

Introduction à l'Electronique Numérique

Jean-Christophe Le Lann

Cours 1A

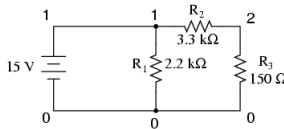
8 octobre 2026



2– Introduction aux langages de description matériels : VHDL

- Les HDL permettent de **décrire** des systèmes.
 - ▶ ...de la même manière que `html` permet de décrire le contenu d'une page web.
- 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
```

- VHDL est l'acronyme de : VHSIC Hardware Description Language.
 - ▶ VHSIC : Very High Speed Integrated Circuit
- VHDL a été commandé par le Département à la Défense Américaine (DoD).
- Sa syntaxe s'inspire fortement d'**ADA**, très utilisé en *logiciel* dans les années 80.
- VHDL est fortement et explicitement **typé**.
- Le but initial était de permettre aux acteurs de la Silicon Valley de seulement échanger des informations/designs de manière cohérente.
- Son concurrent direct est le langage **Verilog**.
- VHDL est un langage *réputé difficile*. Il parle de :
 - ▶ structures qui opèrent en parallèle
 - ▶ signaux
 - ▶ deux formes de temps

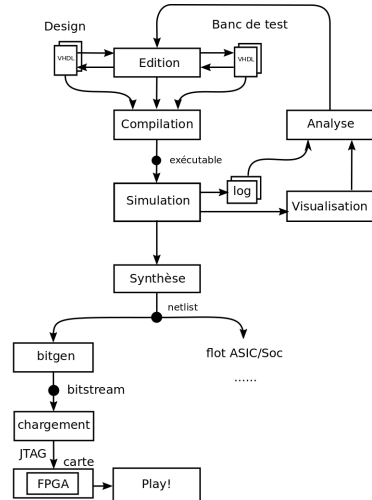
Ces langages sont standardisés par l'IEEE. Plusieurs versions pour VHDL : 87,**93**,08 notamment.

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 essentiellement dans ce cours à décrire le circuit en terme d'équations logiques.
 - ▶ Le niveau directement plus abstrait est le niveau RTL.

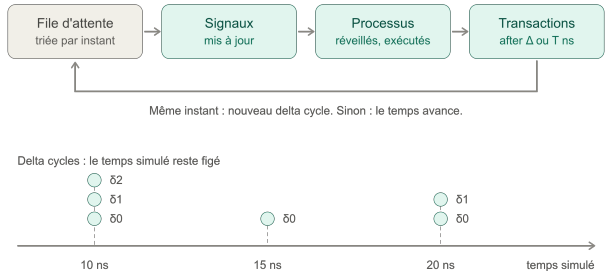
Flot de conception

- L'ingénierie des systèmes numériques requiert une organisation méticuleuse.
- Autant techniquement qu'humainement.



La simulation, réalisée sur PC :

- Repose sur un **mécanisme d'échéancier** :
 - ▶ Event-driven simulator.
 - ▶ Image d'une *to-do list*.
- Le temps est incrémenté petit-à-petit :
 - ▶ **Temps physique** simulé
 - ▶ **Temps causal** : *delta delay/cycles*. 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.



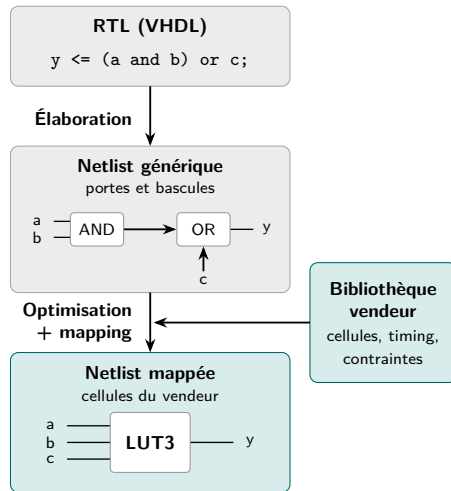
Définition

Transformer une description RTL (VHDL synthétisable) en une *netlist* de cellules d'une technologie cible (FPGA ou ASIC).

Étapes principales

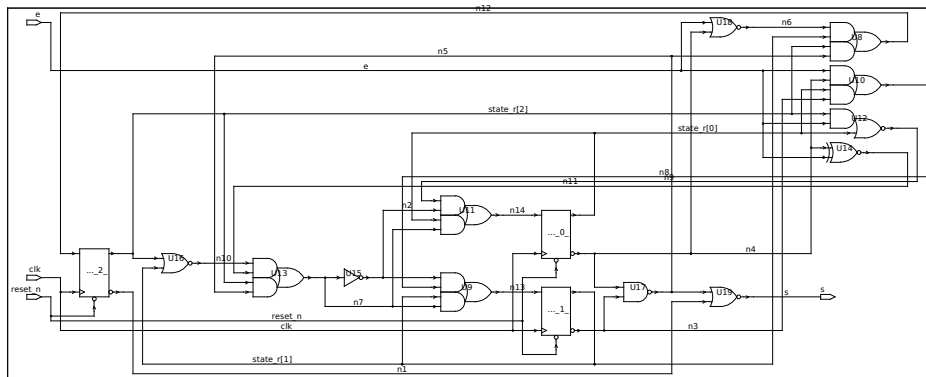
- 1 **Élaboration** : analyse du code, résolution de la hiérarchie, inférence de logique combinatoire et de registres.
- 2 **Optimisation** : minimisation booléenne, partage de sous-expressions, suppression de la logique inutile.
- 3 **Mapping technologique** : recouvrement du graphe de portes par les cellules du vendeur (LUT, bascules, cellules standard).

Le mapping est piloté par la **bibliothèque vendeur** et par les **contraintes** (fréquence, surface, puissance).



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**

Le monde du parallélisme

- Entre le begin...end de l'architecture
- **ordre indifférent** des "instructions"...
- ...ou plutôt des **déclarations**.

```
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'etat (bascules D)
  -----
  regs_etat : process(reset_n, clk)
  begin
    if reset_n = '0' then
      Q <= "00";
    elsif rising_edge(clk) then
      Q <= D;
    end if;
  end process;
end solution;
```

```
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;

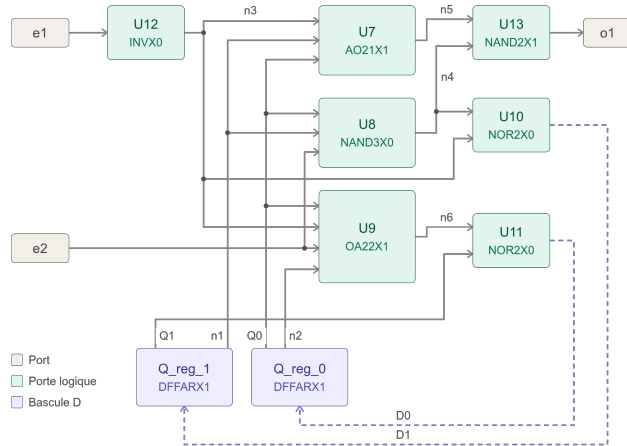
  signal Q, D : std_logic_vector(1 downto 0);
  signal n1, n2, n3, n4, n5, n6 : std_logic;

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;
```



Pointillés : retour d'état D0, D1. clk et reset_n sont reliés aux deux bascules.

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

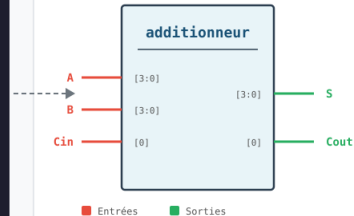
- Nom du composant
- Liste de ports :
 - ▶ Entrées : *nom : in type;*
 - ▶ Sorties : *nom : out type;*
 - ▶ Il est possible de déclarer plusieurs ports comme ceci :
n1, n2, n3 : out type;

Entité VHDL et son enveloppe extérieure

```
-- Fichier : additionneur.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.    ALL;
use IEEE.NUMERIC_STD.       ALL;

entity additionneur is
  port (
    A : in std_logic_vector (3 downto 0);
    B : in std_logic_vector (3 downto 0);
    Cin : in std_logic ;
    S : out std_logic_vector (3 downto 0);
    Cout : out std_logic
  );
end additionneur ;
```

Enveloppe extérieure (Boîtier / Block Symbol)



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...)

Éléments concurrents

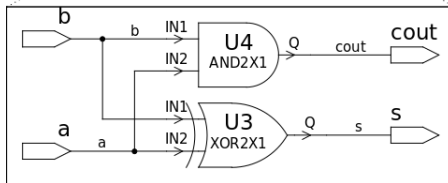
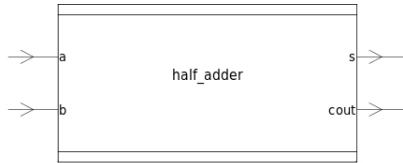
- **Assignations de signaux** : comme des équations.
 - ▶ les assignations peuvent être conditionnelles ou non.
- **Instanciations de composants** : assemblage hiérarchique.
- **Processus** : codes séquentiels isolés les uns des autres.

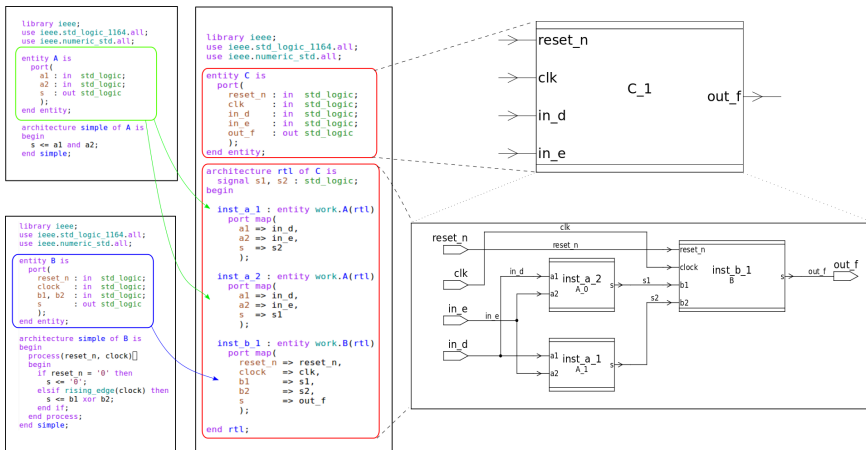
Nous allons les passer en revue ici.

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

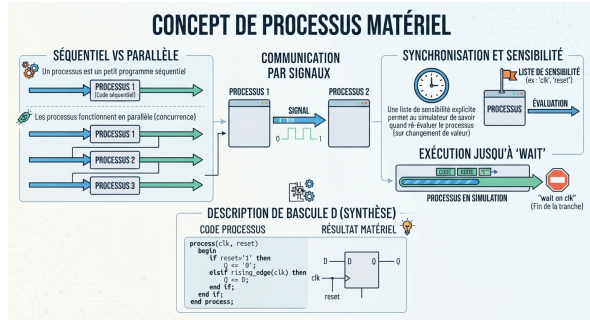
entity half_adder is
  port(
    a, b : in std_logic;
    s, cout : out std_logic;
  );
end entity;

architecture logic of half_adder is
begin
  s <= a xor b;
  cout <= a and b;
end logic;
```





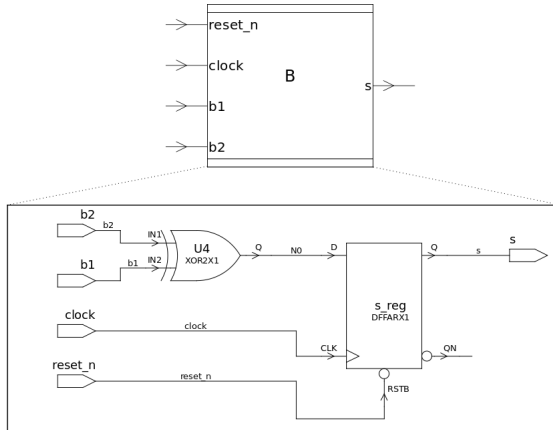
- Un processus est un petit programme séquentiel.
- Les processus fonctionnent en parallèle (concurrency).
- Ils communiquent par signaux.
- Une **liste de sensibilité** explicite permet au simulateur de savoir quand il doit ré-évaluer le processus.
 - ▶ observation permanente des changements de valeurs impactant 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-exécuter.
- Du point de vue de la synthèse, les processus nous permettront notamment de décrire les bascules D.



```
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;
```



Types en VHDL

On dispose de type de base, inclus dans le langage (bit par exemple), mais on repose plutôt sur des bibliothèques 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)
- **numeric_std** : arithmétique traditionnelle.
- etc.

Il est possible de créer ses propres types.

- types **structurés** : record
- types **énumérés** : enum
- types **tableaux** : array..of..

Décrire des automates : en équations

```
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'etat (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

```
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 énuméré pour les états symboliques.
  type state_t is (IDLE, PING, PONG);
  signal state, next_state : state_t;

begin

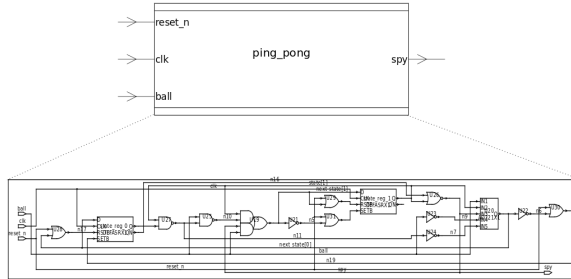
  bascules_d : process(reset_n, clk)
  begin
    if reset_n = '0' then
      state <= next_state;
    elsif rising_edge(clk) then
      state <= next_state;
    end if;
  end process;

  next_state_func : process(ball, state)
  begin
    -- pas default, reste dans le meme état
    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;
```



Le synthétiseur est capable d'*inférer* les équations logiques de ce style de descriptions RTL.

- Les états sont représentés par un **type énuméré**.
- 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*

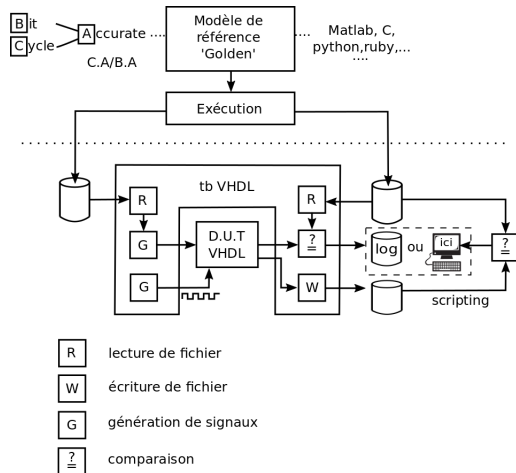
- A ce stade, le synthétiseur a suffisamment d'informations, pour **inférer** les équations logiques.
- Il a au préalable choisi un encodage de l'état. Cet encodage peut être forcé dans VHDL par des attributs ou guidés par des scripts.
- Le synthétiseur est capable d'explorer plusieurs solutions, notamment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

Les *testbenchs* représentent un **laboratoire virtuel** :

- 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

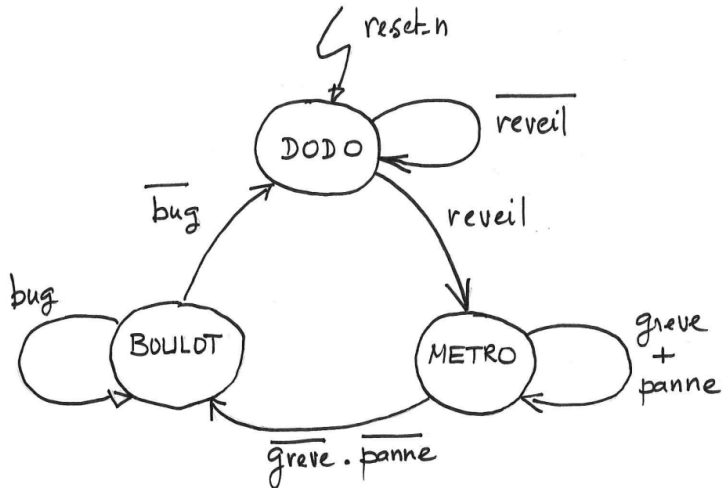
A noter :

- seul un testbench est simulable.
- les bancs de test sont également décrits par un couple entité-architecture.
- *entity* sans port : laboratoire portes closes !



- 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 différentes natures et représentent l'essentiel du travail de concepteur.
 - ▶ 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 VHDL : FSM metro-boulot-dodo



Banc de test VHDL : exemple

```
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 rreset   : 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;
```

```
-----
-- 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
);
```



```
-----
-- 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';

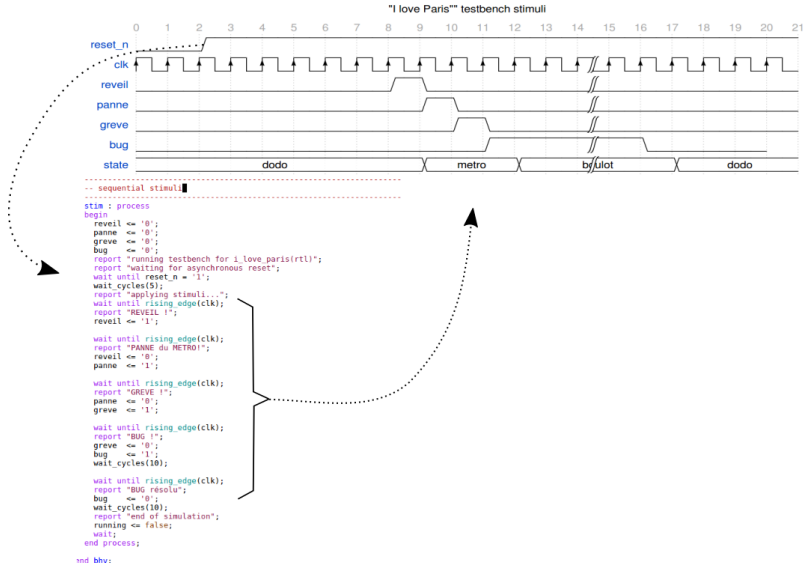
    wait until rising_edge(clk);
    report "PANNE du METRO!";
    panne <= '0';
    panne <= '1';

    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;
```

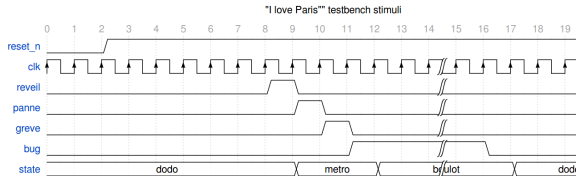
Banc de test VHDL : exemple



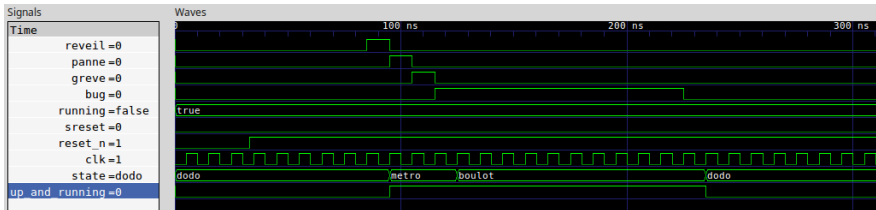
Pour la petite histoire...

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

```
1 { "signal": [
2   { "name": "reset_n", "wave": "0.1....."},
3   { "name": "clk", "wave": "P....."},
4   { "name": "reveil", "wave": "0.....10....."},
5   { "name": "panne", "wave": "0.....10....."},
6   { "name": "greve", "wave": "0.....10....."},
7   { "name": "bug", "wave": "0.....1..0.."},
8   { "name": "state", "wave": "=.....=.=..=..", "data": ["dodo", "metro", "boulot", "dodo"] },
9
10 ], head: {
11   text: "I love Paris" testbench stimuli',
12   tick: 0,
13 }}
14
```



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



GHDL est un simulateur et synthétiseur 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.
- On peut le coupler à gtkwave pour l'examen des signaux.
- Ce n'est pas le simulateur le plus utilisé au Monde : il s'agit de Modelsim.

NVC est un autre simulateur (pas de synthèse) open source développé par un britannique : Nick Gasson.



Figure – Tristan Gingold, ingénieur de Recherche au C.E.R.N, Genève (Telecom Paris 96, DEA Microélectronique (Paris 6) 98.)

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

- **Analyse** : **ghdl -a circuit.vhd**
- ...
- **Analyse** : **ghdl -a circuit_tb.vhd**
- **Elaboration** : **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
- **Run** : **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, sauveée dans ce fichier.

