Introduction au langage VHDL

Par Thibaut Cuvelier

Date de publication : 03/04/012

Dernière mise à jour : 29 août 2014

Le VHDL est un langage de description du matériel utilisé en électronique numérique. En tant que standard, il est indépendant du logiciel utilisé pour la compilation, la programmation des composants, la simulation, etc.

Commentez

I - VHDL

VHDL est le sigle de *VHSIC hardware description language* ; VHSIC vient quant à lui de *very-high-speed integrated circuits*, une initiative de la Défense américaine dans les années 1980 visant la construction de circuits intégrés très rapides. Le VHDL est un langage de description du matériel utilisé en électronique. En tant que standard, il est indépendant du logiciel utilisé pour la compilation, la programmation des composants, la simulation, etc.

Il autorise plusieurs méthodologies de conception (comportemental, flot de données, structurel) tout en étant d'un très haut niveau d'abstraction en électronique (il est indépendant de la technologie utilisée : FPGA, CPLD, ASIC, etc.). Cette abstraction permet d'ailleurs de le simuler sur ordinateur avant de programmer la moindre puce.

Lors de sa conception, l'objectif n'était pas de produire un nouveau langage *ex nihilo* : la Défense avait récemment mis au point le langage Ada, il était préférable de partir sur ces bases et en exploiter autant que possible la syntaxe, en l'adaptant au contexte de l'électronique numérique.

Utiliser ce langage évite de dériver des tables de vérité des spécifications, que l'on simplifie alors en équations booléennes à implémenter. Il permet d'expliciter la fonction à implémenter sans se soucier de ces détails, bien que l'implémentation en pratique se fera à l'aide de portes logiques (le compilateur se chargera de passer du code aux portes).

Il n'est pas le seul sur le marché : Verilog, par exemple, est également un langage très utilisé, possédant une syntaxe fort différente mais des fonctionnalités équivalentes, si ce n'est égales. VHDL semble plus utilisé en Europe. Une belle proportion des outils disponibles supportent indifféremment les deux.

VHDL et Verilog sont des concurrents de la première heure, bien que Verilog fut le premier sur le marché (vers 1983-1984, par Gateway Design Automation, société qui fut rachetée par Cadence) - il s'agissait d'un standard fermé, certains allant jusqu'à dire que cette fermeture était la seule raison de la création de VHDL. Dès 1983, IBM, Texas Instruments et Intermetrics ont commencé à développer le VHDL ; sa version 7.2 est sortie en 1985 et, deux ans après, fut standardisée par l'IEEE en tant que 1076-1987 (un an après en avoir reçu les droits). En tant que standard IEEE, il doit être revu périodiquement, tous les cinq ans au plus (actuellement, les révisions 1993, 2000, 2002 et 2008 sont sorties).

Ce langage n'est pas dénué d'extensions, il ne se limite pas à la description de circuits numériques. Notamment, **VHDL-AMS** inclut des extensions pour gérer les signaux analogiques et mixtes (*analog and mixed-signal*, AMS), en offrant les mêmes avantages (description de haut niveau, vérification, simulation). Également, **VHPI** (VHDL Procedural Interface) est une interface pour du code C ou C++. Plus étonnant peut-être, **OO-VHDL**, qui, comme son nom l'indique, ajoute des fonctionnalités orientées objet au VHDL, pour une abstraction d'encore plus haut niveau, tout en restant utilisable par du code VHDL (ce qui n'est pas sans rappeler les débuts de C++).

II - Outils

Les fabricants des puces programmables fournissent généralement, sur leur site ou avec les puces, tous les logiciels nécessaires pour les utiliser : compilateur ou synthétiseur VHDL, qui se chargera de convertir le code VHDL en sa version logique, prête pour la puce ; simulateur, pour tester le code.

Notamment, **Lattice Semiconductors** fournit la suite ispLEVER, qui comprend notamment un éditeur basique, mais la licence est également valable pour ActiveHDL, un bien meilleur éditeur et simulateur ; de même, **Xilinx** fournit gratuitement sa suite d'outils. À chaque fois, il faut s'inscrire sur le site pour télécharger les outils.

Pour l'édition simple du code, tout éditeur convient également : Notepad++ et jEdit, de base, disposent d'une coloration syntaxique (quoique pas toujours à jour ou à la hauteur d'un outil dédié à ce langage).

Il est possible de s'armer de manière totalement open source : **GHDL** est un compilateur VHDL basé sur GCC et un simulateur, tandis que **GTKWave** permet d'en visualiser les résultats.

III-A - Types disponibles dans ieee.std_logic_1164

Deux types sont à la base de tout en VHDL : bit et bit_vector. Cependant, ils sont très crus, car seules deux valeurs sont possibles (vrai et faux) : comment représenter une haute impédance, par exemple ? Comment gérer les *don't cares* (dans certains cas, la valeur de sortie n'a aucune importance, on peut en profiter pour optimiser encore plus l'implémentation d'une fonction) ?

Le standard IEEE 1164 vient résoudre ce problème avec les types std_logic et std_logic_vector, respectivement un bit et un vecteur de bits. Les valeurs possibles sont principalement des booléens (vrai 1 et faux 2, d'intensité faible au besoin - L et H), le *don't care* - et la haute impédance Z, utile pour relier plusieurs sorties sans risque d'instabilité du circuit. D'autres valeurs sont également possibles (U pour une valeur non initialisée, X pour une valeur inconnue forte, W pour une valeur inconnue faible), bien que moins utilisées.

Au niveau de la syntaxe, pour indiquer la valeur d'un bit, scalaire, on met le symbole entre apostrophes droites : '1' ; dans le cas d'un vecteur, on utilisera des guillemets droits : "11".

Ces types sont définis dans la bibliothèque ieee.std_logic_1164.all (il faudra le spécifier dans chaque entité VHDL).

III - Premiers éléments de syntaxe

III-B - Paire entité/architecture

Tout programme VHDL contient au moins une paire entité/architecture. L'*entité* définit les entrées et sorties de l'*architecture*, l'implémentation proprement dite. L'entité est vue de l'extérieur du composant (interface) ; l'architecture explicitera les liens entre les entrées et les sorties (implémentation).

Dans l'entité, on doit définir les entrées (in) et sorties (out) ainsi que leur type. On en utilise principalement deux : std_logic et std_logic_vector.

Dans une architecture de type *flot de données*, on définit explicitement les expressions booléennes reliant les entrées et les sorties. Peu importe le style utilisé, l'implémentation produite par le compilateur sera identique si l'architecture fait exactement la même chose.

Les commentaires sont indiqués par -- en début de ligne.

III-B-1 - Premier exemple : un demi-additionneur

On peut mettre tout ceci en pratique en implémentant un demi-additionneur, un composant électronique qui prend deux bits en entrée (qu'il faut additionner) et en produit deux bits (le bit de somme S et le bit de report R).

```
-- Déclaration des bibliothèques utilisées
library IEEE;
use IEEE.std_logic_1164.all;

-- Déclaration de l'entité du demi-additionneur (half-adder).
entity HA is
    port(A, B: in  std_logic;
         S, R: out std_logic);
end HA;

-- Déclaration de l'architecture en flot de données.
architecture arch_HA_flow of HA is
begin
    S <= A xor B;
    R <= A and B;
end arch_HA_flow;
```

Le compilateur optimisera autant que possible ces fonctions, possiblement en créant des fonctions intermédiaires et en les connectant. Il décide également de la manière d'attribuer les pins du composant programmé, à moins que cela ne soit explicité.

```
S = (Sxor);
R = (R_c);
A_c = (A);
B_c = (B);
Sxor.X1 = (A_c);
Sxor.X2 = (B_c);
R_c = (A_c & B_c);
```

III-C - Entité

L'entité sert principalement à définir les entrées et les sorties du composant :

```
entity NAME is
    port(IO);
end NAME;
```

III-C-1 - Attribution des pins

Au besoin, c'est ici qu'on attribuera les pins, quand le comportement par défaut du compilateur ne suffit pas (il assigne les entrées-sorties à des pins de manière relativement aléatoire) ; cette assignation sera beaucoup plus dépendante du matériel que le reste du code.

```
entity NAME is port(
        S: std_logic_vector(1 downto 0)
    );

    attribute pin_numbers of NAME: entity is
        "S(1):24 S(0):12"
end NAME;
```

Un vecteur de bit déclaré comme ci-dessus aura le bit de poids fort en l'indice le plus élevé et le bit de poids faible en zéro.

On peut également définir des valeurs par défaut à l'aide d'une expression et de l'opérateur `:=` *:*

```
entity NAME is port(
        S: std_logic_vector(1 downto 0) := "10"
    );

    attribute pin_numbers of NAME: entity is
        "S(1):24 S(0):12"
end NAME;
```

III-D - Architecture

Une architecture décrit le fonctionnement du système, l'implémentation de la fonctionnalité voulue. Son fonctionnement peut être combinatoire ou séquentiel. Pour modéliser un système complet, on utilisera une série de paires entité-architecture.

On y définit des *signaux*, l'équivalent le plus proche des variables en programmation informatique : ils servent à passer les résultats intermédiaires d'un bloc fonctionnel à un autre. On les utilisera en pratique lors de la présentation des architectures comportementales.

Trois grands formalismes coexistent pour décrire les architectures :

- **flot de données** : on écrit explicitement les fonctions booléennes que l'on veut voir implémentées (à réserver aux plus petits circuits pour des raisons de lisibilité) ; c'est lui qu'on a utilisé pour implémenter le demi-additionneur ;
- **structurel** : on décrit le circuit comme une série de boîtes noires interconnectées au moyen de signaux (utilisé pour des circuits moyens ou grands) ; on procèdera de cette manière pour synthétiser un additionneur complet à l'aide de deux demi-additionneurs ;
- **comportemental** : de manière très semblable à un langage de programmation informatique, on précise le fonctionnement voulu à l'aide d'une suite d'instructions de contrôles plus ou moins évoluées (conditions, boucles, etc.), dans un `process`.

IV - Logique combinatoire

La logique combinatoire est relativement simple à étudier : en fonction d'une série d'entrées, on fournit une suite de sorties, sans que ces sorties dépendent des entrées précédentes. Notamment, on traitera tous les problèmes d'arithmétique, comme l'addition ou soustraction de nombres.

IV-A - Flot de données

Comme précédemment, on décrit simplement les équations booléennes que l'on veut implémenter.

IV-A-1 - Deuxième exemple : un multiplexeur 4-1 en portes logiques

Un multiplexeur est un sélectionneur d'entrées : en fonction de la valeur placée sur les entrées de sélection, il choisit l'entrée de données à répercuter à la sortie. Ici, on implémente un multiplexeur 4-1, car il dispose de quatre entrées de données (donc deux entrées de sélection : $\log_2 4 = 2$) et d'une sortie.

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

entity MUX is port(
    E0, E1, E2, E3, SEL0, SEL1: in std_logic;
    S: out std_logic);
end;

architecture FLOT_MUX of MUX is
begin
    S <= ((not SEL0) and (not SEL1) and E0) or
         (SEL0 and (not SEL1) and E1) or
         ((not SEL0) and SEL1 and E2) or
         (SEL0 and SEL1 and E3);
end FLOT_MUX;
```

IV-A-2 - Troisième exemple : un multiplexeur 4-1 avec un WHEN

On peut cependant améliorer la syntaxe à l'aide de `when`, qui indique la valeur que doit prendre le signal ou la sortie dans tel ou tel cas. Cette écriture est déjà plus lisible, elle correspond beaucoup plus directement à la définition en français du multiplexeur.

Il faut bien faire attention à gérer tous les cas possibles, cela influence fortement la sortie du compilateur. Notamment, dans l'implémentation ci-dessous, si une entrée de sélection est en haute impédance, on choisira la dernière entrée de données, ce qui n'est pas forcément le comportement voulu.

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

```
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity MUX is port(
    E0, E1, E2, E3, SEL0, SEL1: in std_logic;
    S: out std_logic);
end;

architecture FLOT_MUX of MUX is
begin
    S <= E0 when (SEL0='0' and SEL1='0') else
         E1 when (SEL0='1' and SEL1='0') else
         E2 when (SEL0='0' and SEL1='1') else
         E3;
end FLOT_MUX;
```

On peut alors employer un *don't care* pour gérer tous les cas restants, ce qui permet de n'avoir une sortie contrôlée que si les entrées de sélection forment une adresse correcte.

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

entity MUX is port(
    E0, E1, E2, E3, SEL0, SEL1: in std_logic;
    S: out std_logic);
end;

architecture FLOT_MUX of MUX is
begin
    S <= E0 when (SEL0='0' and SEL1='0') else
         E1 when (SEL0='1' and SEL1='0') else
         E2 when (SEL0='0' and SEL1='1') else
         E3 when (SEL0='1' and SEL1='1') else
         '-';
end FLOT_MUX;
```

IV-A-3 - Quatrième exemple : un multiplexeur 4-1 avec un SELECT

On peut encore l'améliorer en restant très proche des équations avec un with-select et une concaténation des deux bits d'entrée de sélection.

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

entity MUX is port(
    E0, E1, E2, E3, SEL0, SEL1: in std_logic;
    S: out std_logic);
end;

architecture FLOT_MUX of MUX is
begin
    with SEL1 & SEL0 select
        S <= E0 when "00",
             E1 when "01",
             E2 when "10",
             E3 when others;
end FLOT_MUX;
```

IV-B - Structurel

On assemble ici une série de boîtes noires déjà écrites pour former un circuit plus complexe.

C'est le principe utilisé pour des additionneurs à plusieurs bits : on branche en série une série d'additionneurs complets, chacun pouvant additionner trois bits. Leur sortie est composée du bit de somme et du bit de report, comme les demi-additionneurs. Alors que le bit de somme est directement transmis à la sortie du circuit, le bit de report est transmis comme troisième bit à additionner à l'additionneur complet suivant.

Cependant, ce principe n'est pas forcément généralisable à grande échelle, puisque chaque additionneur a besoin du bit de report de l'additionneur précédent pour effectuer son travail. Pour additionner n bits, on devra donc attendre n temps de propagation du signal à travers les portes logiques. On peut pallier ce problème, par exemple, en prévoyant le bit de report ou en calculant deux valeurs (quand le bit de report est 1 ou 0).

IV-B-1 - Cinquième exemple : additionneur complet

Ici, pour l'exemple, on se limitera à synthétiser un additionneur complet sur base de deux demi-additionneurs et d'une porte OR.

On aura besoin de deux instances du composant demi-additionneur et d'une seule du composant OR. On note qu'il faut répéter les bibliothèques avant chaque couple entité-architecture.

```
-- Entité demi-additionneur
library ieee;
use ieee.std_logic_1164.all;
entity HA is port(
    A, B: in  std_logic;
    R, S: out std_logic);
end HA;

architecture FLOW of HA is
begin
    S <= A xor B;
    R <= A and B;
end FLOW;

-- Entité OR
library ieee;
use ieee.std_logic_1164.all;
entity P_OR is port(
    C, D: in  std_logic;
    E:    out std_logic);
end P_OR;

architecture FLOW of P_OR is
begin
    E <= C or D;
end FLOW;

-- Entité additionneur complet
library ieee;
use ieee.std_logic_1164.all;
entity FA is port(
    A, B, Ri: in  std_logic;
    R, S:     out std_logic);
end;

architecture FA_ARCH of FA is
    component HA
        port(A, B: in  std_logic;
             R, S: out std_logic);
    end component HA;

    component P_OR
        port(C, D: in  std_logic;
             E:    out std_logic);
    end component P_OR;

    signal S1, S2, S3: std_logic;
begin
```

```
    i1: HA   port map(A,  B,  S1, S2);
    i2: HA   port map(S2, Ri, S3, S);
    i3: P_OR port map(S3, S1, R);
end FA_ARCH;
```

Il importe de bien mettre les mêmes noms de variables lors de l'utilisation des composants que lors de leur création. Lors d'une déclaration d'architecture par structure, les seuls symboles utilisés sont les entrées, signaux et sorties du composant créé, pas les variables déclarées dans l'entité concernant l'utilisation de composants externes.

IV-C - Comportemental

On décrit alors l'architecture comme une suite d'instructions, comme en programmation informatique traditionnelle. On définit des process ainsi que leur *liste de sensibilité* : dès qu'un symbole change de valeur dans cette liste, le code est réexécuté. C'est utile notamment en logique séquentielle, pour lancer le code à chaque coup de l'horloge. Les instructions sont exécutées de manière séquentielle ; cependant, les signaux ne changent de valeur qu'à la *fin* du process.

On peut utiliser diverses constructions bien connues (fort semblables à l'Ada) : le if-then-else, le *case*-is when, etc. On dispose de plusieurs boucles, mais leur utilisation ne correspond pas à l'habitude informatique (elles correspondent plus à une simplification de l'écriture qu'à une exécution de code répétitive pour s'approcher du résultat final ou toute utilisation plus « traditionnelle » des boucles), de même pour les fonctions et procédures.

Les signaux sont définis dans l'en-tête de l'architecture ; les variables ne sont valables que dans un process. En dehors d'un *process*, tout se produit de manière concurrente : deux process peuvent être exécutés simultanément. Par contre, *dans* un process, toutes les instructions sont exécutées de manière séquentielle et les affectations sont réalisées à la fin.

IV-C-1 - Sixième exemple : un multiplexeur comportemental avec IF

On peut implémenter un multiplexeur à l'aide d'une série de if, une manière très commode de procéder pour tout qui vient de la programmation informatique.

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

entity MUX is port(
    E0, E1, E2, E3: in std_logic;
    SEL: in std_logic_vector(1 downto 0);
    S: out std_logic);
end;

architecture CMP of MUX is
begin
    process (E0, E1, E2, E3, SEL)
    begin
        if SEL="00" then
            S <= E0;
        elsif SEL="01" then
            S <= E1;
        elsif SEL="10" then
            S <= E2;
        elsif SEL="11" then
            S <= E3;
        else
            S <= '-';
        end if;
    end process;
end FLOT_MUX;
```

IV-C-2 - Septième exemple : un multiplexeur comportemental avec CASE

On peut y préférer un `case` pour des raisons de lisibilité, cela explicite bien qu'on travaille toujours avec le même vecteur de bits.

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

entity MUX is port(
    E0, E1, E2, E3: in std_logic;
    SEL: in std_logic_vector(1 downto 0);
    S: out std_logic);
end;

architecture CMP of MUX is
begin
    process (E0, E1, E2, E3, SEL)
    begin
        case SEL is
            when "00" => S <= E0;
            when "01" => S <= E1;
            when "10" => S <= E2;
            when "11" => S <= E3;
        end case;
    end process;
end FLOT_MUX;
```

IV-C-3 - Huitième exemple : un décodeur avec FOR

Pour illustrer les boucles, on peut implémenter un décodeur, la fonction inverse du multiplexeur ci-dessus. En d'autres termes, un décodeur est un générateur de mintermes : en fonction de ses entrées, seul un bit de sortie sera activé.

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

entity DEC is port(
    E0, E1: in std_logic;
    S: out std_logic_vector(3 downto 0));
end;

architecture FLOT_MUX of DEC is
begin
    process (E0, E1)
    variable N := integer;
    begin
        N := 0;

        if E0 = '1' then N := N + 1; end if;
        if E1 = '1' then N := N + 2; end if;
        S <= "0000";

        for I in 0 to 3 loop
            if I = N then
                S(I) <= '1';
            end if;
        end loop;
    end process;
end;
```

V - Logique séquentielle

Simplement, on passe en logique séquentielle quand on ajoute une horloge en entrée et qu'un bout de code est exécuté à chaque flanc montant/descendant. On se basera donc sur une expression comportementale de

l'architecture, il n'est pas possible de procéder autrement. Les sorties du circuit dépendront des entrées actuelles ainsi que de leurs valeurs précédentes.

Par exemple, la logique séquentielle sera utilisée pour une lampe clignotante (à chaque coup d'horloge, on inverse la sortie, afin que la lampe passe de l'état allumé, puis éteint, puis allumé car on retourne à l'état initial) ou pour tout circuit un tant soit peu plus compliqué (comme un processeur : à chaque coup d'horloge, il faut réaliser une série d'opérations dictées par l'instruction en cours d'exécution ; quand on a fini de traiter une instruction, on passe à la suivante ; à chaque fois, il faut mémoriser à quel endroit on est arrivé dans le traitement d'une instruction et dans le code à exécuter).

V-A - Échauffement : le flip-flop D

À titre d'exemple, on peut modéliser un flip-flop (ou bascule ou verrou) de type D, avec reset synchrone et asynchrone. Quand l'horloge est à 1, on recopie l'entrée D à la sortie Q ; quand l'horloge est à 0, on mémorise la valeur D. Le reset permet de mettre le flip-flop dans un état stable et connu, ce qui n'est pas forcément le cas à la mise sous tension.

V-A-1 - Neuvième exemple : un flip-flop D synchrone

De manière synchrone, on vérifie qu'on est bien à un flanc montant (ou descendant, selon la convention souhaitée ; on aurait alors utilisé `falling_edge`) avant de faire quoi que ce soit.

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

entity FFD is port(
    D, RAZ, CLK: in std_logic;
    Q: out std_logic);
end;

architecture ARCH_FFD of FFD is
begin
    process(CLK, RAZ)
    begin
        if rising_edge(CLK) then
            if RAZ = '1' then
                Q <= '0';
            else
                Q <= D;
            end if;
        end if;
    end process;
end ARCH_FFD;
```

V-A-2 - Dixième exemple : un flip-flop D asynchrone

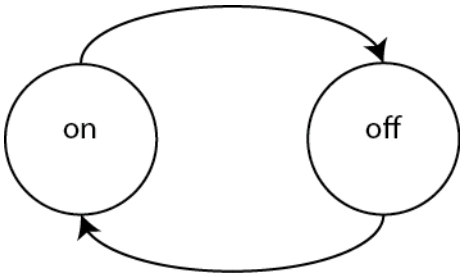
De manière asynchrone, le reset se passe toujours, tandis que l'entrée n'est recopiée dans le flip-flop qu'au flanc montant.

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

entity FFD is port(
    D, RAZ, CLK: in std_logic;
    Q: out std_logic);
end;

architecture ARCH_FFD of FFD is
begin
    process(CLK, RAZ)
```

```
    begin
        if RAZ = '1' then
            Q <= '0';
        elsif rising_edge(CLK) then
            Q <= D;
        end if;
    end process;
end ARCH_FFD;
```

V-B - Machine à états

Une machine à états est l'implémentation en électronique du concept d'automate : dans un circuit séquentiel, on mémorise une certaine forme des entrées précédentes (l'état) qui vont influencer la sortie courante. Par exemple, pour le circuit contrôlant une lampe clignotante, on aura une machine à états très simple : seuls deux états sont possibles (lampe allumée ou éteinte), on passe de l'un à l'autre à chaque coup d'horloge.



Ainsi, si on veut implémenter une machine à états, il suffit de définir une série de bits définissant l'état courant ; ainsi, à chaque coup d'horloge, on exécute un process qui va varier en fonction de l'état courant. On préférera utiliser un signal, étant donné qu'il n'est pas utile de transmettre cette valeur à l'extérieur ou de laisser la possibilité de l'altérer directement.

On peut donc implémenter cette lampe en VHDL. La machine à états ne contiendra que deux états et un bit suffira pour le retenir ($\log_2 2 = 1$). La sortie O varie en même temps que le flanc montant d'horloge et servira à actionner la lampe (ce qui est fait du signal en sortie est à gérer manuellement à l'aide de DEL et résistances, en se basant sur les spécifications du composant utilisé notamment en ce qui concerne le courant de sortie).

```
entity NAME is port(
    CLK: in  std_logic;
    O:   out std_logic;
end;

architecture arch_NAME of NAME is
signal state: std_logic;
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            if state = '0' then
                O <= '1';
                state <= '1';
            else
                O <= '0';
                state <= '0';
            end if;
        end if;
    end process;
end arch_NAME;
```

Si on veut utiliser la valeur d'une sortie comme base dans un bout de programme, ce n'est pas possible : il faut définir un signal qui sera recopié dans la sortie, de manière asynchrone. Pour ce faire, on met l'instruction de recopie dans la sortie en début d'architecture, avant de définir le moindre process.

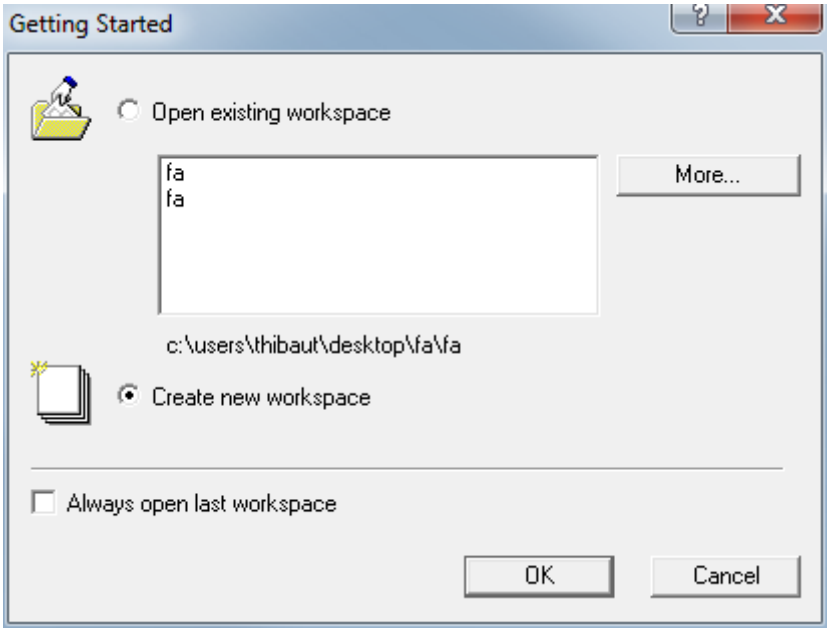
```
architecture Arch of Sth is
```

```
    signal sOut: std_logic;
begin
    Out <= sOut;

    process(CLK)
    begin
        -- ...
        -- On utilise sOut là où on voulait un accès à Out ;
        -- on donne des valeurs à sOut là où on assignait Out.
    end process
end Arch;
```

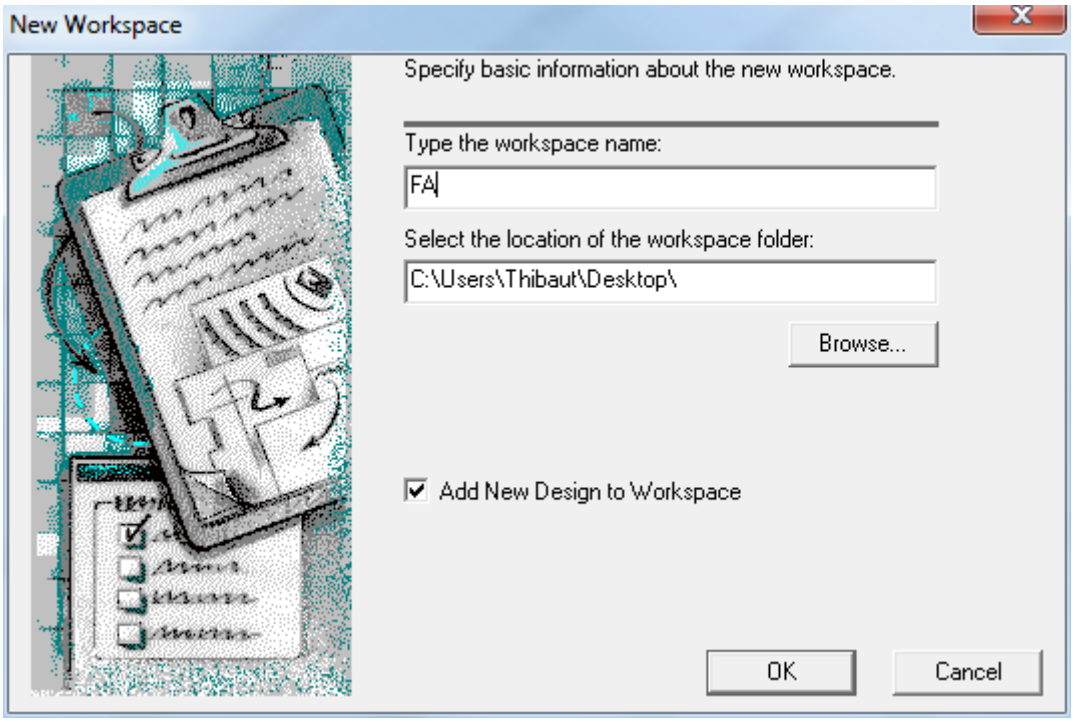
VI - Simuler un circuit

On utilise dans cette section Active-HDL 8.2, bien que la manipulation puisse être totalement différente dans un autre logiciel.

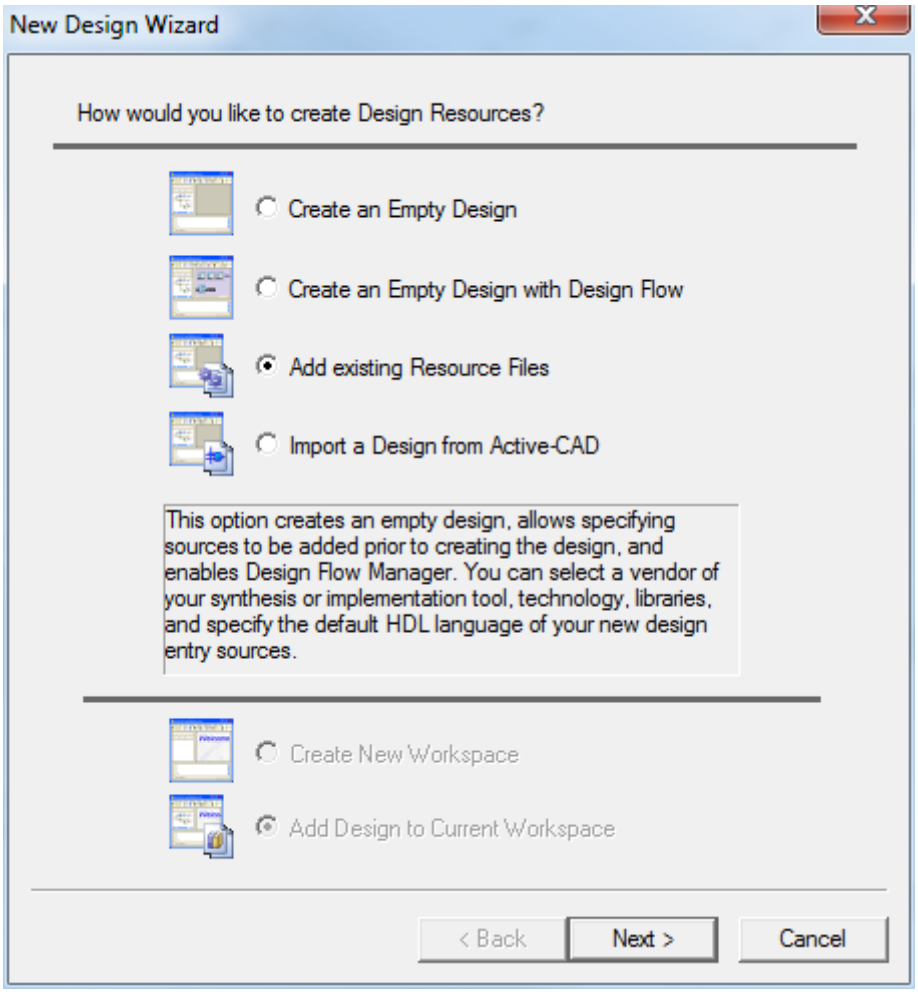
Après avoir lancé le logiciel, créer un nouveau design et ajouter un fichier précédent (ici, on utilise le code d'un additionneur complet). Au lancement du logiciel, choisir la création d'un nouvel espace de travail et profiter de l'option de création d'un nouveau design (accessible depuis le menu *File > New*). L'option *Add existing resource files* permet d'insérer de nouveaux fichiers dans ce design ; après avoir sélectionné le fichier, la page suivante demande de spécifier le langage de description (VHDL).



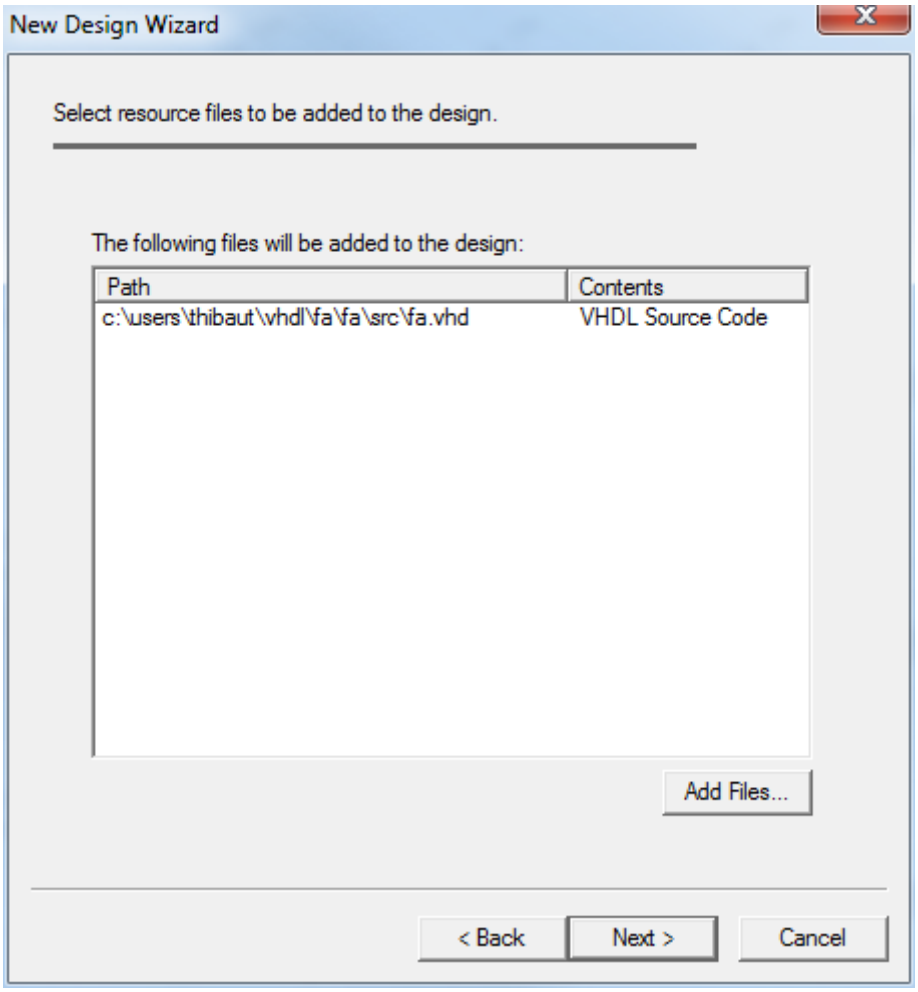
Écran d'accueil.



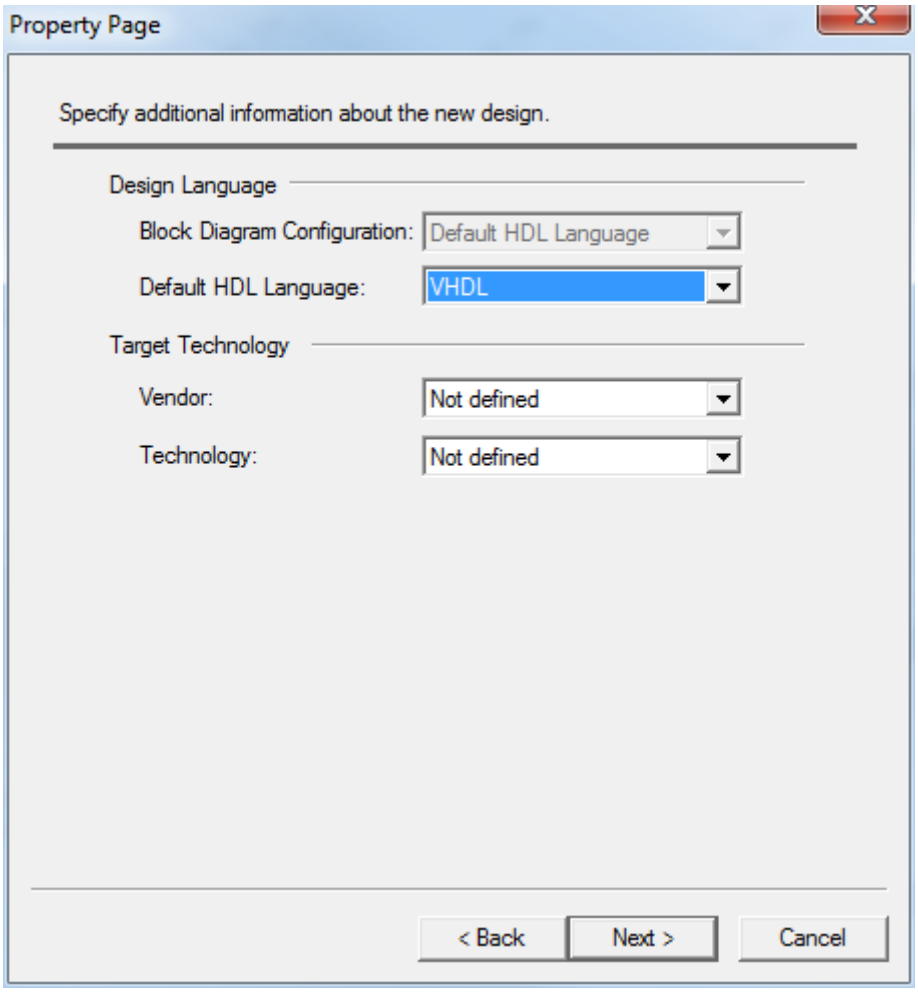
Création d'un espace de travail.



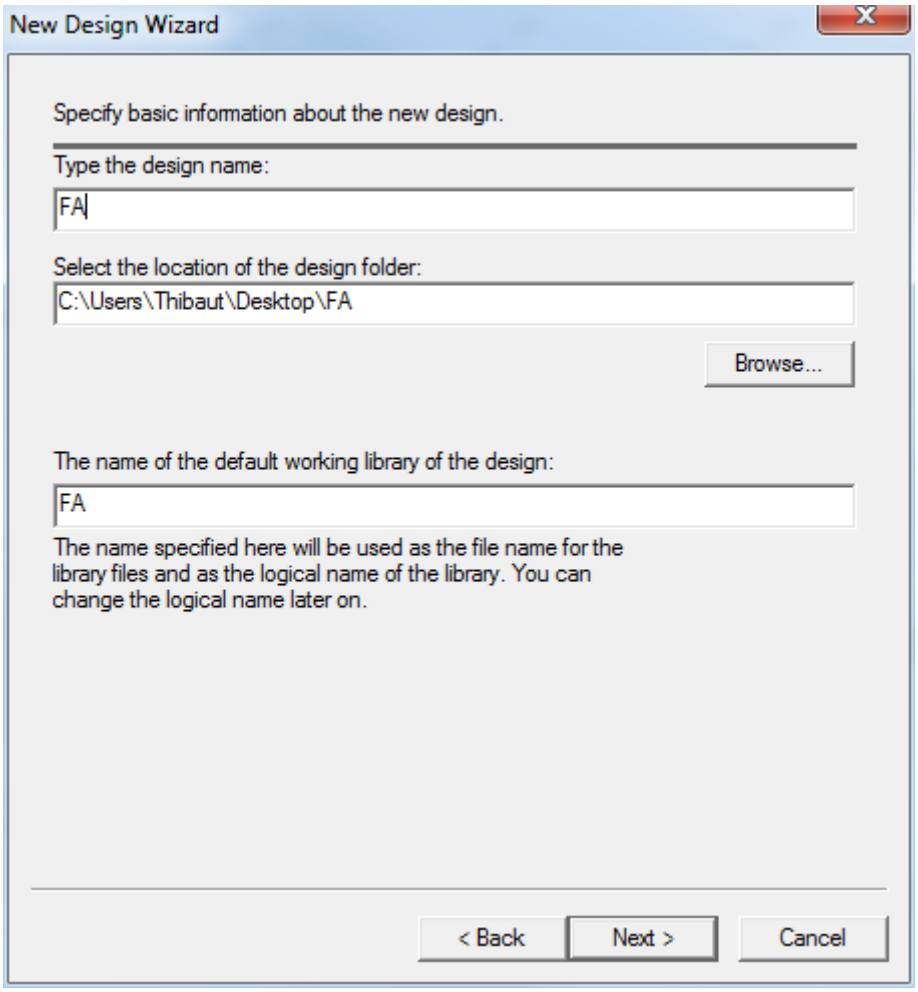
Assistant de création de design.



Fichiers à ajouter.



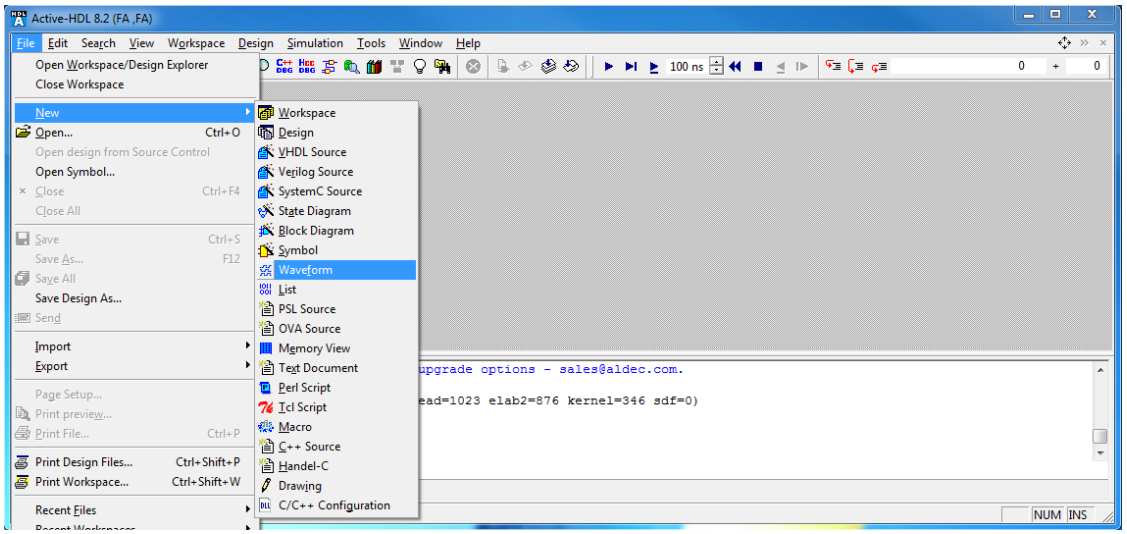
Choix du langage.



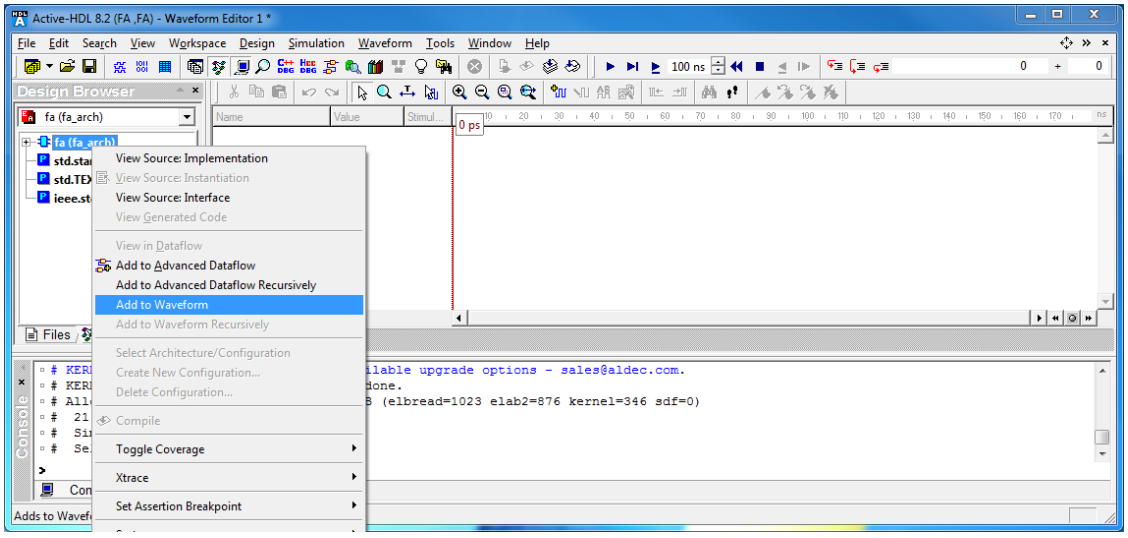
Choix du nom du design.

L'espace ainsi créé, il faut initialiser une simulation (*Simulation* > *Initialize simulation*), créer des signaux de stimulation (*File* > *New* > *Waveform*). Depuis l'explorateur de design, ajouter l'architecture à simuler (dans le menu contextuel, choisir *Add to Waveforme*). Pour chacun des signaux apparus, il faut ajouter un stimulateur (menu contextuel, *Stimulators*). Par simplicité, on utilise des horloges de fréquences différentes.

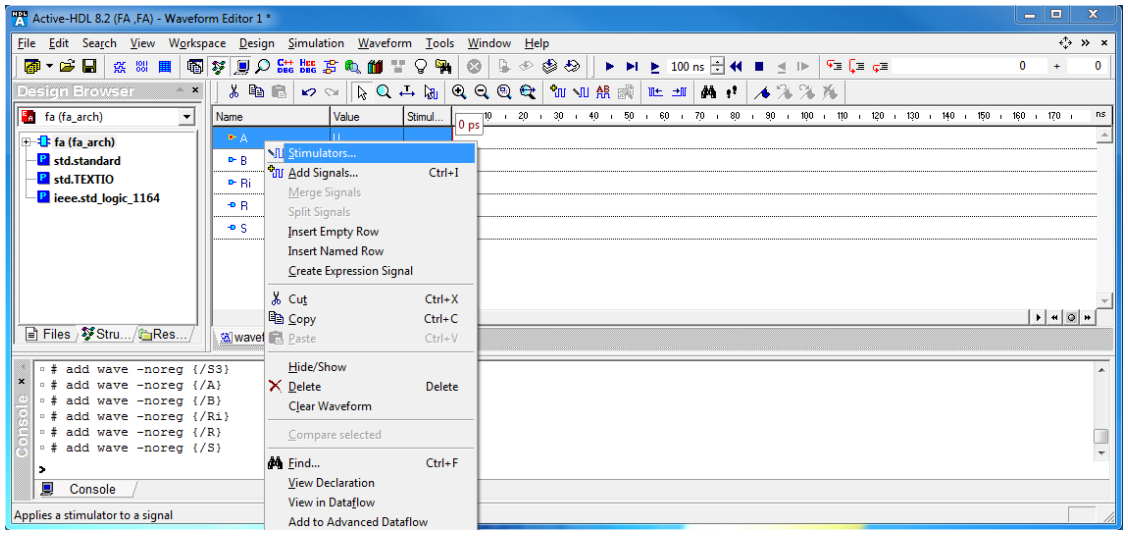
Initialisation de la simulation.



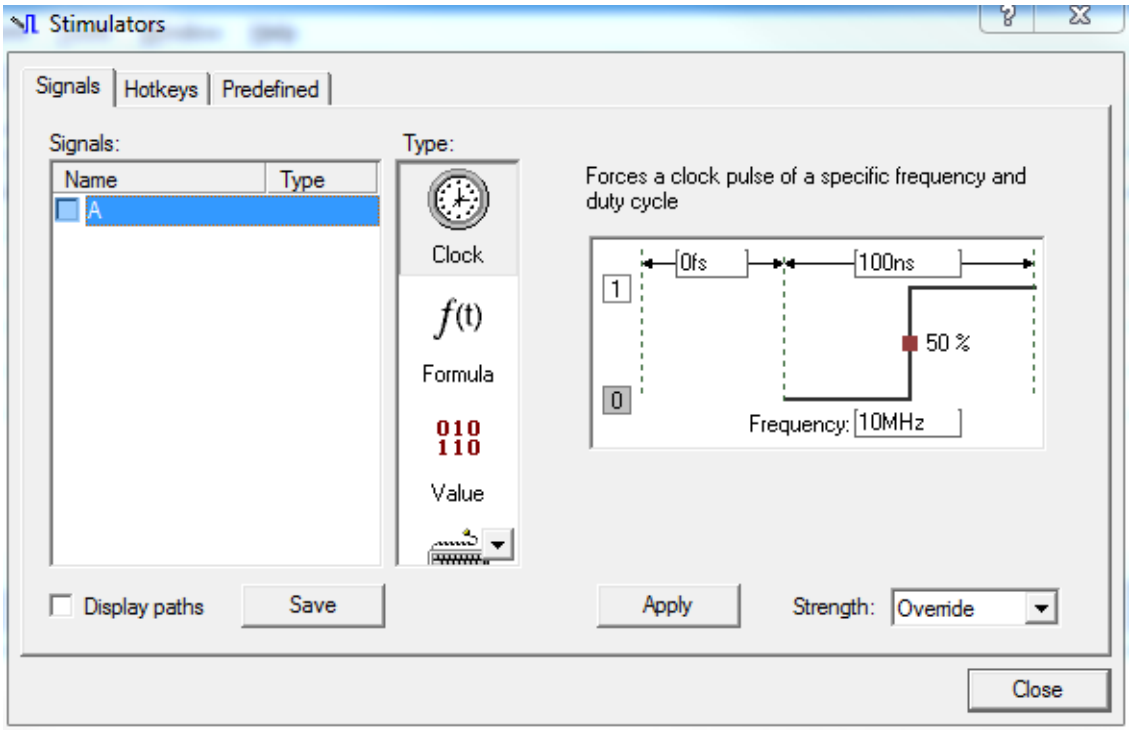
Création d'un signal de stimulation.



Ajout de l'additionneur complet à la simulation.

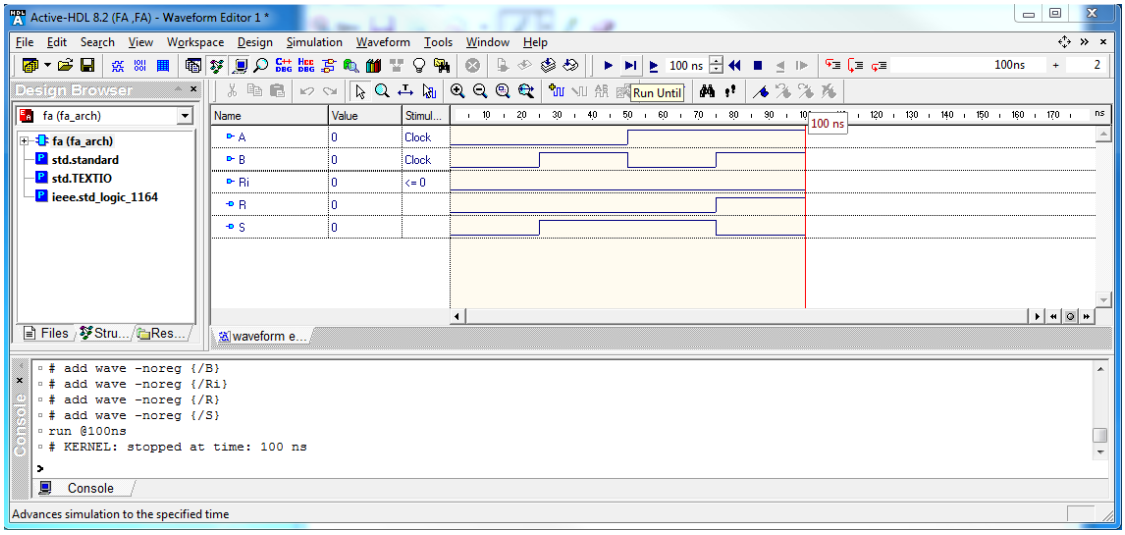


Ajout de stimulateur sur une variable.



Choix du stimulateur.

On obtient ainsi un signal en sortie qui additionne les trois bits plus haut. On a fixé le bit de report à zéro tout le temps.



Résultat de la simulation.

VII - Remerciements

Merci à **Nicolas Vallée** pour son aide à la rédaction, à **vermine** pour ses encouragements et à **Claude Leloup** pour sa relecture orthographique !