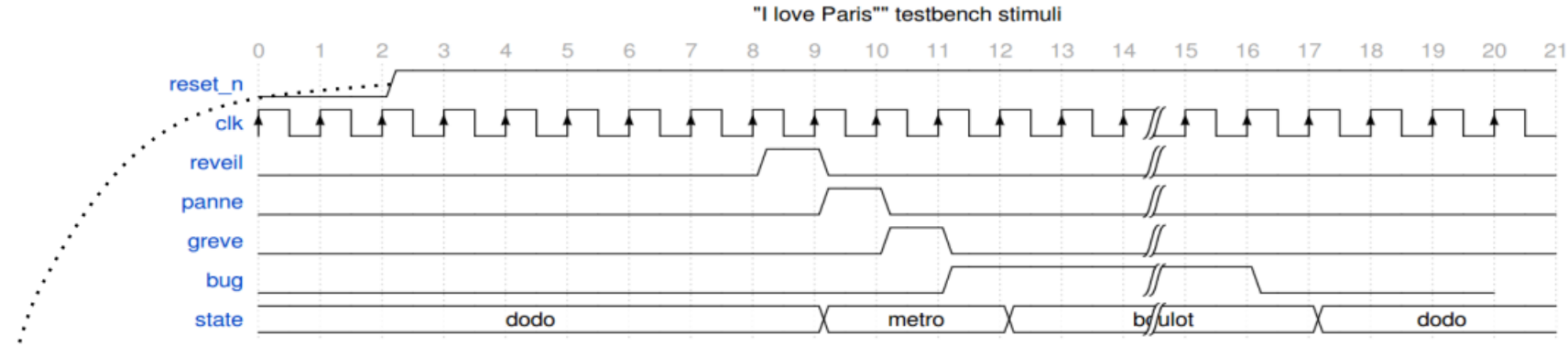
```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity i_love_paris_tb is
end entity;

architecture bhv of i_love_paris_tb is

  constant HALF_PERIOD : time := 5 ns;

  signal clk     : std_logic := '0';
  signal reset_n : std_logic := '0';
  signal sreset  : std_logic := '0';
  signal running : boolean   := true;

  procedure wait_cycles(n : natural) is
  begin
    for i in 1 to n loop
      wait until rising_edge(clk);
    end loop;
  end procedure;

  signal reveil         : std_logic;
  signal panne          : std_logic;
  signal greve          : std_logic;
  signal bug            : std_logic;
  signal up_and_running : std_logic;

begin
  ------------------------------------------------------------------------
  -- clock and reset
  ------------------------------------------------------------------------
  reset_n <= '0', '1' after 33 ns;

  clk <= not(clk) after HALF_PERIOD when running else clk;
```

```vhdl
  ------------------------------------------------------------------------
  -- Design Under Test
  ------------------------------------------------------------------------
  dut : entity work.i_love_paris(rtl)
    port map (
      reset_n         => reset_n,
      clk             => clk,
      reveil          => reveil,
      panne           => panne,
      greve           => greve,
      bug             => bug,
      up_and_running => up_and_running
      );
```




```vhdl
  ------------------------------------------------------------------------
  -- sequential stimuli
  ------------------------------------------------------------------------
  stim : process
  begin
    reveil <= '0';
    panne  <= '0';
    greve  <= '0';
    bug    <= '0';
    report "running testbench for i_love_paris(rtl)";
    report "waiting for asynchronous reset";
    wait until reset_n = '1';
    wait_cycles(5);
    report "applying stimuli...";
    wait until rising_edge(clk);
    report "REVEIL !";
    reveil <= '1';

    wait until rising_edge(clk);
    report "PANNE du METRO!";
    reveil <= '0';
    panne  <= '1';

    wait until rising_edge(clk);
    report "GREVE !";
    panne  <= '0';
    greve  <= '1';

    wait until rising_edge(clk);
    report "BUG !";
    greve  <= '0';
    bug    <= '1';
    wait_cycles(10);

    wait until rising_edge(clk);
    report "BUG résolu";
    bug    <= '0';
    wait_cycles(10);
    report "end of simulation";
    running <= false;
    wait;
  end process;

end bhv;
```

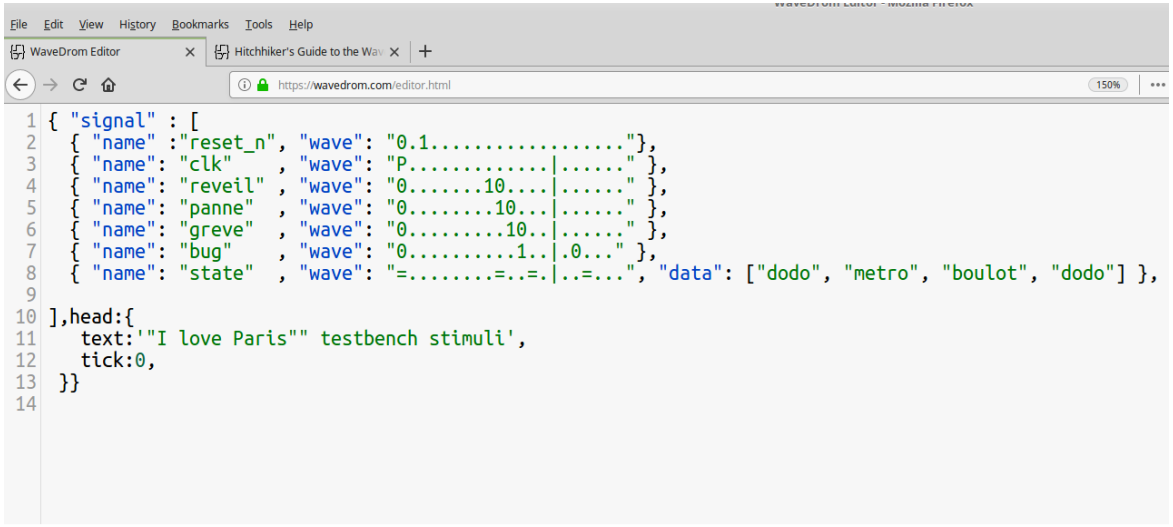
Banc de test VHDL : exemple

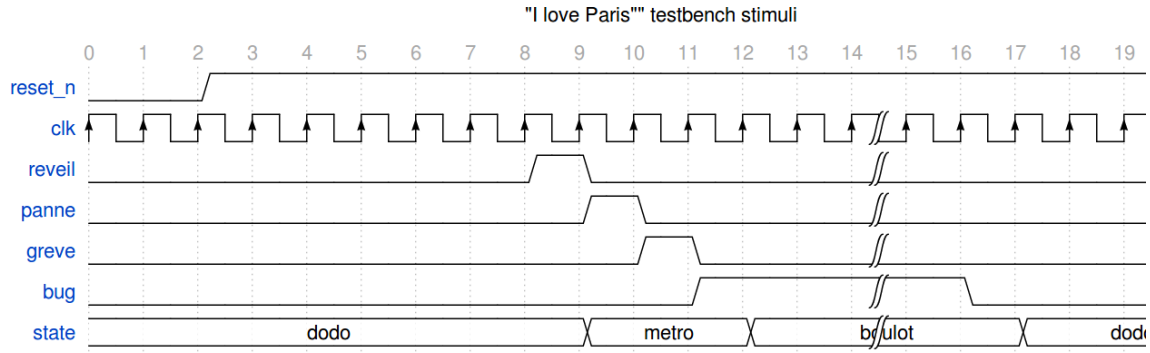



```
-- sequential stimuli
stim : process
begin
  reveil <= '0';
  panne  <= '0';
  greve  <= '0';
  bug    <= '0';
  report "running testbench for i_love_paris(rtl)";
  report "waiting for asynchronous reset";
  wait until reset_n = '1';
  wait_cycles(5);
  report "applying stimuli...";
  wait until rising_edge(clk);
  report "REVEIL !";
  reveil <= '1';

  wait until rising_edge(clk);
  report "PANNE du METRO!";
  reveil <= '0';
  panne  <= '1';

  wait until rising_edge(clk);
  report "GREVE !";
  panne  <= '0';
  greve  <= '1';

  wait until rising_edge(clk);
  report "BUG !";
  greve  <= '0';
  bug    <= '1';
  wait_cycles(10);

  wait until rising_edge(clk);
  report "BUG résolu";
  bug    <= '0';
  wait_cycles(10);
  report "end of simulation";
  running <= false;
  wait;
end process;
```

Pour la petite histoire...

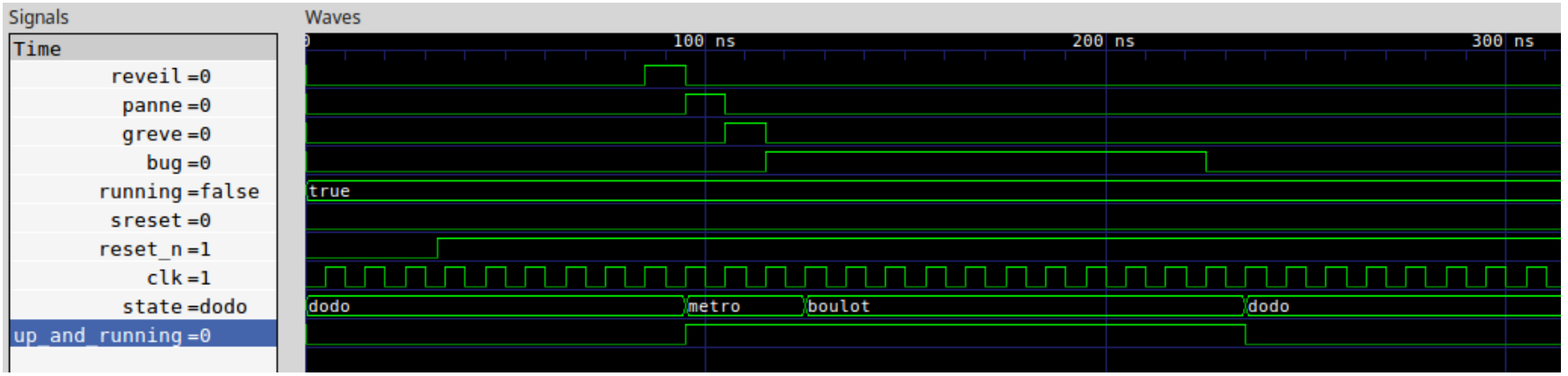
Les *timing diagrams* ont été crées avec une syntaxe textuelle ! Le compilateur transforme ces *timing diagrams* en graphique SVG dans html5.





Banc de test VHDL : exemple

Chronogrammes issus de la chaîne de compilation-simulation-visualisation autour de GHDL+Gtkwave.



Simulation avec GHDL

GHDL est un simulateur open source développé par un français : Tristan Gingold.

- Il est open source. Développé en ADA.
- Il supporte les évolutions récentes du langage.
- Il est scrupuleux sur le respect de la norme.
- Il présente encore quelques bugs.
- Ce n'est pas le simulateur le plus utilisé au Monde : il s'agit de Modelsim.

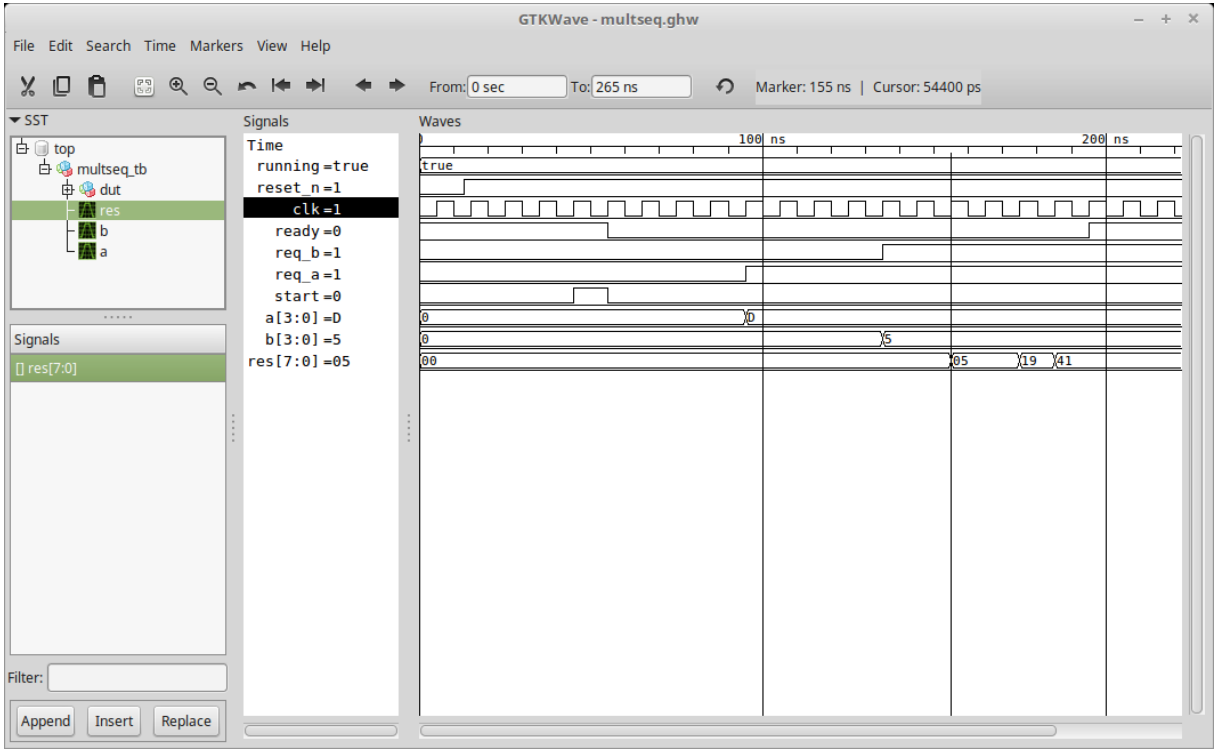
Simulation avec GHDL

Pour simuler un testbench, il faut passer par 3 phases :

- **A**nalyse : **ghdl -a circuit.vhd**
- ...
- **A**nalyse : **ghdl -a circuit_tb.vhd**
- **E**laboration : **ghdl -e circuit_tb** (sans .vhd).
  - cette phase réunit les fichiers binaires produits par l'analyse
  - ...adjoint un *runtime* (le coeur du simulateur)
  - ...et produit un exécutable : le simulateur
- **R**un : **ghdl -r circuit_tb –wave=waves.ghw**.
  - Le simulateur s'exécute et enregistre les signaux dans le fichier waves.
- Un viewer externe est alors utilisé pour visualiser les signaux : **gtkwave waves.ghw**.
- La disposition des signaux est enregistrable dans un fichier spécifique à gtkwave (.sav).
- Il faut **scripter** l'ensemble.

Visualisation avec Gtkwave

gtkwave wave.ghw chrono.sav Le fichier chrono.sav correspond à la mise en forme des chronogrammes, sauvée dans ce fichier.



Un mot sur les prochains TP VHDL

1. Création d'un design hiérarchique : additionneur 8 bits. Compilation et simulation d'un banc de test.
2. Design séquentiel en VHDL. Design logique vs RTL dans le cas des Automates.
3. Synthèse sur FPGA : moyenne mobile *ou* coeur de processeur *ou* serrure numérique.